Pri
to
to
exc
pei

1.

2.

Pl
th

[

3.

Th
be
w

4.

5.

Th
op
to

ADDENDUM

The following sentence should be appended to paragraph 2 of Section 7.8.1 on page 175:

In these experiments the creation of join groups preceded the creation or extension of temporal chains in the major cycle. This biases temporal chain construction to be based upon more specific joins rather than general joins.

An Architecture for Situated Learning Agents

by

Matthew Winston Mitchell, BComp (Hons)

Dissertation

Submitted by Matthew Winston Mitchell

for fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in the School of Computer Science and Software Engineering at

Monash University

Monash University

August, 2003

Contents

List of Tables ix

List of Figures xi

Abstract xv

Acknowledgments xviii

1 Introduction **1**
- 1.1 Situated Agents 1
- 1.2 Requirements for implementing Situated Learning Agents 2
- 1.3 Approaches to Achieving the Requirements 3
- 1.4 Techniques 4
  - 1.4.1 Established techniques 4
  - 1.4.2 Novel Techniques 8
- 1.5 System Evaluation 10
- 1.6 Related Research 11
- 1.7 Contributions 12
- 1.8 Thesis Outline 13

2 Learning **14**
- 2.1 The Role of Memory 14
- 2.2 Using Memory 14
- 2.3 Markov Decision Problems 17
- 2.4 Reinforcement Learning 17
  - 2.4.1 Exploration 19

- 2.5 Identifying States 20
- 2.6 Learning Value-functions for Generalised Representations 21
- 2.7 Hidden-state Problems 22
- 2.8 Summary of Two Approaches to Hidden-state 23
- 2.9 Using Belief States 24
  - 2.9.1 Learning Values for Belief Spaces 25
- 2.10 Markov-k Memory Approaches 26
  - 2.10.1 Fixed-length Memories 27
  - 2.10.2 U-Tree 27
  - 2.10.3 Finite-state Automata 28
  - 2.10.4 Recurrent Networks 28
- 2.11 Indefinite Memory 29
  - 2.11.1 Schema Mechanism 29
  - 2.11.2 Classifier Systems 30
- 2.12 Alternative methods for Learning Policies 30
  - 2.12.1 Memoryless Policies 30
  - 2.12.2 External State 30
  - 2.12.3 Evolutionary Search 31
  - 2.12.4 HQ-Learning 31
  - 2.12.5 Levin Search 31
  - 2.12.6 Heuristic Strategies 31
- 2.13 Input Generalisation 32
- 2.14 Exploration 33
- 2.15 Hypothetical Look-ahead 34
- 2.16 Multiple Goals and Tasks 35
- 2.17 Summary 35

3 System Overview **37**
- 3.1 Model Learning 38
  - 3.1.1 The N-maze Problem 38
  - 3.1.2 Learning Transitions 39
- 3.2 Temporal Chains 39

3.2.1 Chain Composition 40
3.2.2 Chain limitations 40
3.2.3 Using Reinforcement 42
3.2.4 Stochastic Problems 42
3.2.5 Summary of Temporal Chains 43
3.3 Using Joins 43
3.3.1 Limitations of Unary Groups and Nodes 44
3.3.2 Effects of Introducing Join Groups 45
3.3.3 Suppression of Subordinates 46
3.3.4 Computational Completeness 47
3.3.5 Summary of Join Groups and Nodes 47
3.4 Visualising Groups and Nodes 48
3.5 Policy Learning 48
3.5.1 The Effects of Policies 49
3.5.2 Improving Predictions 50
3.5.3 Relating the Model to the Policy 51
3.5.4 Hypothetical Look-ahead 51
3.6 Chapter Summary 52

4 The Basic System 54
4.1 High-level Description 55
4.1.1 Groups 55
4.1.2 Messages and Links 56
4.1.3 Action Selection 57
4.1.4 The Role of Groups and Nodes 57
4.2 System Operation 58
4.2.1 A Simple Maze 59
4.2.2 Initial groups created for the sample problem 60
4.2.3 Join Groups 61
4.2.4 Hierarchies and Chaining 62
4.2.5 Representing NOT 62
4.2.6 Efficiency of Search and Representation 63

4.2.7 Calculating ETP and Utility Estimates 66
4.2.8 Allowing Sufficient Trials for Returns 70
4.2.9 Allowing Sufficient Trials for Comparisons 71
4.2.10 A Problem with Measuring Improvements 72
4.2.11 Dealing with Oscillating Values 72
4.2.12 Local Minima 74
4.2.13 Effector Selection 75
4.2.14 Hypothetical Look-ahead 76
4.3 Diagrams of the Basic System 77
4.4 Steps in the Basic System 80
4.5 Chapter Summary 82

5 Experiments using the Basic System 83
5.1 Introduction 83
5.2 Generalisation Tasks 83
5.2.1 The Monk's Problem 84
5.2.2 Experimental Methodology 85
5.2.3 Validating the Default Training Set 86
5.2.4 Input Representation 86
5.2.5 Input Generalisation Parameter Settings 88
5.2.6 Evaluation Criteria 88
5.2.7 Basic Results 88
5.2.8 Comparing Predictive Accuracy 89
5.2.9 Comparing Training Time 90
5.2.10 Comparing Solution Size 92
5.2.11 Summary of Comparisons 93
5.3 Qualitative Analysis of TRACA's Generalisation Results 93
5.3.1 Comparing TRACA's Performance without Joins 94
5.3.2 Analysis of Monk 1 Performance 94
5.3.3 Analysis of Monk 2 Performance 98
5.3.4 Analysis of Monk 3 Performance 101
5.3.5 Summary of Performance Analysis 101

5.4 Effects of Parameter Changes on Generalisation Performance 104
  5.4.1 Effects of Varying the Exploration Rate 105
  5.4.2 Effects of Varying the Learning Rate 107
  5.4.3 Effects of Varying the Number of Statistical Cases 108
  5.4.4 Changing the Action Selection Strategy 110
  5.4.5 Making Utile rather than Perceptual Comparisons 111
5.5 Discussion of Generalisation 112
5.6 Rule Chaining 112
  5.6.1 Experimental Design 113
  5.6.2 Results on the Grid 114
  5.6.3 Adding Effector Noise 115
  5.6.4 Problems with Rule Chaining 115
5.7 Hypothetical Look-ahead 117
  5.7.1 Experimental Design 117
  5.7.2 Results 118
5.8 Chapter Summary 119

6 Adding Temporal Chains 120
6.1 Representing Hidden-state 120
6.2 Hidden-state Region Types 121
6.3 The Search Strategy 123
  6.3.1 Problem Search Space 125
  6.3.2 Implementing the Search Strategy 127
  6.3.3 A Possible Alternative Approach 127
  6.3.4 Parallel Rule Operation 128
6.4 Implementing the Approach 129
  6.4.1 Temporal Chain Creation 130
  6.4.2 Temporal Chain Extension 131
  6.4.3 Search Restriction 132
  6.4.4 Summary of Chain Operation 135
6.5 Difficulties in Building Temporal Chains 136
  6.5.1 Representing Homogeneous Regions 137

  6.5.2 The Importance of Actions 140
  6.5.3 Selecting Features for Chain Extension 142
6.6 An Example of Representing Hidden-state 144
  6.6.1 The Example Problem 144
  6.6.2 Initial Representation 145
  6.6.3 Improving Predictions with Temporal Chains 146
  6.6.4 Hierarchical Relationship 148
  6.6.5 Temporal Value Estimates 149
  6.6.6 Controlling Temporal Chain Extension 151
  6.6.7 Completing the Chain 151
  6.6.8 Representing Alternative Paths and Outcomes 153
6.7 Steps for Creating and Using Temporal Chains 154
6.8 Chapter Summary 156

7 Experiments using Temporal Chains 157
7.1 Experimental Methodology 157
  7.1.1 Dimensions of Hidden-state Problems 157
  7.1.2 Performance Metrics 158
  7.1.3 Training and Testing Procedures 159
  7.1.4 Parameter Settings 160
7.2 Gap Tasks 160
  7.2.1 Experimental Design 162
  7.2.2 Results 162
7.3 McCallum's M-maze 163
  7.3.1 Experimental Design 164
  7.3.2 Results 164
7.4 Noisy Sensors and Effectors 165
  7.4.1 Experimental Design 165
  7.4.2 Results 165
7.5 Irrelevant Noisy Features 166
  7.5.1 Experimental Design 166
  7.5.2 Results 167

7.6 Ring's Mazes 167
7.6.1 Experimental Design 168
7.6.2 Results on Ring's 5x4 Maze 169
7.6.3 Results on Ring's 9x9 Maze 170
7.7 Discussion of Results on Maze Tasks 173
7.8 Truck Driving 173
7.8.1 Experimental Design 174
7.8.2 Results 175
7.8.3 Discussion of Results 176
7.9 Chapter Summary 178

8 Conclusion 179
8.1 Comparison with Related Systems 179
8.1.1 Utile Distinction Approaches 179
8.1.2 Schema Mechanism 180
8.1.3 Learning Classifier Systems 181
8.2 Evaluation of TRACA 182
8.3 Contributions 185
8.4 Future Work 187
8.4.1 Further Experiments with the Existing System 187
8.4.2 Possible Extensions to the Existing System 188

Appendix A Skeleton Pseudo-code for the Basic System 190

Appendix B Sample Structures Created By TRACA 202
B.1 Structure Created During One Run on the 9x9 Maze Problem. 202
B.2 Sample Structures Created for Truck Driving Problem. 203

List of Tables

1.1 Tasks used in experiments and the system capabilities they demonstrate. 11

5.1 Performance - Monk 1. 89
5.2 Performance - Monk 2. 89
5.3 Performance - Monk 3. 89
5.4 Comparison of percentage predictive accuracy for Monk problems. 90
5.5 Comparison of training times required for Monk problems. 91
5.6 TRACA's predictive accuracy when varying the training time for Monk 1 and 3. 92
5.7 Comparison of structures created for the Monk tasks. Decision tree sizes are reported as (number of nodes, number of leaves). The smallest number of nodes for each task is indicated in bold. 93
5.8 Comparison of predictive performance by TRACA on the Monk tasks with and without the creation of join groups (join creation is turned off). 94
5.9 Performance on Monk 1 when varying exploration rate (averages of 20 runs). . 105
5.10 Performance on Monk 1 when varying the learning rate (averages of 20 runs). . 108
5.11 Performance on Monk 1 when varying the number of cases for the test for noise (averages of 20 runs). 109
5.12 Performance on Monk 1 when varying the number of cases for the Cox-Stuart test for trend (averages of 20 runs). 109
5.13 Performance on Monk problems when changing the action selection strategy (averages of 20 runs). 110
5.14 Performance on Monk problems based on utile improvements. 111

6.1 Observations in order of appearance for each of the tape sequences. The first observation in a trial is one of the letters D, E or F all of which are followed by A and then either X or Y, which can be predicted by remembering the first letter of each sequence. 125

6.2 The sequences of observations for the three new tapes. In each trial, the system starts at t_1 and then moves through t_2 and and t_3 with each read action. The final observation at t_4 can be predicted by remembering the initial observation. 129

6.3 The approximate transition histories (ETPs) calculated by unary nodes (in groups representing each observation) to subsequent observations. As X occurs only a third of the time A has a value of 0.333, and as Y occurs two thirds of the time it has a value of 0.667. 130

6.4 A tape reading problem with two possible sequences of observations containing homogeneous hidden-state. 138

6.5 The two sequences for a letter prediction problem with gap 2. In each trial the letters are presented in order from t_1 to t_4. 145

7.1 Comparative performance on the homogeneous and heterogeneous gap tasks. . 163

7.2 Training times and structure created for maze tasks. 173

7.3 Exploration rate for different timesteps during learning. 175

7.4 The comparative performance of TRACA on the truck driving task, in terms of number of collisions, to other possible policies. The lowest number of collisions obtained by each technique is highlighted in bold italics and the averages on the bottom row are rounded. 176

List of Figures

3.1 A simple N-maze with hidden state. 38

3.2 A group with a single unary node to record a transition estimate from label 4 to label 6. 39

3.3 Internal state memory required for east branch of N-maze. 41

3.4 Internal state memory required for west branch of N-maze. 41

3.5 The four corridors of the new maze problem. 44

3.6 Transitions occuring in the modified maze. 44

3.7 Transition estimates calculated using only unary groups. 45

3.8 Effects of a join group on the transition estimates of nodes in its subordinate groups. 46

3.9 Individual groups and combinations of groups required to solve the maze. . . . 46

3.10 The group encapsulating two unary nodes created for label 7. 48

3.11 A hierarchy with a single join group. 49

4.1 System Overview 56

4.2 TRACA's Match-Fire-Execute major cycle 59

4.3 The sample maze. 60

4.4 Sequence of predictor nodes forming for the maze problem. In each sub-figure, shading indicates detectors matched in one cycle (left), the selected effector (center), and the detectors matched with the following input string (right). . . 60

4.5 Use of a join group to create an AND construct. 62

4.6 Direct chaining of a node in group X activated by environmental input (at t_n) to group Y activated by the subsequent input (at t_{n+1}). 63

4.7 Many systems are constructed using rules whose conditions include every bit of input strings received from the environment. 64

4.8 Possible hierarchies of joins when using explicit negation. 65

4.9 Comparison of internal models for Q-learning. 67

4.10 A sample problem for Q-value updates. 67

4.11 Two groups each containing two nodes. The numbers in the nodes are their utility estimates, the numbers at the end of the arrows are the returns sent by each group to predicting groups. 70

4.12 Depiction of oscillating values in a subordinate node and its equivalent superior (same effector and prediction). 73

4.13 Class diagram showing class relationships for the basic system 78

4.14 Sequence diagram for the system's major cycle 79

5.1 A sample input string and the bit vector format for the Monk problem. 87

5.2 Join groups required to represent the true concepts for Monk 1 (dashed lines show the borders of each attribute's bit positions). 95

5.3 Unsuspended join groups actually created to represent the concepts for Monk 1 in one run in which 100 percent test accuracy was achieved (groups are labelled in order of creation). 96

5.4 Network growth during one run on Monk 1. Training continues for the first 3720 timesteps (30 epochs) and during this time random actions are selected with probability 0.25. Once training is completed, learning and random action selections are turned off. 97

5.5 Join groups required for just one pair of attributes to solve the Monk 2 problem (dashed lines show the borders of each attribute's bit positions). 98

5.6 Network growth during one run on Monk 2. Training continues for the first 8450 timesteps (50 epochs) with random actions selected with probability 0.25. Once training is completed, learning and random action selections are turned off. . . 99

5.7 Joins formed in one run on Monk 2 (dashed arrows show the borders of each attribute's bit positions). 100

5.8 Join groups required to represent the true concepts for Monk 3 (dashed lines show the borders of each attribute's bit positions). 102

5.9 Some of the unsuspended join groups actually created in one run to represent the concepts for Monk 3 (labelled in order of creation). 102

5.10 Network growth during one run on Monk 3. Training continues for the first 4880 timesteps (40 epochs) with random actions selected with probability 0.25. Once training is completed, learning and random action selections are turned off. . . 103

5.11 The 4x4 grid problem. 113

5.12 Results on the 4x4 grid. 114

5.13 Sample of structures developed by TRACA for the 4x4 grid. Unary groups are indicated as broken circles and join groups as unbroken circles. The arrows indicate the subordinates of join groups. 116

5.14 The 4x4 grid for look-ahead and its input strings for each location. 117

6.1 Hidden-state regions (corridors) with a heterogeneous input pattern. 122

6.2 Hidden-state regions (corridors) with a homogeneous input pattern. 123

6.3 Two graphs before and after hidden-state is uncovered. 126

6.4 Valid and invalid search spaces in the solution tree for X in the problem from Table 6.1. 126

6.5 A probabilistic finite-state automata where the transition from A to X or Y is entirely stochastic. 132

6.6 In this figure, the temporal chain does not help solve the letter prediction problem because it has not been extended to include D. However, the primary node in this chain will appear a better predictor than its equivalent subordinate node because it makes an incorrect prediction of X less often ($\frac{1}{2}$ versus $\frac{1}{6}$). 133

6.7 A chain with three nodes (labelled 1, 2 and 3) in the terminal link group (labelled L4) for the three different actions (labelled x, y and z) which have been taken after the chain was matched during different trials. The timesteps shown as t_1 to t_5 are each divided to indicate the *match* and *fire* steps of TRACA's major-cycle.137

6.8 The search tree for the observation X in Table 6.4 showing the path required to be searched to construct a temporal chain to reliably predict X. 138

6.9 Three different stages of chain construction and extension for a sequence to represent the path from D to X. 139

6.10 Two possible paths of the same length which lead to the same final state. . . . 141

6.11 Two possible paths of the same length which do not lead to the same final state. 142

6.12 Hidden-state regions (corridors) with a homogeneous input pattern and the input strings experienced during one trial. The right-most bit indicates the presence of a person in the positions prior to the end of each corridor (with 0.25 probability). The node in the join group (representing the surrounding walls) predicts the door and will have an ETP of approximately 0.1. The node in the unary group representing the person also predicts the door and will have an ETP of 0.5. However, the precedence rate for the node in the join group is 1.0 while the precedence rate for the node representing the person is 0.25. 143

6.13 All the non-temporal nodes and groups developed. The group matched at t_4 in a trial could be either C or D depending on the sequence presented. Bold structures show one path which might be taken during a trial. 146

6.14 An alternative representation of the structures for one path in the gap problem. 147

6.15 The commencement of the temporal chain T and the utility values converged to. 150

6.16 Alternative representation of one path along the temporal chain T. T3 is depicted as hierarchically higher, or superior, to group B. 151

6.17 The Chain T after extension along with the utility values converged to. 152

6.18 Alternative representation for one path along extended temporal chain. 153

7.1 The mazes used in the gap experiments. 161

7.2 The M-maze showing the observations and goal state. 164

7.3 Ring's 5x4 maze. 169

7.4 Ring's 9x9 maze. 170

7.5 Ring's 9x9 maze with orientation incorporated in inputs. The number at the top of each box is the input received when the agent is facing north, the number at the bottom the input received when facing south, and the numbers on the left and right are received when facing west and east respectively. Each number is represented as a single unique bit in TRACA's input string. 171

7.6 Number of training trials required by the 97 successful TRACA agents on Ring's 9x9 maze task. 171

7.7 Comparison of structures to solve 9x9 maze problem. 172

An Architecture for Situated Learning Agents

Matthew Winston Mitchell,
Monash University, 2003

Supervisors: Ann Nicholson, David Albrecht.

Abstract

Situated learning agents are agents which operate in real-world environments. Ideally such agents should be capable of assisting humans by performing complex tasks which involve drudgery or risk. Such agents must be capable of dealing with noisy, non-deterministic environments with large state spaces often requiring various forms of memory.

This thesis addresses the problem of situated learning agents. It draws from the lessons of related work in the area to identify three fundamental requirements to aid in making the complex choices and trade-offs which arise when addressing this problem. Based on the three requirements a number of existing techniques for learning are selected for use in a new system which is appropriate for implementing situated learning agents. Within this new system the selected techniques are augmented with a variety of important novel techniques.

The resulting system is a reinforcement learning system which dynamically develops a connectionist model of its environment while learning. This model consists of *join* groups and *temporal* groups. Join groups are used to address the input-generalistion problem by constructing general rules using a default hierarchy, and temporal groups address the hidden-state problem by implementing a short-term memory mechanism. Groups represent one or more situations in the agent's environment and are connected to detector inputs and/or other groups by arcs which are used to pass a variety of messages. Based on the situations they represent, groups contain nodes which store estimates of action-values and maintain estimated transition probabilities to other situations. New groups are created incrementally while learning and are introduced by joining them to two existing groups as selected by a localised probabilistic mechanism. Each new join group is given a small number of trials to determine its usefulness and is then retained only while its nodes demonstrate improved transition estimates over the nodes in the two groups it joins.

Among the distinguishing features of the proposed system is an ability to reduce the complexity of structures by representing logical NOT using only AND combinations. This is achieved through the organisation of nodes into groups along with a suppression mechanism. When compared to Back-propagation neural networks, vertices in the proposed system store transition and value estimates relatively independently, allowing it to exploit learning from fewer training examples. This independence also avoids common problems with distributed representations such as interference and catastrophic forgetting, but at the expense of a larger internal representation for some problems.

When dealing with problems containing hidden-state, which require short-term memory to be solved, the system will not continue to expend resources extending memory if the solution provides no useful improvement in achieving reinforcement. This, combined with a depth-first search approach to constructing memory, avoids the problem of choosing to have either a fixed-size history window or a-priori restrictions on the amount of structure used for creating short-term memory. However, a requirement for the proposed system is that a common set of paths is frequently traversed during training. Experiments in spatial navigation environments demonstrate how this requirement can commonly be met without difficulty by manipulating the reward landscape.

The system's representation and creation of new groups has strong parallels to both Holland's Learning Classifier Systems and Drescher's Schema Mechanism (Holland 1975; Drescher 1991). Consequently, the system is capable of discovering and representing a large number of rules efficiently using default hierarchies. Furthermore, the rule set can grow with rules continually being added as more experience is obtained or new problems encountered. However, in place of the genetic algorithm used by Learning Classifier Systems for rule discovery, the proposed system makes random combinations using a number of selective mechanisms to reduce the search space. Structures created by the system are incrementally combined to create more complex structures. This method of creating rules is in contrast to Drescher's Schema mechanism in which new rules require the evaluation of a large number of inputs together.

Experimental results are presented which demonstrate clearly that the system described in this thesis is capable of dealing with the difficulties that arise in real-world environments, particularly in relation to input-generalisation and hidden-state. The experiments are based on well-known and commonly used problems from the literature, including concept learning and maze navigation tasks. The results demonstrate that the proposed system performs as well or better than many of the compared approaches in terms of predictive accuracy and the number of training examples required.

An Architecture for Situated Learning Agents

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

[redacted]

Matthew Winston Mitchell
August 27 2003

Acknowledgments

The research in this thesis began in 1996 under the supervision of Jian Chen in the Department of Software Development at Monash University. I am very grateful for Jian's support and dedication, however, in early 1997 Jian left the University and was unable to continue as supervisor. On Jian's departure the department arranged Xindong Wu as a replacement supervisor. However, Xindong also left the department approximately 12 months later which lead me to solicit Ann Nicholson and David Albrecht as supervisors. They agreed to take on the supervision of my thesis as it was related to their research interests of artificial intelligence and machine learning, although its topic was outside of their specific areas of expertise. I am extremely grateful for the assistance of both Ann and David throughout the completion of the thesis. I learnt a lot from them and there is no doubt that their feedback and suggestions greatly improved the quality of this thesis; any of its remaining faults are entirely my own responsibility.

A number of people, including several anonymous reviewers, provided valuable feedback on documents and papers related to this thesis. Their comments and encouragement helped sustain this research and was greatly appreciated. I would particularly like to thank the thesis examiners for their time and their thoughtful feedback. I would also like to thank the developers of the Swarm simulation system which was used for experiments in this thesis.

Many people within the School of Computer Science and Software Engineering have helped in various ways, including Heinz Schmidt for lengthy discussions and comments on key issues within the thesis, and Christine Mingins for her support throughout the thesis candidature.

Finally, I would like to give a special thanks to Allison, who has patiently put up with her partner's highs, lows and late hours.

Matthew Winston Mitchell

Monash University
August 2003

Chapter 1

Introduction

1.1 Situated Agents

Situated agents are agents which operate in real-world environments (Dorigo and Colombetti 1994). Such agents receive sensory information from the environment and take actions to affect that environment. Typically work on developing situated agents is conducted within the framework of animat or mobile robot type tasks where the agent moves in its environment attempting to avoid dangers and achieve goals (Wilson 1991; Dorigo 1995). Ideally such agents should be intelligent enough to be capable of acting as service agents, replacing humans to perform tasks which are dangerous or involve drudgery (Becker et al. 1999).

However, the complexities of real-world environments have largely restricted the use of situated agents to relatively controlled environments such as offices and museums (for example, Millān (1994)). These complexities include a large stochastic state space, noisy effectors and sensors, changing environments, and similar sensory situations in which some form of short-term memory is required to determine the correct action. The need for short-term memory is called *perceptual-aliasing* or *hidden-state* since some information necessary for the agent to make a decision is hidden from it. A situated agent must deal with all these complexities while continuing to be reactive. Being reactive means that actions are selected within a small constant time, so as to respond appropriately to real-time changes (Kaelbling 1993).

Traditionally, service agents for controlled environments (such as robots) have been provided with hand-coded solutions. However, hand-coding controllers for situated agents in uncontrolled environments has been less successful. Hand-coding requires an understanding of the problem which is so detailed that it makes manual specification extremely difficult (Thrun 1994). Furthermore, past attempts to code intelligent agents for non-situated domains have resulted in agents whose behaviour is brittle and difficult to change (Holland 1986; Nilsson 1995).

It appears the most promising method for developing situated agents is to allow such agents to learn solutions to problems for themselves. However, to do this within a practical time frame they need external assistance. Brooks (1991) points out the enormous number of experiments and incredible time it took for evolution to develop even the most basic abilities we would expect from situated service agents. One obvious source of assistance is the robot developer. Furthermore, the better the developer understands the problem the better the help he can provide. However, if the demands on developer time are too high developers might be tempted to resort to hand-coding solutions. As previously mentioned, such hand-coded agents are likely to be brittle, difficult to change and have limited usefulness. Given these issues, there is a need to balance the necessity to assist situated agents to learn while minimising the intervention required from their human developers.

This thesis addresses this balancing problem and a host of other problems related to creating situated learning agents. In doing so a system is presented which integrates a variety of existing and novel techniques. All the techniques are based on established approaches to the development of situated learning agents, and in turn each approach is based around a small number of requirements necessary for success.

1.2 Requirements for implementing Situated Learning Agents

There are three broad requirements for successfully implementing situated learning agents. The agent must support *efficient learning*, *efficient computation* and must be *multi-purpose*. Each of these requirements is summarised as follows:

1. **Efficient learning:** is minimising the amount of experience required to learn a task. This in turn minimises the risk of damage to the agent or its environment and reduces the amount of intervention required by human developers.

2. **Efficient computation:** is learning and reacting with realistic use of computational space and time. This is necessary for situated agents as they are interacting with the real world and must apply their experience within real-time limits (i.e be reactive).

3. **Multi-purpose:** is being capable of learning a variety of tasks within the environment and class of problems the learning system is intended for. This implies minimal task specific system customisation. For example, some neural network systems need to have the number of hidden-nodes customised for different tasks. The use of such system customisation is a means of providing assistance to the learning agent, but one which requires expensive designing and experimentation for each task as well as a specific and detailed knowledge of the learning system itself.

In reality these requirements are a simplification, there is much more that can be said about each (and is said in the following sections and chapters). However, the requirements as presented serve both as an ideal and as a guide. Establishing them from the outset provides a basis for balancing the many complex trade-offs which arise when implementing situated learning agents.

Having adopted these requirements it is now possible to identify the established approaches for achieving each.

1.3 Approaches to Achieving the Requirements

This section describes four approaches for achieving the three requirements. Each of these approaches is significant enough to be a research area in itself and each is commonly used in learning systems. However, for various reasons some systems may only adopt some of the following approaches. As there is not a one-to-one relationship between approaches and requirements, the following description also identifies the requirements addressed by each approach.

1. **Providing biases:** Learning without biases is impossible (Mitchell 1990). Therefore, the issue of biases is really about which biases should be provided and how. One source of bias is the internal representation and learning mechanisms implemented in the system. A system should have a representation appropriate to the problems and environment it is intended for (Mitchell 1990). Given the requirement of a *multi-purpose* learning system, these biases should remain largely unchanged across tasks. However, if we wish to achieve our requirement of *efficient learning* it is also necessary to provide biases specific to each individual task. There are a variety of means for doing this externally (i.e without customising the system for each task) and these are listed in the next section (Kaelbling, Littman, and Moore 1996).

2. **Efficient search and representation:** Efficient search for appropriate internal structures exploits biases provided both externally and built into the system architecture. An efficient representation is one which is compact in size and allows new knowledge to be added with minimal work. Neither search nor representation should adversely affect the reactivity of the agent, therefore implementing efficiency in search and representation supports the requirements of a *multi-purpose system* (since some tasks require reactivity) and *efficient computation.*

3. **Reusing learned structure:** This involves a single agent (i.e a single instantiation of a system) maximising the benefits of its experience by constructing an internal representation of its learned knowledge that can be used for more than one task (Harnad 1990). This ability is commonly called multi-task learning and is closely related to life-long learning (Thrun 1996; Caruana 1997). It requires that the agent can represent, and switch between, multiple tasks. A pre-requisite for reusing learned structure is that the system is *multi-purpose*. Re-using learned structure supports the requirement of *efficient learning.*

4. **Real-world learning:** There are in fact many approaches aimed at allowing situated learning agents to operate in real-world environments. One is to implement some form of short-term memory for problems with hidden-state. Others deal with large state spaces, noise and adaption to change. All of these are necessary for developing a *multi-purpose* learning system.

1.4 Techniques

There are a variety of established computational and representational techniques for implementing the approaches in the previous section. In addition to a selection of the established techniques commonly used by various other systems, the system proposed in this thesis also uses a combination of specifically developed novel techniques.

The following text describes the established techniques for each of the approaches from the previous section and discusses various important trade-offs that are encountered when selecting techniques. Following the established techniques is a brief summary of the techniques novel to this thesis.

1.4.1 Established techniques

Providing Biases

To reduce learning times it is necessary to provide biases (Kaelbling, Littman, and Moore 1996). One way of providing biases is through a *teacher.* Possible teaching methods include providing advice, leading the agent to a solution and breaking large problems down into smaller sub-problems which make learning easier (Lin and Mitchell 1993; Lin 1993; Maclin and Shavlik 1996).

However, it is desirable to minimise the intervention of the human developer/teacher. One way to do this is to implement the agent as a *reinforcement learning* agent. By providing reinforcement appropriately when a task is completed it is possible to avoid specifying the solution (Sutton and Barto 1998). Additional rewards (for sub-goals) and penalties (to avoid undesirable situations) can be provided to assist the agent in learning. Such manipulations of reward functions or provision of training rewards can make intractable tasks tractable and have been used successfully by Randløv and Alstrøm (1998) and Dorigo and Colombetti (1994).

Both teaching and reinforcement learning can be used to direct the learning system along particular paths thereby avoiding inefficient random-walks and exploration of unimportant parts of the state-space. This is especially important when dealing with hidden-state problems where long memories are required. Without some guidance, an agent may search many possible paths, without ever gaining the experience necessary with any single path to resolve perceptually aliased states, a problem which effects all learning programs in large spaces (Markovitch and Scott 1993; Littman, Cassandra, and Kaelbling 1995).

While all of these forms of providing bias are possible in the proposed system, of the above techniques only reinforcement is used to provide task specific biases in the experiments presented in following chapters.

There are also other possible sources of bias. For example, endowing our agent with actions and sensory inputs which make either the task or learning simpler. These techniques are identified when used in experiments with the proposed system.

Efficient Search and Representation

Many systems use an enumerative state representation in which each world state has a single unique observation or input (Chrisman 1992; McCallum 1993). Using this type of input representation is infeasible for situated agents, as there are too many states in the real world to create a unique structure for each. Instead, some form of *input generalisation* is required where the large set of world states is mapped to a smaller set of equivalent internal states (Chapman and Kaelbling 1991).

Another technique is *incremental learning.* The requirement that a program is incremental ensures that incorporating new knowledge into the system does not dramatically affect existing structures. It also requires that the selection of an action be reactive. A learning system that is "strictly incremental" should be able to always receive inputs and select an action within a fixed amount of time called a "tick" (Kaelbling 1993).[1]

One incremental method for input-generalisation is based on the creation of *default-hierarchies.* Default-hierarchies initially consist of general rules which are then specialised as required (Holland et al. 1986). Many researchers have argued for the benefits of hierarchies within learning systems including Newell (1990), Simon (1981), Tyrell (1993) and Dawkins (1976). These benefits include their ability to separate useful information from large amounts of redundant information and their usefulness in constructing improbable assemblies (Dawkins 1976). The most significant argument against hierarchies appears to be that they can be inflexible to changes and may

---

[1] There are a variety of other forms of incremental learning; see Langley (1996) for a discussion of these.

respond to slightly different situations with radically different behaviour (Maes 1991). However, not all hierarchies are subject to this criticism. Free-flow hierarchies use the combined evidence of a number of nodes to suggest preferences and to select actions (Tyrell 1993; Rosenblatt and Payton 1989). Similarly, systems built using default hierarchies may allow a number of parallel hierarchies to compete or co-operate in the process of action selection.

Creating hierarchies does not avoid the need for search in the space of possible structures. This search can be made more efficient (than brute force) using techniques such as *spatial and temporal selectivity* and/or *genetic algorithms* (Holland 1975; Foner and Maes 1994). Ultimately a decision must be made on which created structures to retain. There are two possible techniques for this. One is to decide whether the structure is useful based on its ability to predict some state in the environment (a *perceptual distinction*) and the other is to make this decision based on its usefulness (utility) for a specific task (a *utile distinction*) (McCallum 1995).

Retaining structure based on perceptual distinctions better addresses the requirement of efficient learning. The basic argument for this, presented by Drescher (1991), is that retaining structure based on utile distinctions is infeasibly slow. A learning agent may gain considerable experience exploring in its environment before reaching a goal. Using perceptual distinctions all this experience can be used to create internal structures representing the agent's environment. However, when making utile distinctions no decision can be made on retaining structure until the goal has been reached. Extending this idea further, Drescher (1991) argues that perceptual distinctions allow islands of rules to form (independently of any specific goal) which can later be connected together as the agent finds paths to goals. However, learning based on utile distinctions gradually extends a single continental rule set that surrounds a goal making little use of experiences beyond the fringe of that rule set.

The trade-off for using perceptual distinctions is a potentially larger rule set but a better use of experience versus a smaller rule set (i.e *efficient learning* versus *efficient computation*). This trade-off is complicated by the fact that it is not necessarily desirable to represent all structures which make perceptual improvements in the environment.[2]

A final issue is the common aim of many classical computer science techniques to arrive at the optimal solution for a problem. However, for situated learning agents optimal solutions are often inappropriate. This is primarily due to the size of the real-world state-space and the need for reactivity but is also due to the fact that the simplifying assumptions relied on by many classical solutions are impractical for real-world environments (Bellman 1957; Simon 1981). Indeed some consider that searching for optimal solutions in such spaces is an "unprosperous activity" (Wiering and Schmidhuber 1998). Instead it is necessary to trade-off optimal solutions in return for finding solutions quickly (Bowling and Veloso 1999). Appropriate techniques include *heuristic search* and finding *satisficing solutions* (i.e "good enough" solutions) (Simon 1981).

The proposed system learns incrementally using default hierarchies for its representation. A variety of heuristic search techniques are used which incorporate temporal and spatial selectivity. Both the ability to predict percepts and the ability to predict utility are used to assess structures created during search.

Reusing learned structure

Reusing learned structure for different tasks is critical to achieving our requirement of *efficient learning*. One possible means of achieving this is to use a separate internal structure to represent the goal of each different task. Paths to current goals can then be discovered using an on-line search of the agent's internal model to the goal state. The on-line search technique is called *hypothetical look-ahead* which uses a spreading activation through states in the internal model (Holland 1990). Unfortunately an on-line search of this type conflicts with our desire to be reactive since it may require multiple "ticks". This means it is not strictly incremental. However, the alternative is to sacrifice reuse. Consequently, on-line planning must be conducted carefully, efficiently and the constructed plans must allow for reactive adaption to the unexpected situations that must necessarily occur during plan execution. For this type of planning a system requires the equivalent of *SRS* (situation-action-situation) rules (Holland 1990; Drescher 1991).

The experience of many researchers using Back-propagation neural networks also suggests it is necessary to maintain *independent internal structures* to represent world states (McCloskey and Cohen 1989; Fahlman 1988b; French 1999). This prevents the learning of one task interfering with, and perhaps even entirely destroying, the learning for another task.

The proposed system uses independent structures organised as SRS rules which support both reuse and hypothetical look-ahead.

Real-world learning

Given the complexities of real-world environments, a suite of techniques are necessary if a learning system is to be successful. Some of the problems addressed by these techniques have proved more challenging than others. Dealing with hidden-state is one of these challenging problems. One technique for hidden-state is to maintain a *belief state* which summarises the history of the agent, another is to maintain a more explicit form of *short-term memory*. Short-term memory approaches appear to be more feasible in the real world than belief state approaches and are commonly combined with

---

[2] However, a suitable solution to this may be requiring the teacher to direct the learner away from experiences which provide knowledge irrelevant to any task and to minimise the exploration conducted by the agent after all required tasks are taught.

representations for input generalisation (Littman, Cassandra, and Kaelbling 1995; McCallum 1995; Ring 1994).

The most significant remaining problems are large state spaces, non-determinism and adaption to change. A number of techniques for large state spaces have already been discussed along with techniques for efficient search and representation. Non-determinism in the agent's sensors, effectors and environment transitions is often dealt with by *representing the effects of actions probabilistically* along with alternatives. Adaption to change is commonly supported by various combinations of incremental learning, using *recency weighting* when evaluating internal structures and the integration of *exploration* with restricted *on-line planning* (Sutton 1991b).

Of these techniques, the proposed system uses short-term memory, recency weighting, exploration and representing the effects of actions probabilistically.

All of the above techniques (and many others not mentioned) have been used by various researchers to tackle the problem of situated learning. Often, the work of these researchers has highlighted problems or difficulties in the techniques. To tackle some of these problems a number of novel techniques have been developed specifically for the system presented in this thesis. These are introduced below with the details presented in subsequent chapters.

1.4.2 Novel Techniques

The following techniques are all specific to the system proposed in this thesis. To help explain the techniques it is necessary to point out that this system (like many others) receives sensory inputs in the form of bit strings of fixed length at discrete time steps. The current state of the environment as detected by sensors is presented in a bit string containing a number of zeroes and ones to indicate the presence of various features in the environment.

Combining Perceptual and Utile Distinctions:

Perceptual distinctions and the use of utilities to evaluate structure are not necessarily mutually exclusive. While relying primarily on perceptual distinctions, the system presented in this thesis also makes use of utility estimates in a novel way to retain and remove structures.

Making Binary Combinations of Structures:

When creating and evaluating new structures, some learning mechanisms consider all information about the environment provided by sensors (eg. Drescher's schema mechanism). Other learning mechanisms consider a selected sub-set of the information for each new structure (eg. Learning Classifier Systems using Genetic algorithms) or even individual bits of information (eg. the G-algorithm) (Drescher 1991; Holland 1975; Chapman and Kaelbling 1991).

Considering more sensory information when evaluating structures makes it easier to detect significant relationships in the data, but at the cost of higher space and computational overheads. The proposed system uses binary combinations of the individual bits received from sensors (and/or other extant structures). To compensate for the increased difficulty in detecting the usefulness of structures using only some of the sensory data available at any one time, a novel application of non-parametric statistics is used. This use of statistics, along with the system's method of using binary combinations in its search for new structures, is unique.

Use of Hierarchies:

As mentioned in Section 1.4.1, one benefit of hierarchies is their ability to separate useful information from large amounts of redundant information. Another is their usefulness in constructing improbable assemblies, such as solutions to complex problems, by reusing and combining the solutions for less complex problems (Dawkins 1976).

Using default hierarchies is an established technique which provides the first benefit. However, the proposed system also takes advantage of hierarchies to realise the second benefit. The system constructs complex structures, which are essentially rules, using less complex structures, which are also rules, as components. However, unlike many other rule-based systems, each of the component rules has demonstrated usefulness in its own right, ensuring that all new structures consist only of already useful structures.

Grouping Related Statistics for Reuse:

The proposed system groups SRS rules in a way that allows for increased reuse of created structures. As well as providing efficiencies in representation, this grouping allows more efficient search. Message passing mechanisms are used which take advantage of the system's groups to avoid explicit logical NOT structures. This reduces the search space substantially by eliminating a large number of equivalent structures. The trade-off is that additional input bit positions may be required. The overall effect is a system which is still computationally complete (i.e can represent logical AND and NOT) but whose simplest representation of a concept may be slightly larger than a system which can explicitly represent NOT. However, the proposed system is biased towards creating logically simple structures in preference to complex ones (this is explained in Section 4.2.6).

Selective Mechanisms:

A variety of selective mechanisms are used in the proposed system to reduce the amount of sensory information considered when searching for new internal structures. These include a range of commonly used selective mechanisms such as spatial and temporal selectivity (Foner and Maes 1994). However, what is unusual is the system's method of ignoring large portions of the information received from sensors. The system's ability to do this successfully is due to its unique representation and search techniques and the design of sensory inputs concomitant with those techniques. Effectively the system's structures process only some inputs, treating the rest as irrelevant, which distinguishes it from systems which include all sensory information in all rule conditions.

Representing hidden-state:

Like many other approaches, the proposed system constructs Markov-k short-term memories. Also like many other systems, k can be of arbitrary depth and vary as appropriate in different areas of the search space. However, unlike many other systems, the proposed system builds its structures based primarily on perceptual distinctions. These structures are integrated with the other internal structures for dealing with real-world state spaces. This integration is not unusual, however, the advantage of the proposed system is that unlike some existing systems it does not require a fixed sized history window or fringe to uncover hidden-state (e.g McCallum (1995)'s U-Tree).

The proposed system also uses two novel techniques when representing hidden-state. One is the internal structures used to represent short-term memory, the other is the search technique for discovering suitable memory structures. Combined these two techniques provide various advantages over existing alternative methods.

1.5 System Evaluation

This thesis is an empirical investigation into the unique combination of existing and new techniques that comprise the proposed system. While many analytical arguments are presented, the behaviour of the system is complex to analyse formally. So the claims and results of this thesis are supported by experimental results rather than formal proofs.

Ideally the experimental results of any system should demonstrate the system's capabilities and allow comparison with other systems. To this end, experiments are often designed to reveal the particular capability or advantage of a system. In fact, the experiments presented in this thesis serve three purposes: (i) demonstrating the proposed system's capabilities; (ii) allowing comparison with related systems; and (iii) allowing an analysis of the proposed system's behaviour and performance. In doing this, the thesis uses well-known problems from the literature including those which contain hidden-state and require input-generalisation.

Input generalisation is demonstrated using the Monk problem which consists of three classification tasks, one of which includes noise. The Monk tasks are also used to analyse the system's performance on classification tasks and assess how it is affected by different parameter settings. Rule chaining, and the ability of the system to learn multiple tasks, are demonstrated using 4x4 grid navigation tasks in which two different grid locations are designated as goals at different times. One of these grid experiments includes noisy effectors.

Following this, are a variety of problems which test the ability of the system to use short-term memory to represent hidden-state. At first, a series of gap tasks are used in which an initial state must be remembered for fixed periods of time. The results on these tasks indicate the ability of the system to scale as the amount of memory required increases. Following the gap tasks is a commonly used maze navigation task called the M-maze from McCallum (1993). One version of the M-maze tests the system's ability to cope with both noisy sensors and effectors. The M-maze is followed by another gap task which demonstrates the ability of the system to construct memory efficiently in an environment which contains irrelevant attributes. The system is then tested on two other mazes from Ring (1994). The first of these is 5x4 grid and the second, a larger 9x9 grid. The final task is a truck driving task from McCallum (1995) which demonstrates the combined use of short-term memory and input generalisation. The various tasks used in experiments and the abilities they demonstrate are summarised in Table 1.1.

Experiment	Demonstrated Abilities
Monk problem	Input generalisation
4x4 grid navigation	Rule chaining, Multiple tasks, Noisy effectors
Gap tasks	Hidden-state, Dealing with irrelevant attributes
M-maze	Hidden-state, Noisy sensors and effectors
5x4 grid	Hidden-state
9x9 grid	Hidden-state
Truck driving	Input generalisation, Hidden-state

Table 1.1: Tasks used in experiments and the system capabilities they demonstrate.

1.6 Related Research

The proposed system has various relationships to a number of other systems. However, there are two systems to which it is particularly close. One is the Learning Classifier System (LCS) originating with Holland (1975), the other is the Schema mechanism developed by Drescher (Drescher 1991).

One of the major short-comings of the LCS is its reliance on a genetic algorithm (GA) for rule-discovery. Using a GA, new rules are generated by selecting components from existing moderately useful rules which are recombining to generate increasingly useful rules. However, many of the rules generated perform poorly due to both an inability to identify and select the important components of existing rules and the inappropriate combination of otherwise useful components (Shu and Schaeffer 1991). In fact, it is possible that this problem is due not so much to the use of a GA as to the lack of information on the usefulness of individual components of rules. But regardless of the primary source of this difficulty, the truth remains that in practice it is often necessary to supplement the GA with various other rule-discovery mechanisms (Riolo 1989).

Drescher (1991)'s Schema mechanism discovers new rules using a mechanism called *marginal attribution* to uncover statistically significant relationships. The major drawback of this approach is that it includes all sensory inputs in statistical tests.

Another short-coming of both the LCS and Drescher's Schema mechanism relates to their ability to represent hidden-state. While both systems can in theory represent hidden-state, in practice the LCS and its many variants have so far proven inadequate. Drescher's Schema mechanism also lacks experimental evidence on common problems which demonstrate its ability to represent hidden-state.

Other related systems are described in Chapter 2, and further comparisons with selected systems are presented in Chapter 8.

1.7 Contributions

The system presented in this thesis goes further towards meeting the requirements of a situated learning agent than any other system to date. It does this through a unique combination of representational and computational techniques. The system includes a range of entirely new techniques as well as a number of novel customisations to existing techniques based on the lessons provided by the past research of others.

The thesis provides three specific contributions towards the goal of realising a situated learning agent for uncontrolled environments. These are in the areas of input generalisation, dealing with hidden-state and supporting hypothetical look-ahead search.

1. **Input generalisation:** The proposed system incorporates an efficient input generalisation mechanism which reduces search in three ways by:

    - eliminating equivalent structures;
    - ignoring irrelevant inputs; and
    - reusing created structures.

    The variety of existing and novel techniques which do this are described in detail in Chapter 4. Section 4.2.6 specifically describes eliminating equivalent structures and ignoring irrelevant inputs while Sections 5.3, 5.6 and 7.8 illustrate the reuse of structure. The system's ability to perform input generalisation and state discrimination is demonstrated by the classification and grid navigation experiments presented in Chapter 5.

2. **Hidden-state:** A new technique is used for uncovering hidden-state. It uses independent structures to reduce the search required which integrate with the input generalisation structures. The technique combines a novel search method with additional novel mechanisms to address commonly overlooked difficulties in constructing Markov-k memory. The hidden-state technique is described in Chapter 6 and the ability of the system to solve a number of difficult hidden-state tasks is verified experimentally in Chapter 7. The successful integration of the hidden-state techniques with the input generalisation techniques is demonstrated in the truck driving task in Section 7.8.

3. **Hypothetical look-ahead:** Like other systems with independent structures (i.e Learning Classifier systems and Drescher's Schema mechanism), the proposed system learns a task independent internal model which can be searched for paths to goals. However, the use of such search in conjunction with the system's efficient representation means that a model can be used to achieve a large reduction in the training and space required to perform multiple tasks in common domains. These results are presented in Section 5.7.

Since the proposed system combines input generalisation (classification) capabilities with the use of temporal structures to implement short-term memory it is refered to in subsequent chapters as TRACA (Temporal Reinforcement-learning And Classification Architecture).

1.8 Thesis Outline

Chapter 2 provides a more detailed exposition of many issues related to learning agents supplementing those presented above. Chapter 3 presents an overview of the architecture of the proposed system. This overview divides the description of the system into two parts. The first part is the basic system, which is used for input generalisation and does not include short-term memory for problems with hidden-state. Short-term memory for hidden-state is presented as the second part which completes the implemented aspects of the proposed system.

The description of the basic system and experiments using it are presented in Chapters 4 and 5 respectively. The short-term memory techniques for representing hidden-state are presented in Chapter 6. Experiments using these techniques to solve problems with hidden-state are provided in Chapter 7, which also contains results on a task which combines hidden-state and input generalisation. Finally, Chapter 8 provides an overall evaluation of the system, additional comparisons with closely related systems and a discussion of potential future research using the proposed system.

Chapter 2

Learning

2.1 The Role of Memory

The primary objective of this chapter is to introduce some problems requiring memory for learning and to discuss appropriate approaches. Memory in this sense means the retention of information for a limited period of time rather than recording and storing information which summarises a range of experiences in the environment.

Techniques for tasks that require memory must often rely upon, and interact with, techniques that are not memory based and for which a large amount of learning theory was developed. This chapter describes the theory and attempts to relate together the wide variety of techniques for different problems.

The next section introduces at a high-level the type of problem that requires memory. This is followed by the description of theories and issues related to learning in general, and then by a discussion of different techniques for learning.

2.2 Using Memory

Often in everyday life we require memory of recent events to help us complete tasks. This is because where we are right now (or perhaps where we are looking right now) we might not have access to sufficient information to complete our task. For example, if I am at the supermarket purchasing the items I need to make breakfast tomorrow, I will not know whether or not to purchase milk, unless I can remember if there is milk in the refrigerator at home. The information about the contents of my refrigerator is not available to me at the supermarket and I must rely on my memory. This is a hidden-state problem. There is some important information I need to help me do something (in this case, complete my shopping without spending money unnecessarily) which is not immediately available (it is hidden) (Colombetti and Dorigo 1994; Whitehead and Lin 1995). The hidden-state problem is also

known as perceptual-aliasing since it results from two or more states having the same percept (i.e observation) (Whitehead and Ballard 1991).

The solution to the shopping problem does not seem very difficult. When I first buy milk, I simply assume I have milk at home until I run out, after which I make a point of remembering that I must buy more. I keep remembering I must buy more until such time as I make the purchase. But there are other situations where simply remembering a fact for an indefinite amount of time, such as in this case, is not sufficient.

Take another example problem. Areas in suburbs around cities can frequently look very similar. A number of suburbs may have leafy streets, many others may be dominated by high-rise buildings, even beach-side suburbs may look similar if they are on a uniform coastline. Despite these similarities when travelling across town you generally know which suburb you are currently in, even if it looks similar to many others. This is often possible because you can remember where you came from and use that information to deduce where you currently are and how to get back. However, this deduction requires you to also recall what you have done since leaving your departure point.

It is in situations like this, where additional memory is required about what you have done since you started, that makes the hidden-state problem more complex (this problem is elucidated by Whitehead and Ballard (1991)). To deduce where we are now we must know in what directions, and for what distances, we have travelled since we left. Even if we know this, to know what suburb we are in we must know what suburbs exist at different directions and distances from our starting point.

Consider one final example. Imagine you are a tourist in a famous foreign city and you have caught the wrong bus. The bus is very crowded, so it is difficult to see outside. When you eventually get off you have no idea where you are. You try asking for help, however, you do not understand the local language well enough to interpret people's responses. The street you are standing on could be one of thousands in one of many urban areas and it is difficult to predict where you will end up by going in any particular direction. Rather than risk another bus trip, you now consider two choices. The first is to search around the immediate area for some well-known landmark which could help identify your location, and the second is to make a guess of your location and based on that guess head in the direction you think best. If we do the first, once the landmark is found (however long that may take), we have a new starting point and can use our deduction method to find our way home. If we adopt the second option, we maintain a belief about where we currently are, and update that belief based on the things we see as we travel. For example, a propensity of restaurants may encourage us to believe we are near the central railway station, while a large number of residential apartment buildings may increase our belief that we are away from the the city center.

Of course, even with the second approach we may still look for landmarks which would unambiguously discern our position and this search may make us notice various shops and streets that we pass. Since many of these are not marked on our free, but incomplete, map from the hotel, most are just distractions, irrelevant to the current task of finding our way back. However, noticing restaurants may be useful for later when we are considering having dinner somewhere.

In the end, based on where we think we are, we may just adopt the *policy* that we will head west until we see something familiar. Let us suppose that we eventually find we are near our hotel because we identify the street-trader opposite a cafe which we use as landmark. At this point we head north until we reach the hotel.

So far in our discussion we have seen a number of different issues related to using memory for hidden-state problems. We have seen that in some cases simply remembering something for an indefinite amount of time is sufficient (i.e: whether or not there is milk at home). However, when there are multiple sequences of states which appear the same, but each sequence requires different behaviour, additional information is required (such as when travelling across suburbs). This additional information could come in two forms. One is remembering what we have done since our last unambiguous state. The other is to include additional location information to disambiguate similar states (such as street signs). This additional information may be an unusual combination of common features. In all these cases, we require some information about what the possible states are and their relationship (transitions) to each other. Finally, whatever task we are currently involved in, we cannot help but notice things of interest in our environment that, while not immediately useful, may well be useful later.

The following sections introduce some of the research in machine learning that is relevant to solving problems like those described above. The issues a machine learning agent faces are very similar to those you encounter every day. A learning agent must learn to identify useful combinations of features (such as the street-trader opposite the cafe) and about the area it will be moving in (equivalent to knowing about suburbs and their topology) and also when to keep track of its movements so as to be able to determine where it is (equivalent to having memory of recent situations and actions). In doing this the agent may make interesting observations unrelated to the current task but useful for another task later on.

A lot of machine learning theory is developed around the simpler case where at any time we have complete information about our situation (or *state*). This is unrealistic as it implies we do not require memory. For instance, in the case of our shopping example, we know whether there is milk at home without needing to explicitly remember this, the information is always immediately available. If we make this assumption that we always have complete knowledge about our current state (and it is true), then the problem we are dealing with is said to be a *Markov Decision Problem* (MDP). We will look at learning for Markov Decision Problems before looking again at the complications which arise when the assumption they are based on is dropped.

2.3 Markov Decision Problems

Formally described, Markov Decision Problems have a finite set of states $\mathcal{S}$, possible actions $\mathcal{A}$ and transition function, $\mathcal{T}$. Given that the environment is in a particular state $s_t \in \mathcal{S}$ at time t, the agent can select an action $a_t \in \mathcal{A}$. The next state is then determined using the transition function as $s_{(t+1)} = \mathcal{T}(s_t, a_t)$. This transition function can be stochastic where the probability of moving to the state at s_{t+1} given that we take action a_t in state s_t is given by $Pr\{\mathcal{T}(s_t, a_t) = s_{t+1}\}$ (Howard 1971). The fact that the transition probability from s_t to s_{t+1} given the action a_t depends only on s_t means that this environment has the *Markov property*.

States in a MDP are typically represented using a tabular or *enumerative* scheme where each state has a unique identifier which can be observed by a learning agent. This type of enumerative representation of MDPs makes learning states and transitions between states trivial. The agent can construct a table sufficiently large for all possible states and for each state encountered keep a record of the transitions to other states and their probabilities. The big disadvantage of an enumerative approach is that if the state space is large, the table will also be large. Furthermore, the larger the state space, the longer it will take to learn an accurate transition function. Alleviating this problem often requires reducing the number of possible observations received by the learning agent. However, this removes the one-to-one mapping of states to observations which in turn makes it difficult for a learning agent to correctly represent the states and transitions of the *true Markov model* underlying the observations it receives. The assumption that there is a one-to-one mapping from observations to states in an underlying MDP is called the *Markov assumption*.

While Markov Decision Problems may allow a complete model of the states and transitions of the environment, they do not by themselves allow us to determine what is the best action to take in any given state. If we know which action is the best to take given any particular state, then we know the optimal policy. Finding the optimal policy can be achieved using reinforcement learning to discover the value of states before using those values to derive the actions for the policy. Because reinforcement learning relies on discovering the value of states first it is an indirect method of learning the optimal policy.

2.4 Reinforcement Learning

Reinforcement learning uses the rewards received while acting to associate an estimated reinforcement value with each possible state description. Rewards may be received either immediately on reaching particular states, or received later as a result of passing one or more states, in which case the reward is said to be delayed. To successfully learn a value function in environments where rewards are delayed we must assume that the agent has available (has been given or has learned) the true Markov transition model. We also need to know the expected reinforcement for each state. Given these conditions, a value function, $V(s)$ may be calculated for a particular policy π, which gives the probability $\pi(s, a)$ of selecting action

$a \in \mathcal{A}$ when in state $s \in \mathcal{S}$. After selecting an action the agent is in a new state s_{t+1} and receives a reward $r_{t+1} \in \Re$. $R(s_t, a_t)$ gives the expected value of r_{t+1}. In this case, the value of state s under policy π is denoted as $V^\pi(s)$ and defined as follows (Sutton and Barto 1998):

$$V^\pi(s) = E_\pi \{R_t \mid s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\}, \tag{2.1}$$

where E_π denotes the expected value given that the policy π is followed. γ is a discount factor which can be seen as the cost of taking actions or an expectation of living another step.

Ideally we would like our agent to learn the optimal policy, π^*, which is the policy that maximises $V^\pi(s)$ for all states. The value function of an optimal policy is denoted as V^* (Mitchell 1997).

One method of finding an optimal policy, is to use a dynamic programming technique called *value iteration*. Value iteration combines learning state values and finding the optimal policy. Estimates of values are stored for each state and are used to incrementally update the estimates of neighbouring states. Each individual update is called a *backup* (Sutton and Barto 1998). This process is repeatedly applied with the maximum value of a state's neighbours used to update its value in each iteration (Bellman 1957).

The same principles for state-value functions apply to action-value functions. The action value function for policy π is the expected return for taking action a starting in state s and following policy π thereafter. This function, denoted as $Q^\pi(s, a)$, is defined as (Sutton and Barto 1998):

$$Q^\pi(s, a) = E_\pi \{R_t \mid s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\}. \tag{2.2}$$

Action-value functions have the advantage that they can be used in the absence of a model for state transitions (Watkins and Dayan 1992). To learn action-values, methods such as Q-learning (attributable to Watkins (1989)) are used. Q-learning takes advantage of the relationship,

$$V^*(s) = \max_a Q(s, a) \tag{2.3}$$

to learn the optimal policy π^* (Watkins and Dayan 1992; Mitchell 1997). The update rule to learn the optimal policy in deterministic environments, based on the repeated updates of an estimate, $Q(x, a)$, is (Sutton and Barto 1998):

$$Q(x, a) \leftarrow Q(x, a) + \beta \cdot (r + \gamma \cdot \max_{k \in \mathcal{A}} Q(y, k) - Q(x, a)) \tag{2.4}$$

where β is a positive learning parameter. For guaranteed convergence Q-learning relies on visiting every state-action pair infinitely often and that values are stored using a table based state representation or equivalent (Watkins and Dayan 1992). Because Q-learning will converge to the optimal value function, regardless of the policy followed, it is called an off-policy learning method. Variations on the learning rule are possible (Lin and Mitchell 1993) and a generalised version can be applied to stochastic environments (Mitchell 1997).

State-value update rules are often instances of one-step temporal difference (TD) methods. However, there exist multiple step TD methods that take into account the reward from not just the next state but a series of subsequent states. These methods are know as $TD(\lambda)$, where λ indicates the degree to which subsequent steps contribute to the update. $TD(0)$ applies only the immediate returns/rewards, while TD(1) applies returns and rewards up to and including a goal state and is equivalent to supervised learning (Sutton 1988).

2.4.1 Exploration

In practice, optimal policies for reinforcement learning techniques such as Q-learning can be found without the infinite number of trials required for convergence. However, knowing when to stop learning is a difficult problem. The agent could execute the policy derived from the values obtained in its limited experience, and hope that experience is sufficient to find a useful policy, or it could continue learning in the hope of finding a better policy. To avoid exploring forever, the agent needs to make a decision as to whether it is better to exploit the knowledge it already has, and greedily go after rewards based on that knowledge, or whether it should continue learning and find either a better policy or perhaps new reward states it does not yet know about.

One simple method of addressing the problem of balancing exploration with exploitation is to spend some percentage of time exploiting current knowledge. As learning progresses, this percentage can be gradually increased so that exploratory actions are taken less and less often as the agent gains more experience (Sutton 1991a). Other approaches to exploration are discussed in Section 2.14.

An alternative to increasing exploration is to speed up the propagation of values from reward states to other states in an attempt to gain a better policy with less experience. Dyna-Q uses experience in a model of the environment in place of experience in the real-world to accelerate learning the value function (Sutton 1991b). Variations on this theme, which are more selective about model states to gain hypothetical experience with, include Queue-Dyna (Peng and Williams 1993) and Prioritised Sweeping (Moore and Atkeson 1993).

This section has focused on temporal difference methods which deal with the problem of *temporal credit assignment* which occurs when rewards are delayed. When a reward is received these methods attribute returns to actions which contributed to receiving that reward. However, in many tasks simply assigning credit temporally is not sufficient. In large state spaces it is also necessary to identify what the aspects of the state were which made the action selected appropriate. This is the *structural credit assignment* problem. Solving this requires more effort by the learning agent in to identify the useful aspects states, a problem which is discussed in the next section.

2.5 Identifying States

As mentioned in Section 2.3, for domains with a relatively small number of states enumerative schemes can be used. One possible implementation of an enumerative scheme is to present a fixed length input string to the agent at each timestep with bit values indicating the current state. If the number of bits is equal to the number of states, then the enumeration can be achieved by assigning each bit position to a unique state. The current state can now be indicated by setting its bit to 1 while setting all other bits to 0. This approach to state identification makes updating the value function straightforward. However, in the real world this type of state labelling is infeasible for two reasons. Firstly, there are too many states in the real world to enumerate all of them. Secondly, an agent in the real world cannot always have access to sufficient information at any one time necessary to uniquely identify every state of the world.

These two issues change both the nature of the input and its treatment once received by the agent. Even if an enumerative approach was possible for large state-spaces, an agent will take too long to get enough experience with all possible states to learn a value function across those states (Chapman and Kaelbling 1991). To cope with such a large state space the agent must generalise; a number of states must be treated as the same or similar (Holland, Holyoak, Nisbett, and Thagard 1986; Chapman and Kaelbling 1991). Input generalisation is possible if features of states can be represented in the input string using the assignment of values to a set of state variables (Koller and Parr 2000). This representation of different features in the environment requires a *distributed sensor* scheme, where a number of bits in the input string contain a 1 (one or more bits are set to 1 for each possible feature), as opposed to a *localised sensor* scheme, in which a single bit is used (a single bit it set to 1 for all features) to summarise the features of each state (Ring 1994).

For input generalisation to be useful, the mapping of inputs to generalisations (*internal states*) must retain the important features of the state. For reinforcement learning, the inputs mapped must share the same action and value. Inputs received from the real world are then classed according to features they contain that are useful for identifying similar states. Features that are not useful for generalisation (irrelevant features) can be ignored.[1] The role of the reinforcement learning agent now is not just to learn a value function, but to also learn important features which can be used to group states together correctly.[2] If this grouping is adequate, the agent does not need experience in all possible situations, it can map previously unseen inputs to the correct generalisation and use the value function for that generalisation when updating the policy to select an action. A diverse range of approaches have been applied to generalisation some of which are discussed in Section 2.13.

Ignoring some information in states (or bit positions in input vectors) means that the agent does not receive (or use) all the information necessary to uniquely identify a state. However, this may not prevent the agent from finding a good policy, as long as the information received (used) is sufficient for the agent to map its current situation to a correct (i.e useful) internal state and utility value.

This is the big jump from classical reinforcement learning to approaches used for large state spaces. Now we are conducting reinforcement learning for sets of features, perhaps based on boolean functions of the input vector, rather than for individual states. Sets which indicate useful *features* or *attributes* of the environment are represented as internal states. A generalisation is successful if the input vectors mapped to an internal state share the same utility value and transition function to other states, which must also be represented as internal states.[3] All learning algorithms targeting large state spaces (for example, those described by McCallum (1995) and Chapman and Kaelbling (1991)) face this same problem and must somehow map a large number of potential inputs to a smaller number of generalised internal structures.[4] However, using a relatively small set of generalised states denies some of the conditions necessary for guaranteed convergence of learned value-functions.

2.6 Learning Value-functions for Generalised Representations

The creation of generalisations may affect learning of the value function in a variety of ways. During learning generalisations may be too general, meaning that the values for the states it represents are being updated inappropriately. In some cases, it is possible that a completely correct generalisation is never found. In other cases, function approximators, such as neural networks, may use a shared distributed representation where interference between nodes may cause disruptions during learning (Fahlman 1988a). Since these are not table based representations of the state space, they do not meet the requirements for convergence of learned value functions (Papavassiliou and Russell 1999). This has been found to cause difficulties in achieving successful learning of useful value functions. Approaches used to address this problem include multi-step updates where updates to value functions are not based simply on the intermediate results of a previous iteration of a one-step backup algorithm, but on the (either real or simulated) following of a complete path to a goal in a manner similar to $TD(1)$ methods (Boyan and Moore 1995; Sutton 1996). However, in one set of experiments Sutton (1996) found that using the *sarsa* algorithm, an algorithm similar

---

[1] Note that just using generalisation is often not sufficient, even humans can not process large inputs without them being passed through complex visual processing such as discussed by Chapman (1991).

[2] Traditionally, this is a task often done prior to applying supervised learning techniques. Laird and Saul (1994) use a learning algorithm to identify features which can then be used by a supervised learning algorithm to learn classifications. TRACA does both these tasks simultaneously.

[3] Here appears Drescher (1991)'s and Moore, Baird, and Kaelbling (1999)'s chicken-or-egg problem, until we have the generalised internal states how can we assess them?

[4] TRACA does this by starting with a number of generalisations ($O(n)$ where n is the length of the input vector) then incrementally specialises these to create internal structures (default hierarchies) which represent useful sets.

to $TD(\lambda)$ but using state-action pairs rather than states, and a different function approximator (CMAC), the best results were obtained with $1 > \lambda > 0$. The issue of reliably learning value functions when using function approximators, has lead to a variety of learning methods being proposed (see for example; Papavassiliou and Russell (1999), Mahadevan (1996) and Baird and Moore (1999)) and analyses (for example, Singh, Jaakkola, Littman, and Szepesvári (2000) and Tsitsiklis and Van Roy (2002)).

2.7 Hidden-state Problems

One problem that occurs when the current input does not uniquely identify the current state, is the hidden-state problem which was introduced earlier in this chapter. This problem may also occur when generalising if a number of different states are mapped to a single internal state (Whitehead and Ballard 1991). When faced with a hidden-state problem the learning agent may have to either rely on memory of recent inputs or specifically take actions to find the information necessary to make good decisions. Agents in environments with hidden-state may violate the Markov assumption by treating multiple different states as the same, resulting in the agent learning incorrect transition estimates and utility values and making poor decisions. Problems containing hidden-state are also called Partially Observable Environments (POEs), since the true state cannot be observed directly.

There are a number of possible strategies for handling hidden-state. One strategy is to try and find regions of the state space in which hidden-state does not occur and operate only within these regions (such as done by Whitehead and Ballard (1991)). Another strategy is to adopt stochastic policies which allow a learning agent to "break out" of hidden-state regions (some of these are described in Section 2.12.1). A third strategy is to attempt to uncover the hidden-state by remembering previous states or observations. This type of problem is called Markov-k since k state identifications (and possibly state/action pairs) are required in sequence to correctly identify a state. When using a tabular or enumerative state-space representation, the space complexity of a problem requiring such memory is N^k (Howard 1971). However, for feature based representations suited to input generalisation this complexity may be considerably less.

The Markov-k strategy is one of the more common memory based strategies for solving hidden-state problems (some others are discussed in Section 2.11). Another common strategy is to use belief states which summarise the history of the agent. Both strategies aim to overcome the problems of incorrectly identifying states due to a failing Markov assumption.

Belief methods for overcoming the problem of hidden-state are based on Hidden Markov Models (HMMs). Like the Markov-k strategy, HMMs use the sequence of state observations experienced so far to track the true state in a partially observable process (i.e a hidden-state problem) (Rabiner 1989). However, rather than explicitly remembering past observations, belief states maintain a summary of past experience. HMMs approaches can be extended with the ability to select an action in each state. When actions effect future states, the selection of an action is a decision. Extending HMMs with actions leads to Partially Observable Markov Decision Problems (POMDPs), which are the classical approach for learning solutions to POEs (Lovejoy 1991). POMDPs also deal with hidden-state using belief states, where the belief we are in a particular state represents the probability of being in that state. POMDP approaches depend on having available observation and transition probabilities for a set of states. While it is possible to update observation and transition probabilities using the Baum-Welch procedure or an equivalent (this is discussed further in Section 2.9), often the set of states is not known before hand. One approach to finding an appropriate set of states is to collect statistics on transitions to observations for varying sequences of prior observations (see for example Chrisman (1992)). Once a state space is learned (or while it is being learned), the agent must also learn a policy for the space which leads to the selection of useful actions. As mentioned in Section 2.4 this can be achieved by learning a value function using a technique such as value-iteration. However, belief states are continuous and most methods for learning value functions apply only to discrete state spaces. Fortunately, it has been shown that optimal value functions can be approximated very well for POMDPs (and in some cases found exactly) (Smallwood and Sondik 1973; Sondik 1973). The problem is how to efficiently achieve that approximation.

2.8 Summary of Two Approaches to Hidden-state

In summary of the previous section, when dealing with hidden-state two common approaches are:

1. Use Markov-k memory to uncover the hidden-state in important regions of the state space; and
2. Maintain a belief about which state you are currently in and update the belief state according to observations you receive.

The first approach is the one used by TRACA (and also by McCallum (1995)'s U-Tree and Ring (1994)'s Temporal Transition Hierarchies). This approach involves representing the problem as Markov-k for arbitrary k in the areas of the state space which contain hidden-state. Its major disadvantage is that it can only reliably determine the current state for areas of the state space for which it has appropriate memory structures. Since building such structures for the entire state space is generally infeasible there may be paths which when taken cannot be tracked as a Markov process.

The second approach, using belief states, also converts the process to a Markov process. This is done by using the current belief state to summarise the history of the agent since it started. Any uncertainty about the current state is represented by maintaining a set of beliefs about which state the agent may currently be in. This is done using a vector with a value between 0 and 1.0 for every state. The vector values must sum to 1.0 at any given time and values closer to 1.0 indicate a higher probability that the corresponding represented state is in fact the current real state. The range of values the vector may take is the *belief space.*

The next section looks at the use of belief states in more detail before returning to the discussion of Markov-k and other hidden-state approaches.

2.9 Using Belief States

Maintaining a useful belief vector requires having a model of the problem. The four things necessary for the model are (Rabiner 1989):

- a set of states;
- the transition probabilities between these states;
- the observation probabilities for each of the states; and
- the initial state probability distribution (i.e the probability of starting in each state).

If we have such a model we can estimate the current state for a given sequence of observations. One method of using our model to find the most likely sequence of states that corresponds to our observation sequence (and therefore our most likely current state) along with its probability of occuring is to use the *Viterbi Algorithm* (Rabiner 1989). Another is to use Bayesian conditioning (as done in Chrisman (1992)). Once the belief state is known it summarises all the information that can be known about the agent's state, in other words the belief space is Markov (Littman 1994b). This is important, as it means that using belief states a POMDP can be treated as Markov Decision problem, which is a necessary condition for the use of value functions to determine the best action in each state.

As mentioned in Section 2.7, it is possible to update our model's transition and observation probabilities using learning procedures such as the Baum-Welch algorithm or any equivalent Expectation-Maximisation (EM) algorithm (Rabiner 1989; Dempster, Laird, and Rubin 1977).[5] However, these procedures do not provide us with a method to determine how new states should be added to our model so the usefulness of any current state estimations we calculate will be restricted by this.

If new states can be somehow introduced into a model the update procedures can be used to modify the observation and transition probabilities for the new model. Introducing new states is the problem addressed by Chrisman (1992)'s Perceptual distinctions method, particularly in relation to perceptual-aliasing, which uses statistical measures to detect significant differences between experiences in an environment based on the recorded

---

[5] Some problems with using the Baum-Welch algorithm are discussed and addressed in Shatkay and Kaelbling (1997)

transitions between states and observation frequencies. The test used by Chrisman for introducing new states is not always reliable and too many states may be created or relevant states not introduced due to insufficient statistics.

McCallum (1993)'s Utile Distinction test also introduces new states, but uses statistical differences in the prediction of utilities (returns and rewards) to introduce new model states rather than differences between expected transitions and observations (perceptual distinctions). McCallum (1993)'s method also has the ability to remove introduced states if they are later found to be unnecessary.

2.9.1 Learning Values for Belief Spaces

As well as having various methods for learning a model for Partially Observable problems, it is also necessary to determine which actions are best in each state. This is typically done using reinforcement learning techniques to learn a value function. As described in Section 2.4, reinforcement learning uses the reinforcement (utility value) associated with states to determine which actions to select. Ideally, in any given state we will select the action that will lead to the most reinforcement (i.e immediate reward and future returns). Recall that one choice for constructing the value estimate for states is to use *value iteration* which requires problems to have the Markov property (Bellman 1957).

Using belief states meets the Markov requirement, however, it introduces another problem. One sweep of value iteration requires each state to update its value by obtaining the value of each of its successor states. When using belief states this requires iterating across a continuous space (an infinite number of points) which is impossible. Fortunately, the value function for belief spaces can be calculated exactly for finite-horizons and approximated arbitrarily well for infinite-horizons using piecewise-linear and convex representations (Smallwood and Sondik 1973; Sondik 1973). This allows the value function to be treated as a finite set of multi-dimensional vectors. Using this approach there are a number of exact methods for calculating the value function (for example, Cassandra, Littman, and Zhang (1997) and Littman (1994b)). Unfortunately, such exact solution methods are still infeasible for all but very small problems and in practice approximate methods are used to calculate value functions (Lovejoy 1991; Littman, Cassandra, and Kaelbling 1995). One approximate method is Truncated Exact Value Iteration which terminates an exact method prematurely, in the hope that a near optimal policy has been reached (Littman 1994b). Tractable approximations are also possible based on generalised Q-learning rules, although there are situations in which these will fail (Chrisman 1992; McCallum 1995; Littman, Cassandra, and Kaelbling 1995).[6] Another approximation method for learning value-functions is SPOVA-RL (Parr and Russell 1995).

---

[6] There are special cases of POMDP's called Hidden-Mode Markov Decision Processes that may solved more efficiently (Choi, Yeung, and Zhang 2001).

On top of these problems, even the representation of a belief state may be intractable for large complex processes. Once again, approximations are used (Roy 2000). In this case, this can be done with the knowledge that any error due to the approximation will contract over time (Boyen and Koller 1998; Rodríguez, Parr, and Koller 1999). One method of reducing the overhead in maintaining beliefs is to use a compact representation. These rely on the state space being factored using a set of state variables (see Section 2.5). Boutilier and Poole (1996) describe a tree-based approximation method based on the work of Monahan (1982) which uses factored Bayesian-network representations. However, the efficiency of this approximation depends on appropriate pruning mechanisms. Boyen and Koller (1999) looks at a way in which efficient approximations may be made for belief states when factoring. Sallans (2002) uses gradient-ascent to learn a factored Bayesian network to calculate belief states along with approximate methods for learning value functions. Sallans (2002) successfully learns moderate sized partially observable problems, however, requires more training experience than some alternative methods. This is not unusual as known problems with using gradient-ascent include finding local-optima and the potential for long learning times (Neal 1990).

Bacchus, Boutilier, and Grove (1997) build on the work of Bacchus, Boutilier, and Grove (1996) which introduces new states into Markov Decision Problems to overcome non-Markovian rewards. One major modification is the use of a tree-like structured representation which again uses state variables to create a compact representation rather than enumerating the state space. Another modification is that states irrelevant to the optimal policy can be excluded from the representation. Even though the approach in Bacchus, Boutilier, and Grove (1997) is more efficient than an enumerative representation, a number of unnecessary variables may be introduced.

2.10 Markov-k Memory Approaches

An alternative to belief state approaches is to use Markov-k memory structures. Here the difficulty of learning value functions is overcome by converting a hidden-state problem to a Markov-k problem by using memory. However, it raises two other issues. The first issue is determining how much memory is required to predict a particular state. The second issue is the large number of possible memory sequences that can be created. Without memory sequences for all possible sequences of observations and actions in the environment, the agent may lose track of the current state and the problem can no longer be treated as Markov.

Different algorithms use different strategies to reduce the amount of memory created when learning hidden-state problems. It is common for these algorithms to rely on some form of state generalisation strategy to reduce the number of states represented internally (with an effect similar to factoring in belief states).

2.10.1 Fixed-length Memories

One of the simplest means to restrict the amount of memory used for learning hidden-state problems is to place a hard limit on the maximum length of memories. This type of limit has been applied to Neural Network methods for learning hidden-state by maintaining a fixed size window for previous sensory experience. At each time step, the Window-Q architecture used by Lin and Mitchell (1993) presents the current sensory experience to the network and the n previous experiences. In this case, n is the maximum length of memory that is maintained by the system. This approach has two problems. Firstly, the window must be large enough for the problem, if not the problem cannot be solved by the network. Secondly, for states where history is not required learning times may be increased because of the presence of the history inputs.

2.10.2 U-Tree

McCallum (1995)'s U-Tree is a decision tree based approach that introduces and extends memory sequences if it appears that doing so will help the system predict reinforcement (rewards or returns). Improvements in predictions are measured by a statistical test (the Kolmogorov-Smirnov test) to determine whether the reinforcement received with the added memory is from a different distribution than the reinforcement received without the added memory.

U-Tree can also introduce internal state distinctions based on features of the input space. Again this is done by testing whether there is a statistical difference between reinforcements received with and without the extra percept information.

U-Tree therefore allows a perceptually aliased state to be uniquely identified by either incorporating additional immediately available features or by introducing memory of recent states and actions. Because of the test for significance, unlike many fixed length memory approaches, U-Tree will not introduce new distinctions (new memories or features) if there is variance due to a stochastic environment. However, determining whether any differences are due to stochastic processes or not requires statistics to be collected and analysed. The statistics are collected by having a fixed size fringe under the leaf nodes of the multi-dimensional tree. The fringe nodes are additional nodes in the tree under nodes which have been included as "official leaves" by prior significance testing. They include additional features or memory that statistics are collected for. If a fringe node proves significant, then it is included in the tree as a new leaf and the fringe is extended one level further on that branch.[7]

---

[7] Chapman and Kaelbling (1991)'s G-algorithm also introduces features (but not memory) to identify states and shares U-Tree's requirement that the significance of items is detectable in isolation of other items. TRACA also has this requirement, but for pairs of items rather than individual items. Chapman and Kaelbling (1991) argue that input spaces should be designed orthogonally to avoid problems with these dependencies.

The maximum fringe size (or fringe depth) in U-Tree is important as it determines the length of memories possible. If a statistical difference is detected between one of the fringe nodes and its corresponding official leaf, the fringe node can be included as a new leaf in the area of the tree that has been determined as significant. However, if the fringe does not extend far enough to include such a node, the distinction will remain undetected.

2.10.3 Finite-state Automata

One way to represent the state transitions of systems is to use finite-state automata. Finite state automata have the advantage of being able to represent loops, however, suffer the severe disadvantage that in the presence of hidden-state entire sequences of subsequent states must be duplicated creating representations proportional to the size of the state space (Howard 1971). Rivest and Schapire (1994) and Rivest and Schapire (1993) present methods for learning large finite-state automata through input/output interactions. However, their method is restricted to deterministic problems which contain no noise. Mozer and Bachrach (1991) use a connectionist approach to learning update graphs for noisy environments called SLUG, but the learned structures can be difficult to interpret and scalability to complex environments is yet to be addressed.

Finite-state controllers are finite-state automata representing a policy where the states are actions and the transitions observations (Lusena, Tong, Sittinger, Wells, and Goldsmith 1999). Hansen (1998) uses a policy iteration algorithm to improve a finite-state controller by adding, changing and pruning automaton states (this is a search directly in policy space)(Lusena, Tong, Sittinger, Wells, and Goldsmith 1999). Meuleau, Kim, Kaelbling, and Cassandra (1999) also use a finite state controller with finite memory along with the Value and Policy Search (VAPS) algorithm of Baird and Moore (1999). Their algorithm performs gradient descent on an average error function with guaranteed convergence, however, not necessarily to the optimal solution.

2.10.4 Recurrent Networks

Another approach is to use a recurrent neural network with feedback loops such as Back-Propagation Through Time (BPTT) (Williams and Peng 1990) or one of the networks used by Elman (1990) or Mozer (1992). Theoretically, recurrent networks should be able to distill the relevant history information from the feedback. However, in practice this has proven difficult to achieve (Lin and Mitchell 1992; Lin and Mitchell 1993). Recurrent neural networks suffer from a variety of problems, from scalability problems (eg. requiring $O(n^3)$ storage space and $O(n^4)$ computations), dependencies on careful parameter tuning and requiring batch training (Williams and Zipser 1989; Mozer 1992; Fahlman 1991). Lin (1993) avoids some of the problems inherent in neural networks by using one network for each action

rather than a single monolithic network in architectures similar to SLUG (discussed in Section 2.10.3).

One variation on the theme of fixed length memories is that used in Ring (1994)'s Temporal Transition Hierarchies (TTH). The TTH network creates new nodes during learning which extend memory incrementally backward in time. The system allows for different length memories in different parts of the state space (arbitrary k), extending memory as necessary to improve predictions. However a "stay" operation is required for non-linear problems. Determining an appropriate number of steps to stay is one difficulty, in addition other parameters require problem-specific tuning to restrain the amount of internal structure created.

More recently attempts have overcome some of the problems common to recurrent neural networks. Hochreiter and Schmidhuber (1997) describe the Long Short Term Memory (LSTM) network which can store information for long time periods and with reduced training times. However, LSTM uses a supervised learning method and the number of required training examples is still very large when applied to reinforcement learning tasks (Bakker 2002).

2.11 Indefinite Memory

The approach used in LSTM to create memory is close to a different class of memory architectures. Rather than store information for an arbitrary amount of time, *indefinite memory* techniques store data for an indefinite amount of time (Holland 1990). Some such methods, called indexed memory methods, require a means of storing and clearing memory appropriately at the start and end of perceptually aliased regions. Learning when to store and clear memory presents a major difficulty for indexed memory approaches. Note, the triggers to store and clear memory may be the same as used in fixed length memory approaches (i.e those with fixed length memory but for arbitrary k in different areas of the problem), in which case the primary difference between the two approaches is that indefinite memory methods do not represent entire histories (Bakker 2002).

2.11.1 Schema Mechanism

Drescher (1991)'s Schema mechanism creates *synthetic items*, in addition to immediate primitive items (i.e features in the input vector), to explain hidden-variables in the environment. Since these variables are set and unset when appropriate conditions are detected, they can store information for indefinite lengths of time.

2.11.2 Classifier Systems

Cliff and Ross (1995) added temporary memory capabilities to a minimalistic Holland Style Learning Classifier System based on setting the values of memory bits. They demonstrated that this memory could be used successfully if the number of bits required is small, but that the approach would not scale well. Robertson and Riolo (1988) used triggered chaining operators in Learning Classifier Systems to create chains to predict letters in a sequence that required remembering earlier characters in the sequence. While it could do this successfully for some simple sequences, it failed for more difficult sequences.

2.12 Alternative methods for Learning Policies

While it is common to learn first a value function and use the value function to derive a good policy, it is also possible to search for a policy directly. Several approaches have been applied to learning policies for partially observable environments. Some of these combine value and policy search approaches.

2.12.1 Memoryless Policies

Littman (1994a) examined learning in hidden-state problems without using memory and found that in general finding optimal memoryless policies is difficult (NP-hard), however optimal policies can be found quickly for some problems. Loch and Singh (1998) later demonstrated that Sarsa(λ), an on-policy, multi-step method of learning Q-values, could learn policies for a number of hidden-state problems with lower computational costs than belief state approaches.[8] Jaakkola, Singh, and Jordan (1995) and Singh, Jaakkola, and Jordan (1994) overcome the problems of state-estimation associated with belief-state approaches by using stochastic policies.

2.12.2 External State

Externalised state approaches take advantage of the agent's environment to encode state or instructions that may assist it in its task (for example, Werger and Matarić (1996)). This use of external state is related to both the memoryless and indexed-memory approaches discussed in Sections 2.12.1 and 2.11. One method is to allow the agent to store and clear items from an external memory where the state of this memory is provided as an additional percept to the agent. By learning to use this additional percept an agent may avoid the need for internal memory and can learn a reactive policy. Peshkin, Meuleau, and Kaelbling (1999) used the VAPS algorithm of Baird and Moore (1999) with external memory to successfully solve a number of small partially observable problems.

---

[8] By using *eligibility traces*.

2.12.3 Evolutionary Search

Glickman and Sycara (2001) use evolutionary search to modify weights in a range of recurrent neural networks. They found the most consistent results were achieved with stochastic policies. While better policies were found than produced by the compared system (U-Tree), a much larger number of trials were required in the environment.

2.12.4 HQ-Learning

In some cases, having the same percept for different world states does not prevent the agent finding the optimal policy, if the action for those states is the same. It is only when different policies are required in different aliased regions of the state space that the aliased regions need to be distinguished. HQ learning distinguishes these regions using separate agents for each region. Each agent executes its policy in turn until it reaches a sub-goal after which the next agent takes over. Thus, rather than remembering prior sequences of states and actions, HQ-learning stores which agent is currently active (Wiering and Schmidhuber 1997).

HQ-learning depends on appropriate subgoals being found with a single agent learning the region for each subgoal. In problems with a small number of observations, the number of possible sub-goals is easily manageable, however, in larger state spaces the number of possible sub-goals will increase considerably. Also many observations will appear infrequently slowing down learning, this problem may be addressed by using some form of input generalisation. Finally, HQ-learning is not appropriate for maintenance tasks, where a desired state needs to be maintained over a period of time. A related approach is that of Pineau and Thrun (2002), which requires that a human provides a structural decomposition.

2.12.5 Levin Search

Schmidhuber, Zhao, and Wiering (1997) successfully use Levin search to search through the space of possible programs for a solution to a large maze problem. However, Levin search becomes less useful as the algorithmic complexity of the solution increases. To address this Schmidhuber, Zhao, and Wiering (1997) propose an adaptive version of Levin search which can incrementally improve using experience from previous problems.

2.12.6 Heuristic Strategies

It is possible to find a sub-optimal policy using a number of heuristic control strategies for POMDPs. One approach is assumptive planning, which assumes the most likely state with complete certainty, then constructs a deterministic finite-state automata which is searched for a path to the goal. During plan execution if the sequence of expected percepts is not experienced, the system replans based on the current most likely state (Nikovski and Nourbakhsh 1999). This approach is used by Nourbakhsh, Powers, and Birchfield (1995) and requires the provision of accurate topological maps which indicate useful landmarks. A similar approach is used by Simmons and Koenig (1995) but they enhance topographical information with metric information. Simmons and Koenig (1995) do a search using A* which associates directives with states in a Markov model. These directives are essentially the policy. Uncertainty is catered for by using a voting mechanism which takes into account the probability mass for each directive given any uncertainty in location. However, their planner does not take actions to gain information which could help the agent discern its position.

Another approach, used by Cassandra, Kaelbling, and Kurien (1996) and Zubek and Dietterich (2000), is to learn a policy for the underlying Markov problem, then use this policy to derive a policy for the partially observable case. Problems with these methods include the necessity to have a lengthy look-ahead in some cases if a delayed need-to-observe is to be considered in the final POMDP policy. Also, for problems where delayed opportunities-to-observe are important these approaches may result in poor POMDP policies.

2.13 Input Generalisation

As mentioned in Section 2.5, input generalisation requires the development of internal states which represent multiple world states based on a vector of state variables. Whitehead and Ballard (1991) point out that this is often a process of deliberate perceptual aliasing and that such a mapping can be achieved by omitting features (or state variables) in a state description. Consequently, an inadequate internal mapping for generalisation may also introduce a hidden-state problem. However, there may be problems where sufficient features to overcome perceptual aliasing are not immediately available (i.e without an action).

Many of the techniques discussed already for hidden-state problems are also used for input generalisation. For example, using tree structured representations. Sometimes these representations are used to tackle both problems simultaneously (eg. U-Tree).

One approach to the input generalisation problem is to introduce additional features to distinguish states (another is to introduce additional history). Tan (1991) does this taking into account the cost of obtaining additional features. Whitehead and Ballard (1991) operate within an active vision framework where multiple sensor readings are selectively taken to achieve an unambiguous input. Both these approaches have limitations, one of the most significant is that their tests to detect non-Markov states require a deterministic environment (Whitehead and Lin 1995). Chapman and Kaelbling (1991)'s G-algorithm also builds a tree representation, but uses a statistical test to detect non-Markov state distinctions which allows non-deterministic environments.

The tree structured statistical approaches used for input generalisation in the G-algorithm re-appear in hidden-state approaches such as U-Tree (McCallum 1995). An alternative to using tree structures is to use Algebraic Decision Diagrams (ADDs) which can create more compact representations, however, so far only for Markov Decision Processes (Hoey, St-Aubin, Hu, and Boutilier 2000). Another possibility is clustering techniques such as used by Mahadevan and Connell (1992). These are just some of the methods used to generalise across states. Others include various Decision Tree algorithms, CMAC's and Neural Networks (Sutton and Barto 1998). Comparative evaluations of some of these techniques for a set of input generalisation tasks are presented in Chapter 5.

One remaining issue that all learning techniques must deal with, whether for input generalisation or for hidden-state, is the exploration/exploitation trade-off.

2.14 Exploration

Exploration raises a number of difficulties. For off-policy learning techniques (such as Q-learning) once the value function approximation is sufficient to yield the optimal policy it may be desirable to eliminate exploratory actions. However, to be sure of finding the optimal policy, updates must continue for all actions infinitely. Furthermore, in changing (non-stationary) environments, continuing a small amount of learning is desirable to allow adaption (Singh, Jaakkola, Littman, and Szepesvári 2000).

Another problem with learning value functions is the time it may take to reach the goal for the first time and therefore achieve a reward which can be used to direct learning. An approach to this is to both penalise actions and encourage unexplored actions to be taken more often to avoid random walks (Koenig and Simmons 1996). However, even once the goal has been found, finding the shortest (or best) path to the goal often requires continued exploration. In these cases, exploratory action selection schemes often favour actions that have demonstrated prior usefulness (Singh, Jaakkola, Littman, and Szepesvári 2000).

Two approaches to exploration include ϵ-greedy methods and Boltzmann exploration. ϵ-greedy methods select the action which appears best most of the time, but with a small probability (ϵ) select an action with uniform random probability from the set of possible actions. This ensures that over time, all actions are tried infinitely often (Sutton and Barto 1998).

Using a Boltzmann distribution allows the favouring of actions that appear better based on the proportions of current value estimates. The amount of favourtism is adjusted by a temperature, low temperatures increase the favourtism of better actions, while high temperatures make action selection more equi-probable (Sutton and Barto 1998). The use of Boltzmann distributions is similar to the roulette-wheel approach of Goldberg (1989), which selects actions in proportion to their relative values.

Methods using ranking (called *restricted rank-based randomised learning policies*) also base action selection on value estimates. However, in this case the relative proportions of the values are not significant. The actions are ranked according to their value and selected according to a probability distribution over the ranks. This ranking scheme allows for both greedy action selection (always selecting the action with the highest value) and ϵ-greedy action selection (Singh, Jaakkola, Littman, and Szepesvári 2000).

A third method of determining when to take exploratory actions is by using Interval Estimation (IE) which stores a degree of confidence in estimates of the value associated with actions. By keeping track of the number of times the action has been executed and the number of times it has received the reinforcement 1 it is possible to calculate the probability of receiving 1 given that the action is executed. Actions can now be selected according to their confidence interval. As confidence intervals are initially 1, a high interval may indicate either that the action is good, or that there is little known about the action. This interval can be used in conjunction with another parameter to adjust the relative balance of exploration to exploitation (Kaelbling 1993).

These are just some of the exploration policies possible. Broadly speaking they can be divided into two categories: directed and undirected (Thrun 1992). Directed strategies seek to take exploratory actions to maximise information gain, rather than making selections entirely randomly. However, in problems with hidden-state, directed policies often fail because of assumptions that the world is fully observable. One approach to this is to associate exploration statistics with sequences of states and actions (McCallum 1997).

One final approach, related to IE, was used by Sutton (1991b) in experiments with Dyna-Q in which an estimate is kept of the uncertainty about a value estimate based on the time elapsed since the value was tested with real experience.[9] In this case, exploration can be assisted even further by using a world-model to plan exploration in areas of the state space. This planning is done on-line using hypothetical look-ahead.

2.15 Hypothetical Look-ahead

Hypothetical look-ahead involves taking simulated actions in a model of the world in place of real actions in the environment. It therefore favours computation in place of, at least partially, real-world experience. This can be beneficial in reducing learning times if the cost of experience in the real-world is high. One way in which learning times are reduced when using world-models with hypothetical experiences (hereafter just called look-ahead) is by their effect on the propagation problem (Whitehead 1989). Using temporal-difference methods of learning it may take many real experiences to propagate values back from a goal along solution paths. This problem can be addressed using look-ahead with world models. The use of world models to perform shallow look-ahead and speed up learning was used by both Sutton (1990) and Lin (1992).

A technique similar to look-ahead is to retain in memory previously seen examples from the environment. These examples can then be presented to the system between real experiences to speed up learning and address the propagation problem. This type of learning, called *experience replay*, has been used by Davis, Wilson, and Orvosh (1993) with Learning Classifier Systems and by Lin (1992) with neural networks. However, look-ahead has an advantage over experience-replay, in that it can also be used to find paths to goals even though taking that path has never previously lead to a goal. This ability has been demonstrated using Learning Classifier Systems by Riolo (1991) who used context sensitive look-ahead search to find such a path. Also, experiments by Thrun, Möller, and Linden (1991) using neural networks demonstrated the advantages of look-ahead for a robot approaching a rolling ball. However, Thrun (1992) notes that look-ahead over longer distances using neural networks is susceptible to local minima due to gradient descent. A final way in which look-ahead may be useful is in the achievement of multiple goals. Multiple goals are discussed in the next section and using look-ahead to support multiple goals is described in Section 3.5.4.

---

[9] Wiering and Schmidhuber (1998) explore a similar theme in which IE is combined with model learning in place of Q-learning.

2.16 Multiple Goals and Tasks

So far the discussion of policy learning has been in relation to a single goal, however, agents may have more than one goal. Matarić (1994) suggests that reinforcement learning (RL) systems often specify a monolithic goal and when using a monolithic reward function, multiple goals must be formulated as sequential subgoals of the reward function. Sutton (1991a) suggests a similar approach for his Dyna architectures in which goals can be explicitly encoded in the state space. However, the coding of goals in state spaces scales poorly leading to state size increases exponential in the number of goals (Tenenberg, Karlsson, and Whitehead 1993). Tenenberg, Karlsson, and Whitehead (1993) address the problem of multiple goals by having a module which learns the policy for each goal. Karlsson (1997) also uses a modular approach to achieve multiple goals in large state spaces.

Multiple goals may also be used to specify different tasks within an environment. If these tasks are related, then they can be used to discover inductive biases that can be useful for novel tasks in the same environment (Thrun 1996; Caruana 1997; Baxter 2000).

2.17 Summary

This chapter commenced with a discussion of the problems related to learning with hidden-state and input generalisation. It then looked at the formulation of problems as Markov decision problems and the associated methods for learning value functions which in turn determine action selection policies. A range of techniques were reviewed for both input generalisation and dealing with hidden-state. In addition, a number of other issues in learning were reviewed, including a discussion of exploration versus exploitation and dealing with multiple goals.

Many of these issues are raised again in the following chapters in relation to TRACA. In the next chapter TRACA's description commences at a high-level, followed by more detail in subsequent chapters along with experimental results. In the final chapter comparisons are made between TRACA and other systems selected from those mentioned above.

Chapter 3

System Overview

This chapter provides an overview of the proposed system. TRACA can be broadly divided into two aspects: a predictive model and a policy. The predictive model is a type of state estimator, which for a current state, observation and action, provides an estimate of the next state. The policy then maps that state into an appropriate action (ideally one that helps maximise the agent's return over time). TRACA's design is primarily based around the construction of the predictive model of the state space, so the first part of this chapter provides a high level discussion of model building in TRACA. This discussion is divided into two parts: temporal structures and non-temporal structures.

After model building our attention is turned towards constructing a policy using the model. A critical factor in determining the policy is credit assignment. As described in Chapter 2, credit assignment involves allocating utility value estimates to the appropriate model structures. These values should reflect the usefulness of being in the real-world state (or states) represented by the internal structures. In addition to determining a policy, credit assignment can also be used to guide construction of the model. The usefulness of an internal structure can be assessed based on its utility value estimate, to decide whether or not that structure should be retained in the model or removed. This decision is necessary for large state spaces as the size of the model must be constrained to be within workable limits. More detailed discussion of the role of utility estimates in creating internal structure is deferred to later chapters.

The next section begins explaining how TRACA's model is constructed using an example problem. The example requires temporal structures to be created which implement the short-term memory necessary to uncover hidden-state. Section 3.2 describes the temporal structures used to represent this first example. Section 3.3 then introduces a second example which is used to introduce non-temporal structures. These structures make state identifications using only immediately available sensory information. Both examples are used simply to introduce TRACA's operation. Further details on how TRACA uses non-temporal structures to discriminate states and develop generalisations are provided in Chapter 4 while details on using temporal structures to create short-term memory and solve hidden-state problems are presented in Chapter 6. Following our two examples, is the discussion of how policies can be associated with TRACA's model (in Section 3.5). In particular, how the model can be used to speed up credit assignment and for multiple tasks.

3.1 Model Learning

3.1.1 The N-maze Problem

The first example problem is a simple maze called the N-maze (derived from McCallum (1995)) which is depicted in Figure 3.1. This maze has two branches, each with a number of discrete states indicated by boxes. Each state has an associated observation which is a numeric label that the agent can observe when in that state. The boundaries of the maze represent impassable walls. In each state the agent can select one of four actions which will move the agent to the north, south, east or west neighbour of its current state, unless the action leads the agent into a wall, in which case its state remains unchanged.

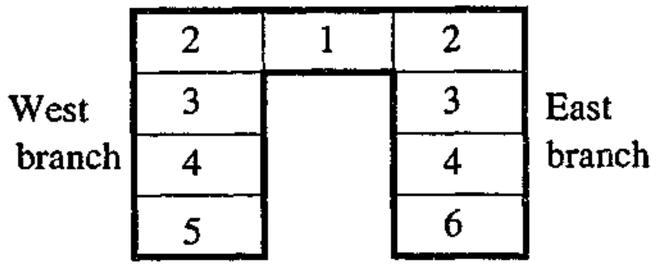


Figure 3.1: A simple N-maze with hidden state.

The problem the agent needs to solve is how to select the minimum number of moves that take it from an initial start state in the maze to the state labelled 6. Such a sequence of moves from one state to another is refered to as the *minimum length path* between states. A single trial in the maze consists of placing the agent at an initial state and allowing it to execute a sequence of moves until it reaches state 6. On reaching state 6 the agent receives a positive reward and is removed from the maze, all states other than 6 have a zero reward. Over repeated trials the agent must learn both the transition function and the reward function. For the purposes of the example the initial state is always the state with observation 1.

In the following discussion, the N-maze problem from Figure 3.1 is assumed to be represented using a localised sensor scheme; all information on the current state is located in one bit in a fixed length input string, with one bit for each label. Another possible scheme is to use distributed sensors, where information on the current state is represented by multiple bits in the input string which is the topic of Section 3.3.

3.1.2 Learning Transitions

The agent learns about the transitions between states during trials in the maze. For each label experienced the agent initially creates a *group* to represent it. Groups which represent only a single label are called *unary groups.* A group is matched when the observation associated with the agent's current state contains the labels the group represents. Within each group, nodes are created to represent estimated transition probabilities from one group to another given the selection of one of a number of possible actions. The group resulting from the node's action is called the node's *prediction.* Often transitions are non-deterministic, such that for a given observation a single action may result in different subsequent observations at different times. In these cases, a node is created to predict each of the possible subsequent observations.

Figure 3.2 shows how over a number of trials a node may record a transition estimate of 0.5 from the group representing the observation 4 to the group representing the observation 6 given that the action Move-South is selected. Nodes which record transitions from a unary group are called *unary nodes.*

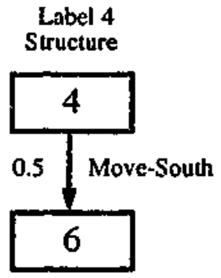


Figure 3.2: A group with a single unary node to record a transition estimate from label 4 to label 6.

When distributed sensors are used a number of unary groups may be matched simultaneously by an observation in the state-space (i.e multiple labels are present at the same time). In these cases, the observed labels represented by unary groups may be combined using a *composite group* called a *join group.* In small state-spaces, join groups may constructed in a manner which allows each world state to map to a unique set of nodes in a single group. In large state-spaces, this becomes infeasible and nodes representing a world state will be spread across many groups (in fact these groups are Q-morphisms)(Holland, Holyoak, Nisbett, and Thagard 1986). Join groups are described in Section 3.3.

3.2 Temporal Chains

For the N-maze example we are interested in another type of composite called a *temporal chain.* Temporal chains are used to store transition information based on sequences of observations and actions which occur consecutively over time. For example, during a trial in the N-maze, the agent may remember the sequence of observations and actions it has experience since the initial state . Remembering such sequences can allow the agent to keep track of its current state in problems with hidden-state where multiple world states have the same observation. Temporal chains are constructed incrementally, with each new increment extending the agent's memory to include an additional prior observation and action in the remembered sequence. They are necessary for the N-maze problem because many states share the same observation (this is the hidden-state problem from Chapter 2).

3.2.1 Chain Composition

To demonstrate how temporal chains help solve the N-maze navigation problem, two chains covering each branch of the maze are described. This description first explains how completed chains operate before examining the process in which they are created. A potential chain for the east branch of the maze is depicted as Chain 1 in Figure 3.3 and a chain for the west branch is presented in Figure 3.4. Chain 1 in Figure 3.3 extends back from position 6 (our goal position). Link 1 in this chain is a composite of the unary node and group representing the action Move-South from label 4, and the unary node and group representing the action Move-South from the label 3. Link 2 is a composite of Link 1 and the unary node and group representing the action Move-South from label 2, while Link 3 completes the chain as a composite of Link 2 and the unary node and group representing Move-East from the label 1.

Chain 1 allows the agent to know its location while traversing the east branch by keeping track of where it has travelled since the unambiguous state 1. It works because it extends beyond the region of labels with hidden-state between positions 1 and 6. Transitions to position 6 are recorded and as long as each label in the chain is passed in the correct sequence, the transition for the entire chain will always lead to position 6.

A similar chain can be used to represent the west branch, the main differences being an initial action of Move-West rather than Move-East and the termination of the chain's sequence of links at label 5 rather than label 6 (see Figure 3.4).

Each of these chains form the memory of the agent and, collectively with recent experience, allow the agent to keep track of its location when following the shortest path across the regions with hidden-state. Many other chains are possible for other paths, and techniques for controlling which paths are represented by chains are presented in Chapter 6. The type of memory implemented by these chains is Markov-k, where k is the length of the path represented by the chain.

3.2.2 Chain limitations

One limitation of Markov-k chains is that they are only useful when the path (sequence of observations and actions) they represent is followed exactly. Any deviation from this path destroys the agent's memory. For example, if our agent decided to deviate from the minimum

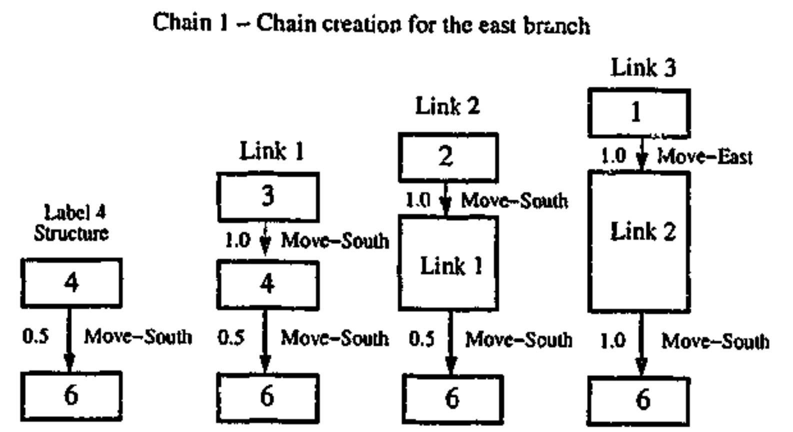


Figure 3.3: Internal state memory required for east branch of N-maze.

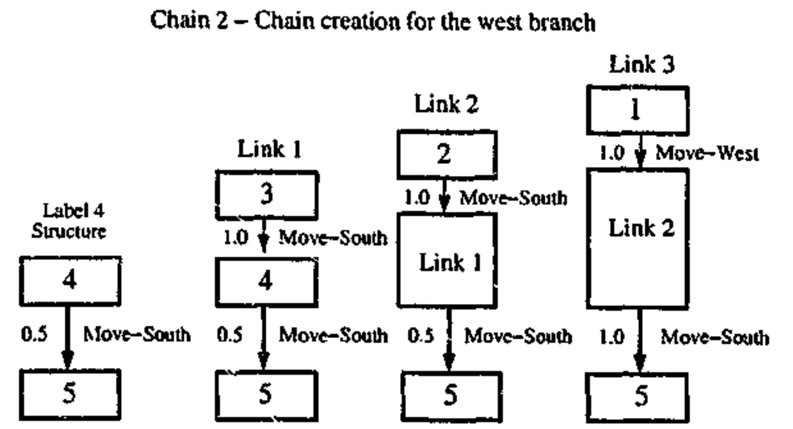


Figure 3.4: Internal state memory required for west branch of N-maze.

length path from position 1 to 6 on the east branch, by taking an action which moved it into a wall for example, then tried to continue directly south, Chain 1 would no longer be useful as it does not represent this alternate path. However, for situations where chain paths are followed, the agent may learn useful estimates of the transition probabilities for the underlying Markov model and of the utility of the states in that model.

In cases where the chain paths are not followed, the agent may not be able to identify its current state. For example, without further chains, any deviation from the path of either Chain 1, when on the east branch, or Chain 2, when on the west branch, and the agent may become lost. A similar problem arises if the agent is placed in a state other than 1 at the start of a trial. This is a potential problem for agents which wish to take some exploratory actions while seeking rewards and is discussed further in Chapter 6. Furthermore, if the initial starting state for trials alternates between any of the states without a unique observation, such as those with an observation of 2, 3 or 4, the agent will always be initially lost regardless of what chains it may have available. In these cases, the agent must either guess its location or try to move to some landmark position that unambiguously identifies the current state.

3.2.3 Using Reinforcement

A decision made in a state without a unique observation may still be better than random guessing if the agent uses information from prior experience. One possibility is to keep track of prior rewards received. Take for example, an agent with prior experience in the N-maze which has just been placed at the start of a new trial in one of the states with the observation 4. If there is no penalty associated with reaching state 5 from a state labelled 4, the agent may decide to move south knowing that there is a 0.5 probability that this action will lead it to state 6 and the corresponding reward. However, if there is a large negative reward associated with state 5, it may decide that a 0.5 probability of receiving this negative reward warrants travelling the extra distance north to unambiguously determine its location.

Given the hidden-state, the agent would need a large number of trials in the maze to have sufficient experience to make the best decisions possible in all cases. Furthermore, it needs to make the best use of these trials and the information received from them. The collection of this information can be complicated both by the agent's own reward seeking behaviour and the influence of construction processes for chains (or other structures) within the agent.

One effect of the agent's goal seeking behaviour may be to influence the amount of experience it has with particular aspects of the maze. For example, if the agent is greedy, as soon as it discovers a way to reliably reach the high reward state of 6, it may neglect exploration of other aspects of the maze. This lack of experience in various areas of the maze may effect the agent's performance when it encounters novel situations, and raises all the issues of exploration which were discussed in Section 2.14.

These are some of the behavioural issues relating to learning from experience. How the agent's construction of an internal state space representation effects the recording of information during learning requires a range of issues to be addressed. These issues involve trade-offs between memory, computation and the amount of training experience required.

3.2.4 Stochastic Problems

One other problem for chains arises when the underlying environment contains stochastic (non-deterministic) transitions. In stochastic environments (eg. problems with invisible hidden-state (McCallum 1995)), no set of past features can help improve predictions and attempting to achieve deterministic state transitions is futile. Even in some deterministic environments building internal structures to completely uncover the true Markov model may prove too expensive to be worthwhile. Finally, in problems where it is practical to construct internal structures to reveal the underlying Markov model, until these structures are complete the agent may need to make decisions based on its incomplete model.

3.2.5 Summary of Temporal Chains

Temporal chains implement the memory mechanism that allows TRACA to improve its ability to predict state transitions in environments with hidden-state. Further details on temporal chains and their creation are provided in Chapter 6. One issue which was omitted from the discussion of chains is the effect of using a distributed sensor scheme. These schemes lead to a number of additional problems and opportunities which is the topic of the next section.

3.3 Using Joins

This section introduces *join groups.* These are another type of composite structure which can be created when distributed sensors are used. Unlike temporal chains, which use information from previous sensory input, join groups combine sensory information that is available on multiple features of the current state. The use of a distributed sensor scheme rather than a localised sensor scheme allows the system to create generalised structures which may result in a smaller internal state space than is possible when using a localised sensor scheme.

The problem in Figure 3.5 is used to demonstrate join groups. In this problem, there are four corridors each with two states. The southern state in two of the corridors has the observation 5 while the other two have the observation 6. In each trial one of the four north states will be selected with equal probability as the initial state for that trial. For simplicity, in this problem the only possible action is Move-South. The task is to construct groups so that each initial state has a set of nodes which uniquely represent transitions from it to other states.

The four possible state transitions for the new problem are presented in Figure 3.6. The arrow in this figure from each possible initial state is labelled with the actual transition probability to each resulting observation given the action Move-South. Each of the four initial states is identified by an observation which may include one or more labels. Observations are represented using distributed sensors, with each label having a corresponding unique bit in the agent's input string. If the label is present the corresponding bit contains a 1 and is *on* and if the label is absent, the bit contains a 0 and is *off.* Groups are matched when the labels they represent are on.

Figure 3.6(a) and 3.6(b) show the observations for the two world states that may precede the observation 5. The west preceding state has labels 4, 7 and 8, while the east state has just label 4. Taking the action Move-South in either of these states will lead to the observation 5 with probability 1.0. Figure 3.6(c) and 3.6(d) show the observations for the two states that may precede the state labelled 6. The west state has labels 4 and 7 while the east state has labels 4 and 8. Taking action Move-South from either of these leads to the observation 6 with probability 1.0. Reaching either observation 5 or 6 ends the trial and the agent is removed from the maze.

The problem for the agent is to reliably predict for all four possible initial states what the resulting observation will be given the selection of the action Move-South.[1]

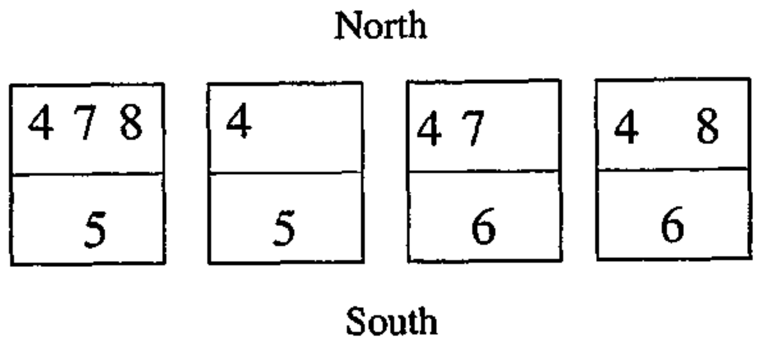


Figure 3.5: The four corridors of the new maze problem.

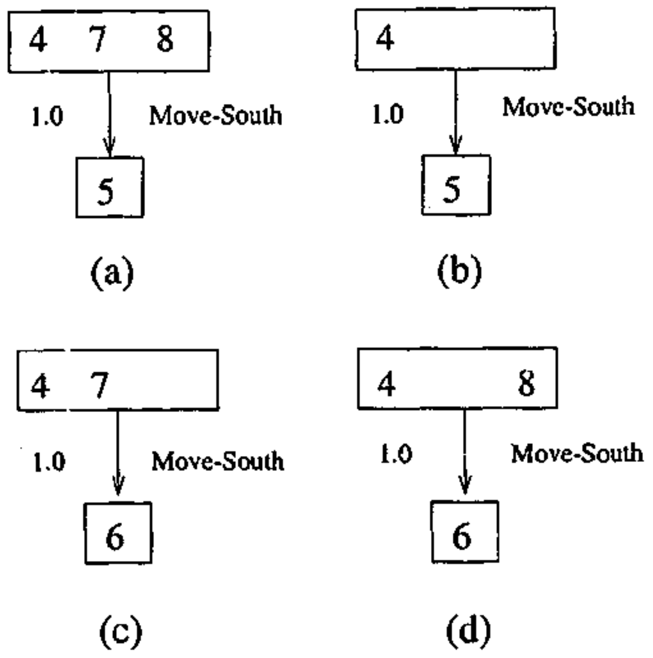


Figure 3.6: Transitions occuring in the modified maze.

3.3.1 Limitations of Unary Groups and Nodes

During initial trials in the maze of Figure 3.6 the agent will create a number of unary groups. The combined nodes in these groups simply keep track of the transitions from one label to possible subsequent labels. Figure 3.7 depicts the transition estimates created using these nodes and groups during a set of initial trials. There are two independent unary nodes associated with each label; 4, 7 or 8. Each of these nodes has recorded an approximate 0.5 probability of observing either label 5 or label 6 given the action Move-South. Since we are interested in the transition between world states, and not individual labels, it is apparent that the unary groups and nodes alone are insufficient to represent the true Markov model underlying this maze problem. Additional groups and nodes are required which combine the evidence of multiple labels to identify individual world states.

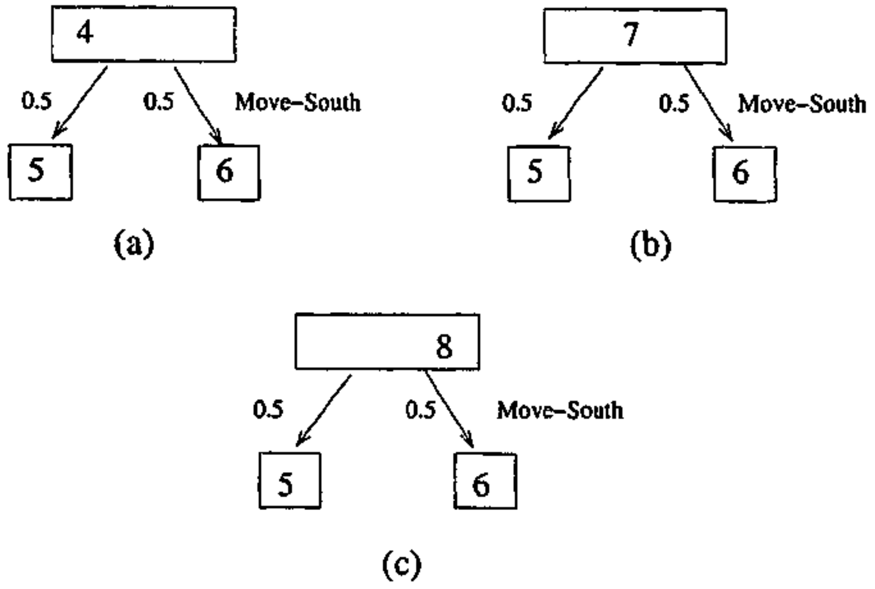


Figure 3.7: Transition estimates calculated using only unary groups.

3.3.2 Effects of Introducing Join Groups

Join groups are combinations of either unary groups or other join groups. A join always combines two other groups to form a logical AND. For example, if the two joined groups each represent the presence of a single label a join that combines them will represent the presence of both labels. In all cases, a join is the *superior* of the two groups it combines which are its *immediate subordinates.* A group's *indirect subordinates* includes the immediate subordinates of its immediate subordinates and so on recursively (Dawkins 1976).

Like unary groups, a join group contains nodes which represent transitions from the join group to other groups. Figure 3.8(a) shows the result of the agent introducing a join group which combines the unary groups associated with labels 7 and 8. This join group has nodes which record a transition to another join given that both the labels 7 and 8 are present simultaneously. In this situation an action of Move-South will always result in label 5, therefore the transition estimate associated with the corresponding node in the join group is 1.

---

[1] This is an unusual formulation of the XOR problem.

3.3.3 Suppression of Subordinates

Join groups *suppress* updates to estimated transition probabilities in their subordinates. A join group combining the unary groups for individual labels 7 and 8 such as represented in Figure 3.8.(a), causes the nodes in the combined unary groups to only make updates to their transition estimates when labels 7 and 8 do not appear simultaneously. In these situations, the label resulting from action Move-South will always be 6, and the estimated transition probability for nodes in these groups will approach 1, as indicated in Figure 3.8(b) and 3.8(c). Estimates of either transition probabilities or utilities which are directly affected by the use of suppression, are refered to as *dependent* estimates, since their value is dependent on their being suppressed by superior groups. Values which are calculated independently of suppression are called *independent* estimates.

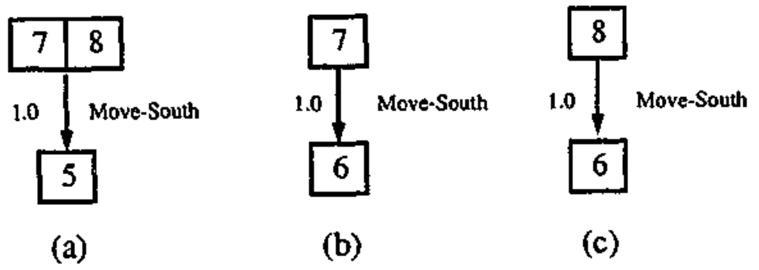


Figure 3.8: Effects of a join group on the transition estimates of nodes in its subordinate groups.

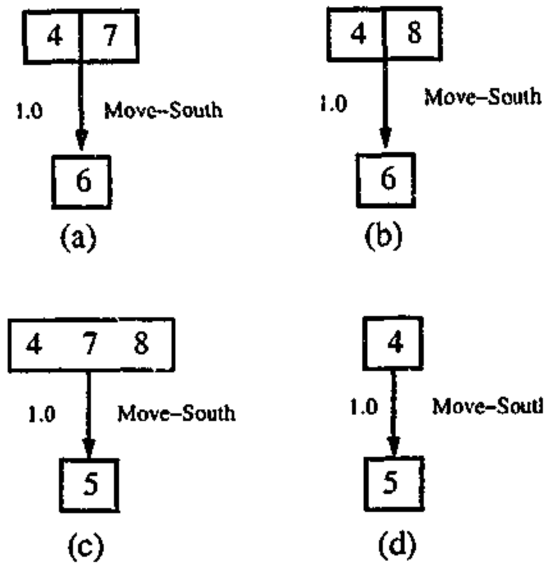


Figure 3.9: Individual groups and combinations of groups required to solve the maze.

With the unary groups, join groups and nodes developed so far the agent can accurately predict whether an action of Move-South will result in label 5 or 6 for all situations except when the only label present is 4. As no join group suppresses the unary group associated with label 4, and label 4 is always present, the nodes for this unary group will predict an equal likelihood of arriving at label 5 or 6 given action Move-south. To allow the agent to accurately predict which label will result if only label 4 is present, the unary group representing label 4 also needs to be suppressed in some situations. Allowing the nodes in the unary group for label 4 to reflect the certainty of a transition to label 5, given that 4 is the only label present, can be achieved by suppressing the unary group for label 4 whenever there is a transition to label 6. This transition occurs whenever label 4 appears simultaneously with exactly one other label. The correct suppression requires three new join groups to be introduced. The resulting join structures are presented in Figures 3.9(a), 3.9(b) and 3.9(c). Two of these are based on combinations formed by join groups which *suppress* the unary group associated with label 4 (see Figure 3.9(d)) when either label 7 or 8 is present (see Figure 3.9(a) and 3.9(b)). The other combination suppresses its subordinates, which are the groups represented by Figure 3.9(a) and 3.9(b), when both labels 7 and 8 are present (see Figure 3.9(c)). Note that this is just one possible solution for this problem. Other sets of structures may be created depending on the order of presentation of the training data and randomness within the selection process for nodes used as subordinates in joins.

3.3.4 Computational Completeness

The combinations illustrated in Figures 3.8 and 3.9 are sufficient to correctly represent all world states encountered in the maze in Figure 3.6 given the single action Move-South. They also demonstrate how TRACA can represent NOT and XOR given appropriate inputs. In this case, the solution relies on a label such as 4 always being present. The use of suppression to represent NOT eliminates the need for different types of logical gate connections between structures and in in doing so reduces the search space for solutions by reducing the number of possible logically equivalent combinations of observations (see Section 4.2.6 for a more detailed explanation of this). The result is that representations developed by TRACA are relatively simple, yet TRACA is still computationally complete.

3.3.5 Summary of Join Groups and Nodes

TRACA's representation is constructed from groups, each of which may contain a number of nodes. Groups represent features or combinations of features which occur in the environment while nodes represent transitions between groups. In the example problems presented above labels are the environment features. Each feature has a corresponding bit in the agent's fixed length input string which is either on or off depending on whether or not the feature is present in the current world-state. The bit positions represented by a group form the *direct condition* of the group, which is *matched* when all the represented bits are on. Unary groups represent an individual bit, join groups represent the bits of all their indirect subordinates. However, because of suppression a group and its nodes are also effected by its superior groups. The *complete condition* of an group is only matched when its direct conditions are matched and any of bit positions which will cause it to be suppressed are off.

Nodes can be viewed as rules whose conditions consist of their group's conditions as well as their action. A possible *estimated transition probability* (ETP) that can be stored by nodes is one which records the probability of the node's prediction being matched given that its action is taken when its group's complete condition is matched (therefore it is a dependent value).

The set of complete conditions for all groups defines the agent's internal state space. For example, the groups representing the individual labels and combinations of labels presented in Figure 3.9 (and their hierarchical relationships) define the internal state space for the maze problem in Figure 3.6.

3.4 Visualising Groups and Nodes

This section presents two figures which represent how TRACA implements nodes and their groups. Figure 3.10 shows the encapsulation of two nodes into a unary group. Figure 3.11 shows how this grouping, and the hierarchical relationship of superior join groups to their subordinate component groups, allows the superiors to suppress all nodes in subordinate groups for all possible actions, not just a single action as in the situation in Section 3.3.

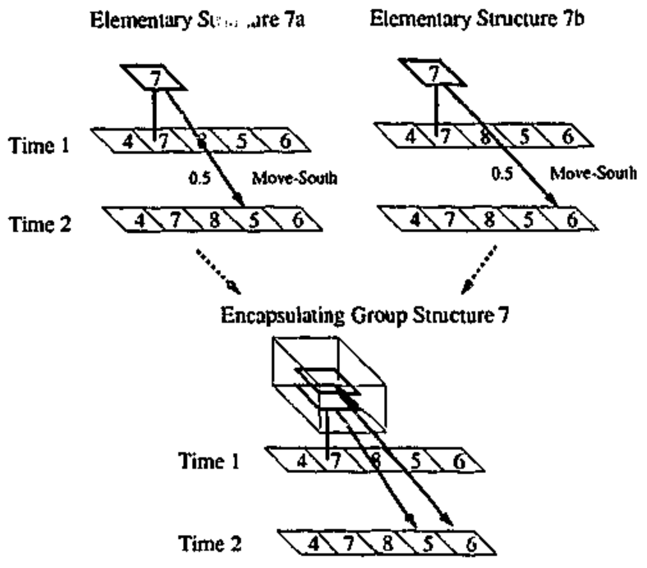


Figure 3.10: The group encapsulating two unary nodes created for label 7.

This completes the high level description of the groups and nodes used to construct TRACA's internal model. Further details of these structures are presented in Chapter 4 for joins and Chapter 6 for temporal chains.

3.5 Policy Learning

This section discusses the use of reinforcement learning with TRACA's internal state model to construct policies for action selection. TRACA learns a policy indirectly by using

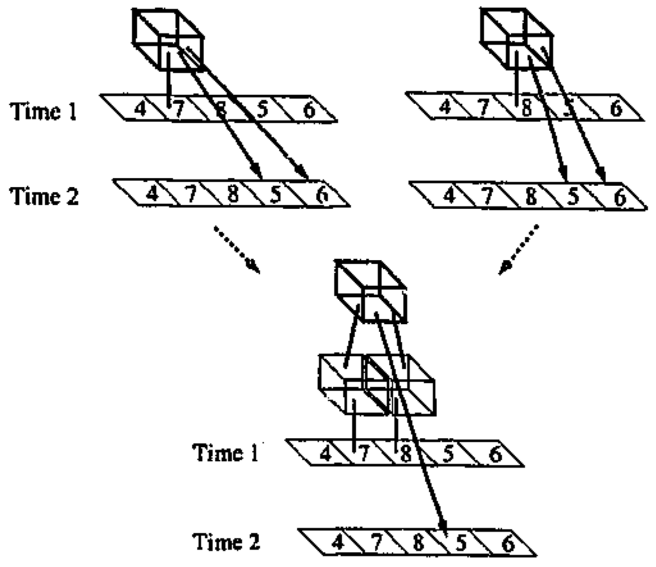


Figure 3.11: A hierarchy with a single join group.

reinforcement learning to learn a value function estimate. There are a variety of possible techniques for learning value estimates, and specifics of the techniques used by TRACA are left until Chapter 4. This section focuses instead on various interactions between policy learning and model learning.

In TRACA each node maintains an estimate of the utility of taking its action given that its group's complete condition has been satisfied (this is a dependent value). These utility estimates are then used to determine the policy which drives system behaviour.

However, in addition to driving behaviour utility estimates may also be used to prune states from the agent's internal-model. TRACA uses utilities in this way to remove temporal chains with low utility relative to the resources they require (see Section 6.4.3). However, it may be possible to use utilities to prune join structures also (see Section 8.4).

The learned policy may also have other effects on the model. One of these effects is discussed in the following section.

3.5.1 The Effects of Policies

Ideally we would like our agent to travel and learn about different areas of its environment, particularly in the early stages of learning. Often this is managed by some exploration strategy which reduces exploratory actions over time in favour of actions which more reliably lead the agent to rewards. Typically exploration is never stopped entirely as some fruitful area of the environment may remain undiscovered, or the environment may change in a way that requires a change in the agent's behaviour (policy). In TRACA, transition and utility estimates are recency-weighted to allow adaption to such changes in the environment.

However, when using recency-weighting, frequent selection of actions which lead to rewards can affect the estimated transition probability maintained by nodes.

Consider the N-maze problem in Figure 3.1. If a high reward is associated with the state labelled 6 and a low or negative reward with the state labelled 5, the agent during learning may come to prefer the east branch. After discovering a chain to represent the east branch (such as the one in Figure 3.3) a greedy agent may follow this path in almost every trial (depending on the exploration strategy) even before completing a temporal chain to uncover the hidden-state on the west branch.

Now each time the agent traverses the east branch the estimated transition probabilities are updated for nodes in groups along the path of the branch. The ETP maintained by the temporal chain for predicting the label 6 will be correct at 1.0. However, because of recency weighting, eventually the ETP for the node with a condition of label 4, prediction of label 6 and action Move-South, will also approach 1.0. On the other hand, the ETP of the corresponding node to predict label 5 will approach 0. This is an incorrect representation of the real environment. If our agent is now initially placed in a state with the label 4, the agent will mistakenly assume that taking action Move-South will almost certainly result in the state labelled 6. This is not true, it is just as likely to result in the state labelled 5.[2]

3.5.2 Improving Predictions

It is desirable to alleviate this type of transition estimate probability distortion as much as possible. Since in this case, it is the development and following of temporal chains that leads to the problem, a solution is to prevent updates to ETPs which would otherwise be distorted when a chain is being followed. Doing this for all affected rules is difficult, however, it can be easily done for some rules using the suppression mechanism.

Let us continue our example, in which a chain is being used to navigate down the east branch of the N-maze in Figure 3.1. The first link of this chain (see Figure 3.3) is constructed containing the node which predicts that label 6 will follow label 4 when the action Move-South is taken. This node belongs to the group of nodes which all have their conditions matched by label 4. The relationship between chains and the subordinate nodes used to construct their first links is one of superior and subordinate (see Section 3.3.2). Therefore, Link 1 in Figure 3.3 is in fact the superior of the group containing all nodes in the internal state representing label 4. This includes the nodes which predict label 5 and label 6. The chain's link is superior to the subordinate group, so once the chain's path has been followed to label 4, the chain can suppress updates of ETP's in the subordinate group. This prevents incorrect updating the ETP's of the two nodes which predict labels 5 and 6. This may not entirely eliminate distortions due to following the chain, however, it does improve the situation.

---

[2] This is closely related to the problem of over training as discussed by Lin (1992) in relation to experience replay.

3.5.3 Relating the Model to the Policy

The success of using reinforcement learning to learn a policy requires that the agent's internal model contains sufficient internal state distinctions to allow all relevant utility estimates to be calculated. Some learning systems use the utility of state distinctions as the sole basis for retaining the distinction in the agent's internal model, arguing that any structure which is not relevant to the task should be removed (for example, (McCallum 1995). This argument assumes that the agent is only ever going to follow a single policy based around the rewards associated with a single task. TRACA, on the other hand, attempts to separate the state distinctions in the agent's internal model from any single task. This separation is intended to prevent TRACA learning models which are only useful for a single task.

TRACA's structures are intended to support multiple tasks, so there must be a means by which multiple policies can be learned, one for each task. Since TRACA learns policies indirectly via utility estimates, the current policy can be changed by propagating the internal model with a new set of utility estimates appropriate to the next task. However, it is desirable to avoid having to gain these new estimates from scratch, which would require expensive experience each time the task changed. One way in which this can be avoided is to store the rewards associated with each task in an internal state that represents the achievement of that task. Now each time the task is applicable, the rewards for that task can be propagated back through the model replacing the current utility estimates stored by nodes. The mechanism proposed to achieve this in TRACA is hypothetical look-ahead.

3.5.4 Hypothetical Look-ahead

Hypothetical look-ahead was introduced in Section 2.15. It entails, in part, using simulated experience in a model to propagate values through the states of the model. This process is implemented in TRACA in a manner similar to how it is described in classifier systems as proposed by Holland (1990) and used in experiments by Riolo (1991). However, Riolo (1991) only applied look-ahead using a mechanism which does not directly support input generalisation. In this sense, his mechanism is similar to using a localised sensor scheme. In Chapter 5, TRACA demonstrates the capability to do look-ahead using distributed sensors in conjunction with TRACA's mechanism for input generalisation.

TRACA's look-ahead planning aims to avoid criticisms aimed at the reactivity of classical planning approaches and their coherence (e.g by Brooks (1991)). TRACA does this by implementing a hybrid approach which offers two benefits (Matarić 1997). Firstly, planning as proposed by Holland (1990) is context-sensitive, which means it is not necessary to learn utility values for every state in the model, just those from the current state to the goal. Secondly, TRACA's parallel rules allow the possibility for competing behaviours within plans. For example, if executing a plan leads to a situation where there is an obstacle, a reactive rule which predicts a collision with the obstacle can prevent the execution of the original plan by sending a negative value in support of the planned action. This leaves an evasive action as the prefered action and may also force replanning once the negative situation has been avoided.

Multiple tasks (or policies) in TRACA are implemented by associating a state variable with the completion of each task's goal. The group which represents this goal state stores a reward value for use with look-ahead planning. When the goal is activated, hypothetical look-ahead (simulated experience) is used to propagate the goal's reward value back through the model. These hypothetical values are then used in support of actions for effector selection to allow the following of a path to the goal.

Past attempts to use look-ahead planning techniques with Back-propagation neural networks have presented difficulties, which were in large part attributed to local minima due to gradient descent (Thrun 1992). While TRACA may be susceptible to local minima, it does not use gradient descent, and TRACA's internal structures are updated with a much greater degree of independence than the Back-propagation neural networks used by Thrun (1992). The hypothetical look-ahead planning experiment presented in Chapter 5 demonstrates the ability of TRACA to support look-ahead planning and multiple tasks.

A possibility for even more efficient planning is to restrict the normal spreading activation of look-ahead to include only the more probable state transitions (Holland 1990). This would allow a form of planning similar to the assumptive planning used by Nourbakhsh, Powers, and Birchfield (1995), however, would reduce the ability of the system to switch to alternative paths without replanning. This possibility is raised again in Section 8.4.

3.6 Chapter Summary

The structures we have seen so far are all that are required to build TRACA's internal model. The use of composite groups, such as joins and temporal chains, allows two methods of discriminating states. Temporal chains extend back in time, using prior features to distinguish one state from others, while joins combine immediately available features in each of the states to be distinguished.

In both cases, there are situations in which they may fail. Joins will fail if the information necessary to distinguish two or more states is not present in the immediately available input string (for example, the N-maze in Figure 3.1). Temporal chains will fail if no amount of history or combination of previous features can improve predictions. However, an agent without prior knowledge of its environment will not know which type of structure is appropriate and must consider both possibilities.

TRACA's learned internal model is intended for use on multiple tasks. This is achieved using simulated experience in the model, along with memory of the rewards received on completing a task, to learn the policy appropriate to the current task.

The following chapters describe TRACA's structures and operation in more detail and present experimental results. These chapters first present details and experiments using joins (called the *basic system*) before doing the same for temporal chains. Once both joins and chains have been described and demonstrated experimentally, a final experiment is provided that demonstrates these two types of structure being used together on a single task.

Chapter 4

The Basic System

This chapter describes in detail TRACA's basic system which creates unary and join groups for state discrimination and input generalisation. The basic system does not include temporal chains for representing hidden-state, these are described in Chapter 6.

Input generalisation is necessary in state spaces which are too large to be modeled exactly. In TRACA this is achieved by incrementally constructing multiple default hierarchies during learning. These hierarchies capture important relationships in the environment while omitting unimportant details, typically by excluding irrelevant features.

One method of deciding whether features, or combinations of features, are relevant is to assess them based on their utility in performing in a particular task. TRACA, however, is intended to perform multiple tasks. Irrelevant features are therefore those which are not relevant for *any* task the system must perform. This makes assessment using task dependent utilities more difficult.

One alternative measure of relevance is to base assessments on the ability of structures to reliably predict world states. Using this measure, structures which make unreliable predictions are irrelevant in the presence of more reliable predictors. This is the problem addressed by TRACA's basic system, discovering structures which best predict world states. Only groups containing nodes which provide more reliable predictions than currently existing nodes are retained in the system. In addition, once a group is reliably predicted in a situation, the search for further structures to predict it in that situation can stop.

In Section 4.2.1 TRACA's use of join groups is explained using a simple maze example to illustrate construction of its network. The context of this example is then used in Section 4.2.3 to explain the role of higher level groups in the system and how these are used to achieve hierarchical control. Representing logical NOT is discussed in Section 4.2.5. Section 4.2.7 describes details of how value estimates are calculated and used to constrain network growth. Experimental results using TRACA's basic system are presented in Chapter 5 where they are compared to results obtained by other learning systems.

4.1 High-level Description

TRACA receives information from the environment through an *input interface* while the decision of the system to take some action in its environment is passed out through the *effector interface.* TRACA is a reinforcement learning agent so a further input is the reinforcement signal indicating the agent's success or failure at achieving its goal(s) (Sutton 1991b).

TRACA's input interface consists of an individual detector for each bit position in a fixed length bit vector. The current observation of the environment is presented on the input interface using fixed length bit strings. For each string received, the agent is capable of selecting an action before a new observation input string is presented based on the environmental state resulting from the action. In this respect TRACA is similar to Holland style Learning Classifier Systems which also receive environmental inputs (other than reinforcement) as bit strings (Holland 1975).

The effector interface consists of a fixed number of effectors, one for each possible primitive action the agent may take. The remainder of the network consists of a variable number of *predictor* nodes and their containing groups. Predictor nodes represent transitions between groups given a current state and the selection of a particular action. It is these nodes and their containing groups that are constructed incrementally during learning.

Figure 4.1 provides an overview of TRACA. At the bottom of the figure are the environment interfaces through which input strings are received and actions output. Within the agent are four processes. The first process is *internal state discovery* which is responsible for creating new groups which represent internal states. The second process is *internal state chaining* which creates nodes to link groups matched at one time step to groups matched in the next timestep given the actions selected by the agent. The third process *attributes rewards* received from the environment to nodes within the agent's internal state space which were responsible for achieving the rewards. The fourth process is the *pruning* (removal) of created internal states. In the basic system, pruning is based on assessments of how well nodes contained within join groups predict other groups.

4.1.1 Groups

There are two types of groups: *unary* groups and *join* groups. Each detector in the input interface has one associated unary group. Groups participate in hierarchies in which join groups form binary combinations of two other groups which are its *immediate subordinates.* Subordinates may be either unary groups or other join groups. If the subordinate is a join group, then its subordinates are also subordinates of the superior join group and so on recursively (Dawkins 1976). Nodes in a join group's immediate subordinate groups (i.e those to which it has a direct connection) which have the same action and prediction as a node in the superior join group are called the *equivalent subordinates* of the superior node. Unary

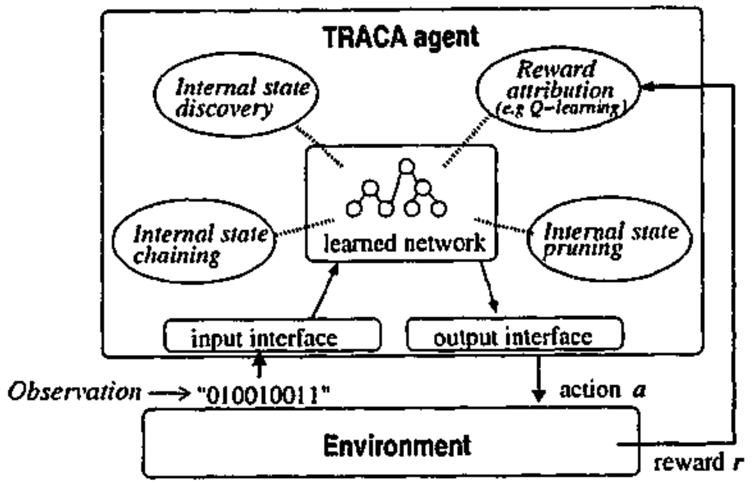


Figure 4.1: System Overview

groups are *matched* when the input string matching the detector they represent has a 1 in the detector's corresponding bit position. Joins of unary groups, and joins of joins, are matched when their subordinate groups are matched.

The matching of superior groups based on their subordinates is implemented using message passing. In the case of matched messages, unary groups pass the message to their immediate superiors. These groups in turn send messages to their superiors which are passed up to top of the various hierarchies. In this way, matched messages are propagated through the entire network by being passed up hierarchies by subordinates. However, there are also messages which are passed down hierarchies. These messages are initiated by groups at the top of hierarchies and are propagated to all the subordinates of the initiating group. All state changes to groups as a result of message passing are accessible to the nodes they contain.

There are two types of *suppression messages* which are passed down hierarchies of groups. The first of these is the *create suppression* message which is used to restrict the creation of new groups. The second is the *support suppression* message which is used to shift control to nodes in groups higher in the hierarchy. Support suppression also allows TRACA to represent logical NOT (see Sections 3.3.4 and 4.2.5 for details on suppression).

4.1.2 Messages and Links

Messages between groups are passed along links which exist between superior groups and their immediate subordinates. Predictor nodes also have links to groups, separate from the links between groups.

Each predictor node has two links, a link to the node's predicted group and a link to an effector. Effectors represent the actions that can be selected by the system. The links from nodes to effectors are used to send utility estimates maintained by nodes as support for particular actions. This support is then used by TRACA for action (effector) selection (see Section 4.2.7). Unlike the links from groups to other groups, the link between a node and its predicted group is relatively independent of the node's position in any hierarchy. If we view links between groups and their immediate superior and subordinate groups as vertical links up and down a network hierarchy, the links between predictor nodes and their predicted groups can be visualised as horizontal links across the network between hierarchies. These links connect one part of the network to another across time based on input strings received and actions selected. Both the horizontal link to the predicted group and the link to an effector are used by a node to update the Estimated Transition Probability (ETP) and utility estimate it holds.

4.1.3 Action Selection

During learning TRACA can select, either deterministically or probabilistically, an effector based on the support received from nodes. If no effector has support, an action is selected with uniform random probability. For probabilistic effector selection, TRACA uses the roulette-wheel method from Learning Classifier Systems (Goldberg 1989). This is just one solution to the problem faced by all learning systems of when to take exploratory actions versus when to exploit current knowledge (Kaelbling, Littman, and Moore 1996). Effector selection in TRACA is discussed in detail in Section 4.2.13. To allow utility estimates (and therefore effector support) to be based on actual experience, nodes are given a small number of trials before they are eligible to send support to their associated effector.

4.1.4 The Role of Groups and Nodes

Groups act as containers for nodes representing the conditions under which the node is eligible to send support for its associated effector. Nodes, for their part, store estimated transition probabilities to other groups and the estimated utility value of their associated action given that their group is matched. The two associations nodes have with groups, are the group as a container and the group as a prediction (a node's prediction may or may not be the same group that contains it).

Nodes, combined with their groups, represent probabilistic SRS (Situation-Response-Situation) rules. These rules are of the form (s_a, r_x, s_b, p, u) where s_a is an initial state, r_x is a response (action) and s_b is the state that results from taking the action r_x while in state s_a. p is the probability of the node's prediction being matched at time t_n given that at time t_{n-1} its containing group was matched and its action was selected. u is the estimated utility of the node.

The rules represented by nodes can be interpreted as:

```
if we are in $s_a$ and action $r_x$ is taken
    then we will be in $s_b$ with probability $p$
```

Each node corresponds to one of these rules. The antecedent comprises being in state s_a and having the response (action) r_x selected, the consequent is the transition to state s_b. The linking of groups through the predictions of rules (represented by nodes) allows chains of rules to form across varying sequences of situations.

The rules are probabilistic because the consequent may not always follow the antecedent. This could be due to a stochastic environment, or because a rule is a general rule and the antecedent, represented by the containing group's unary subordinates, does not capture all the conditions required for reliable prediction. The presence of groups whose conditions are general enough to be matched by many world states allows for many different groups to be matched by the same world-state, in which case the rules represented by the nodes in those groups all operate in parallel (they all send support and make predictions depending on the effector selected).

The probability of a node's predicted group being matched at t_{n+1}, given that the predicting node's antecedent group was matched and its action selected at t_n, is estimated based on actual transitions experienced since the node was created. This estimated transition probability (ETP) is both updated and stored by nodes. Nodes also store utility value estimates (using Q-learning) for the rules they represent based on reinforcements received during learning. Both these estimates can be used to assess whether structures should be retained or removed from the system (although in the basic system only the ETP is used) and both estimates are calculated using a recency weighted averaging process (described in Section 4.2.7).

4.2 System Operation

TRACA's basic system operates in discrete time within a major cycle. A new cycle begins each time an input string is received and detectors are *matched* based on the value of their corresponding bit in the string. Detectors then send messages up the hierarchy to groups which allow the groups to determine if their conditions (subordinates) are matched by the bit values in the current input string. Once all groups are matched, nodes within these groups which have had sufficient trials and are not support suppressed are eligible to *fire* and send support for their associated effector (the conditions for eligibility are described in more detail in Section 4.2.9). The system will then use this support to select an effector. Nodes in matched groups whose effectors were selected then *execute*, preparing to update their ETP and utility value estimate based on whether their predictions are matched with the next input string.

This match-fire-execute major cycle is presented in Figure 4.2 and its operation is demonstrated in the next section along with the process of rule creation.

1. **Match:** an input string is presented to the system with bits set to 1 matching respective detectors. Messages are passed through the network to match groups. Nodes are matched when their containing group is matched.
2. **Fire:** eligible matched nodes send support to their associated effectors.
3. **Effector selection:** the system selects an effector based on support sent by nodes.
4. **Execute:** matched nodes whose effector was selected prepare to update their ETP and value estimate based on the detectors matched in the next match cycle. Groups matched in the next cycle send any returns to nodes which executed in the previous major cycle which predicted them.
5. **Repeat.**

Figure 4.2: TRACA's Match-Fire-Execute major cycle

4.2.1 A Simple Maze

A trivial maze navigation example is used to demonstrate the building of the network and its operation. In the example, input strings are used to represent the system state as TRACA moves through the maze.

The maze consists of an aperiodic grid with four states (positions) arranged from left to right. The states are identified (from left to right) by the input strings "100", "010", "110", "001" (see Figure 4.3). The agent is initially placed in a randomly selected state other than the state identified as "001". The agent has two effectors, effector EA and effector EB. In any state the learning agent can select one of these effectors, selecting effector EA corresponds to the action Move-East and selecting effector EB the action Move-West. Selecting Move-East in any state will move the agent one position to the right, selecting Move-West will move the agent one position to the left unless the state is "100", in which case the agent's position will remain unchanged. Each action incurs a reward of -0.1 except when the state "001" is reached when a reward of 1.0 is provided. Receiving the positive reward constitutes the end of the trial. Once the trial is complete the agent is removed from the maze, its internal state is reset sufficiently so that trials are independent of each other and it is placed in another randomly selected position for the next trial. The task is to learn to follow the optimal path (based on maximising returns) for all possible initial positions. The system has no knowledge of the meaning of the input strings or of the effects of actions prior to learning, it bases its learning on the transitions between states that it experiences and any rewards received. For simplicity, the following discussion focuses on the building of the network and the operation of the major cycle, omitting the use of rewards and calculation of utilities until Section 4.2.7.

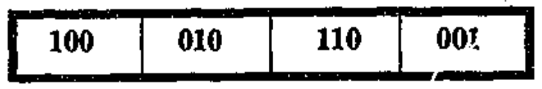


Figure 4.3: The sample maze.

4.2.2 Initial groups created for the sample problem

TRACA initially has one unary group for each detector and zero predictor nodes. For the purposes of the example we assume that the initial position of the agent in this first trial is the left-most state of "100". Network construction then starts with this initial input which matches detector DA and in turn unary group GA, an effector action is chosen at random, following which DA and GA are reset to unmatched. In this case, the selection of effector EA when GA is matched moves the system to the state "010" and the resulting input string matches detector node DB and unary group GB. Since there is initially no node in the system to predict the matching of GB, given the previous matching of group GA and selection of effector EA, a new predictor node, GA1, is created. GA1 will now predict the matching of group GB the next time detector DA is matched and effector EA is selected.

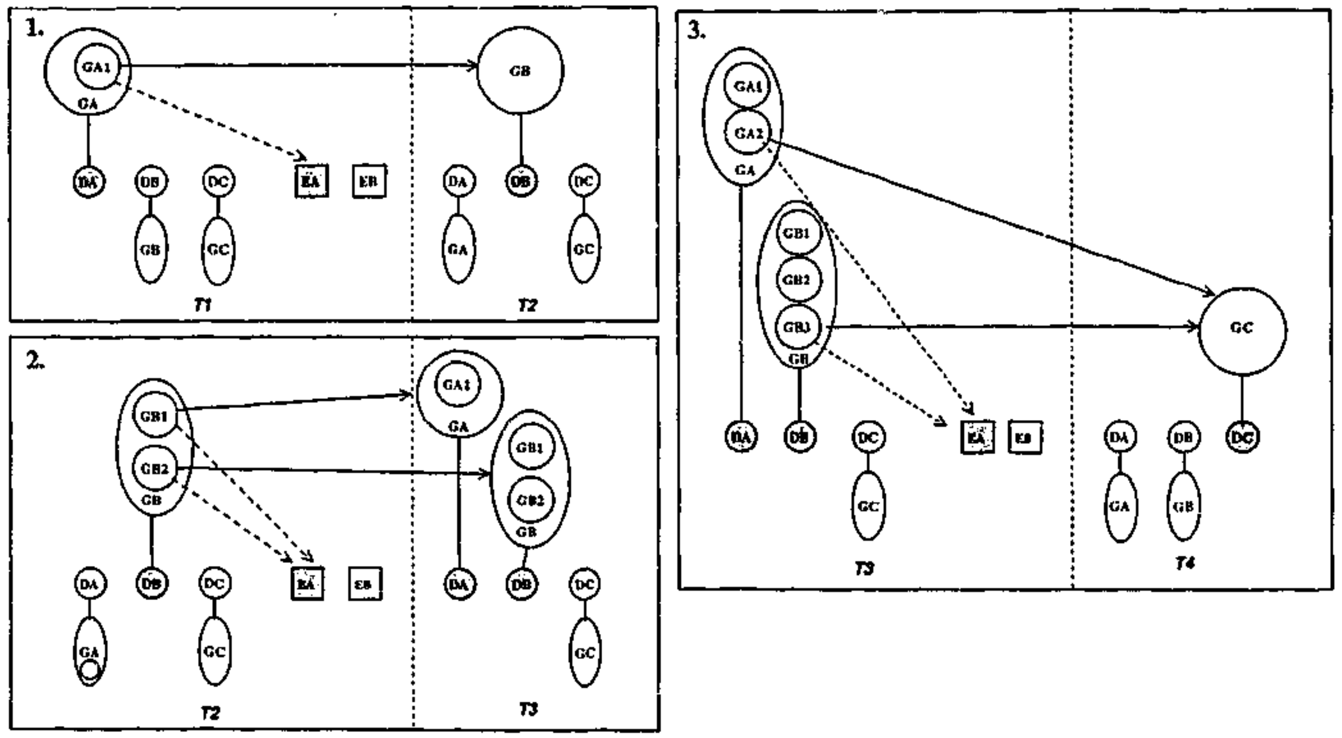


Figure 4.4: Sequence of predictor nodes forming for the maze problem. In each sub-figure, shading indicates detectors matched in one cycle (left), the selected effector (center), and the detectors matched with the following input string (right).

This first cycle is shown in Figure 4.4(1). In each of the cycles depicted in Figure 4.4, matched detectors and selected effectors are highlighted as darker. Matched groups are also shown as expanded to reveal the contained nodes. Each node whose effector was selected and whose prediction is *correct* (a node is correct when its predicted group is matched in the cycle following the node's execution) has a broken line to indicate its supported effector and a solid line to indicate its prediction.

In the second cycle (see Figure 4.4(2)) EA is again randomly selected. Following this the input string "110" is received resulting in the matching of both GA and GB. Since GB has no nodes to predict either GA or GB given the selection of EA, two new nodes are also created in GB. One to predict GA (GB1 predicts this) and one to predict GB (GB2 predicts this). Both these will execute with the next occurrence of GB being matched and EA being selected.

In the third cycle (see Figure 4.4(3)) we assume EA is once more selected randomly. As a result of this, the agent moves east, nodes GA1, GB1 and GB2 all execute and the input string "001" is received. After GA1, GB1 and GB2 executed, all three nodes' predictions were incorrect, as the groups they predict are not matched with the new input string. However, GC is matched, and GA and GB each create a new node (GA2 and GB3) to predict GC next time they are matched and the effector EA is selected. At this point the trial is completed and the agent is removed from the maze.

The groups created so far are insufficient to correctly represent and navigate the maze. If the group GA is now matched and effector EA is selected, one of the nodes GA1 or GA2 will make an incorrect prediction depending on whether the current state is "100" or "110". A similar problem exists for nodes in the group GB. The following section describes how these groups are used as components in the creation of a join group which influences them and in doing so correctly represents the problem domain.

4.2.3 Join Groups

If learning trials continue, at some stage the input string "110" will be received again and the groups GA and GB will both be matched in the same cycle. If the effector EA is selected while the agent is in this state, the nodes GA2 and GB3 will now correctly predict the subsequent matching of group GC (all the other nodes in each of these groups which support effector EA will make incorrect predictions).

When a group (such as GC) is correctly predicted by nodes in multiple other groups (but is not *always* correctly predicted by these nodes), it randomly selects and combines two of the predicting groups (which are not support suppressed) to create a new join group representing an AND combination.[1] In this case, a new join group, GAB, is created which contains a node, GAB1, to predict GC given the matching of both GA and GB and the selection of

---

[1] Whether or not a group creates a new join can be controlled by determining if the dependent ETPs of its current predictor nodes are higher than some minimum threshold value.

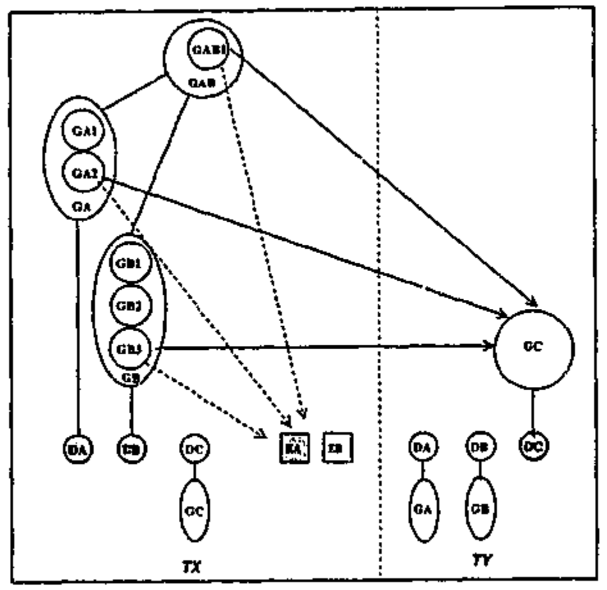


Figure 4.5: Use of a join group to create an AND construct.

effector EA (see Figure 4.5).[2] This join group implements a logical AND construct and its node GAB1 can be interpreted as representing the rule:

> if detector DA is matched and detector DB is matched and effector EA is selected
> then detector DC will be matched.

Join nodes construct a hierarchy by combining other nodes. In our example, group GAB can be seen as *superior* to groups GA and GB which are subordinate *inputs* into GAB (Dawkins 1976). GAB will only be matched when GA and GB are both matched in the same cycle.

4.2.4 Hierarchies and Chaining

As mentioned in Section 4.1.4, nodes together with their containing groups and their predicted groups, implement SRS rules which chain together hierarchies of groups. This type of chaining can occur between groups at any level of the hierarchy allowing representations such as the one depicted in Figure 4.6 in which nodes in groups at the top of a hierarchy can predict groups at the top of other hierarchies.

4.2.5 Representing NOT

In our example maze, the problem of having nodes GA1 and GB1 incorrectly predict GB and GA respectively when in state "110" is overcome by the group GAB sending a *support suppression* message when it is matched. The support suppression message prevents nodes in the subordinate groups from sending support and updating values when executing and in our

---

[2] Predicted join groups do not create joins until they are unsuspended (see Section 4.2.2).

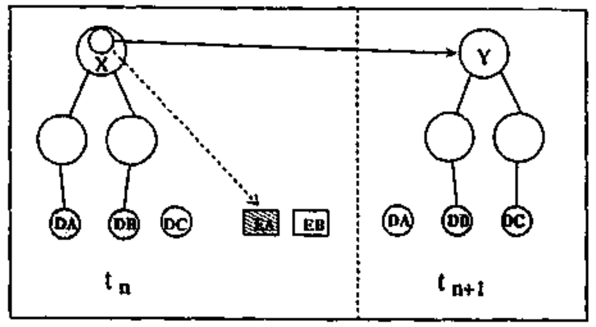


Figure 4.6: Direct chaining of a node in group X activated by environmental input (at t_n) to group Y activated by the subsequent input (at t_{n+1}).

case, can prevent the node GA1 from making an incorrect prediction. In our example, now whenever GA1 executes without being suppressed it will be correct (its prediction will be matched in the next cycle). The join group GAB, in conjunction with the support suppression message, now represents a logical NOT as nodes in each of the groups GA and GB will only send support to their effector if the superior group is not matched, and the superior group is only matched when both subordinate groups are matched.

In general, for the suppression mechanism to suppress a group, GA for example, there must exist another group, GB which is matched when GA is matched and can be used to create a join, GAB of GA and GB. When GB is matched in conjunction with GA the nodes in GAB represent a logical AND of GA and GB. When GA is matched and GB is not matched, the nodes in GA logically represent NOT GB. Note that without GA, it is impossible for TRACA to represent NOT GB. The presence of a group such as GA can be guaranteed in problems where such a group is required by providing one detector position which is matched every cycle (i.e an input bit position that always contains a 1).

In our example, with the addition of group GAB, node GA2 will still make incorrect predictions if the input string "100" is received and effector EA is selected. However, node probability estimates are updated using recency weighting (see Section 4.2.7 for details) and, since suppression prevents GA2 ever making a correct prediction, its dependent ETP will eventually reflect this fact[3]. Similarly, suppression and dependent ETP updates over subsequent trials will soon result in GA1's dependent ETP approximating 1.0.

4.2.6 Efficiency of Search and Representation

TRACA's combination of support suppression with join groups avoids the need for groups which explicitly negate one input. This in turn reduces both the number of possible equivalent structures and the search space accordingly. The effect of this is to bias the system away from complex structures based on negatives towards equivalent simpler structures using

---

[3] At this point the node could possibly be removed.

joins. When combined with appropriate sensory representations, this method of representing NOT provides an efficient means of excluding irrelevant inputs from the conditions of rules.

To illustrate this, consider an agent with two feature detectors, each of which includes two bit positions from the input string. Assume for now that it is only possible for a single bit in each feature detector to contain a 1 at any given time. Therefore, each detector is capable of assuming only 3 different values: "00", "01" and "10".

Systems such as Drescher (1991)'s Schema mechanism and Holland (1975) style Learning Classifier systems represent possible input strings by having a rule whose condition includes every bit position in the input vector (although in classifier systems any bit values can be matched using the "don't care" symbol). This type of rule is illustrated in Figure 4.7. On the other hand, TRACA's rules are specifically intended to only explicitly include a selected subset of the input vector's bit positions in each rule's condition.

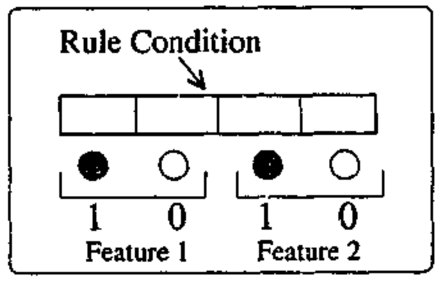
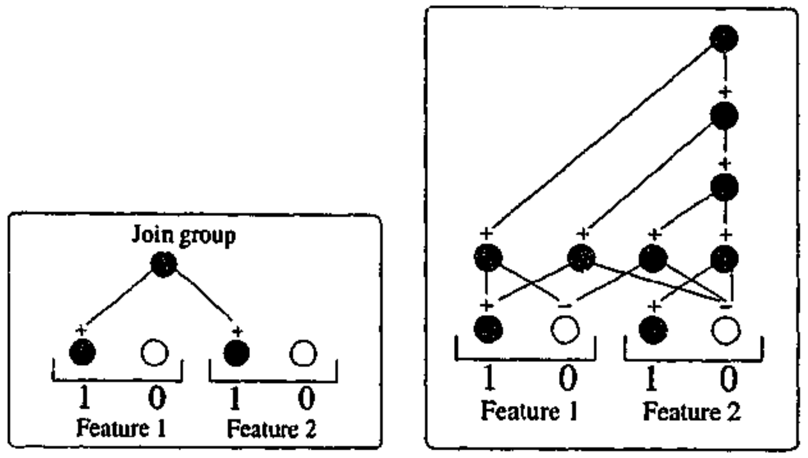


Figure 4.7: Many systems are constructed using rules whose conditions include every bit of input strings received from the environment.

Consider now a scenario in which TRACA allows explicitly negated inputs. Explicit negation means that each bit position using a join structure may be included in rule conditions either positively or negatively. If included negatively, a value of 0 in the bit position in a timestep allows the join to be matched (depending on the polarity of the join group's other subordinate and the current value in the corresponding position of the input string), while a value of 1 ensures the group cannot be matched in that timestep. The input pattern used in our example scenario is one in which the first bit position of each feature detector contains a 1.

Using explicit negation, one of the smallest possible structures would be to simply positively include both these bit positions. This structure is illustrated in Figure 4.8(a) which shows a possible join. The polarity of the subordinate group is indicated with either a positive or negative sign and the single matching input string (given our restrictions on possible input strings) is depicted below. In the figure, matched groups are indicated by a filled circle and unmatched groups by a hollow circle. The join illustrated in this figure is similar to the type of structure TRACA creates without explicit negation. However, with the ability to do explicit negation there are many possible equivalent structures which are far more complex. One such structure is illustrated in Figure 4.8(b). The problem of complex structures when using explicit negation is exacerbated as the number of join structures in the system increases



(a) A simple hierarchy to represent the input string "1010".

(b) A complex hierarchy to represent the input string "1010".

Figure 4.8: Possible hierarchies of joins when using explicit negation.

as all of these can also be positively and negatively included in joins. The resulting large number of equivalent structures both increases the size of the system's representation and makes search for new useful structures more difficult.

By not having explicit negation TRACA reduces the problem of equivalent structures. However, there are two remaining problems: (i) the need for additional joins to improve predictions; and (ii) representing the absence of a feature. The first problem arises if we allow multiple bits in each feature detector to contain a 1 in the same timestep. In these cases, TRACA requires additional joins to send a support suppression message to unary groups; otherwise unary groups may send support in inappropriate situations because they ignore the information provided by the other bit positions representing the feature.

The need for such suppressing joins can be reduced by increasing the number of bits for each feature so that each possible value of the feature is represented using a single unique bit. Since TRACA ignores bit positions containing 0 when creating joins, this increase in the number of inputs does not cause a corresponding increase in either the search or internal state spaces (as it may with a system such as Drescher (1991)'s Schema mechanism or with Holland style Classifier systems). The second problem, of representing the absence of a feature, can be resolved by including an additional bit which contains 1 in the absence of the feature.

A final mechanism contributing to TRACA's efficiency of search is the use of a *create suppression* message. When superior groups are matched, they send a message to their subordinate groups setting their state to *create suppressed.* Joins which are create suppressed are not eligible to be used as immediate subordinates in any new joins which may be created by the groups predicted during a cycle in which they were create suppressed. This reduces the search space by removing groups from the search for new structures during cycles in which they are create suppressed. It means that the nodes in newly created groups represent rules which are specialisations of the existing generalised or specialised rules in their containing group's subordinate groups. This mechanism effectively allows existing rules to be selectively used as components for new rules.

These aspects of TRACA's design allow it to provide efficient search and representation on many problems in addition to the efficiencies provided by default hierarchies. This efficiency is demonstrated by the structures created and illustrated for the problems presented in Chapter 5.

4.2.7 Calculating ETP and Utility Estimates

Omitted from the discussion so far are details of the calculation of utility estimates and estimated transition probabilities (ETPs). From now on, unless stated otherwise, we will only concern ourselves with the *independent* ETP which is updated (independently of support suppression) in each cycle the node executes. The utility estimate is updated for each cycle the node executes and is not support suppressed by a superior group (the utility estimate is therefore a *dependent* value).

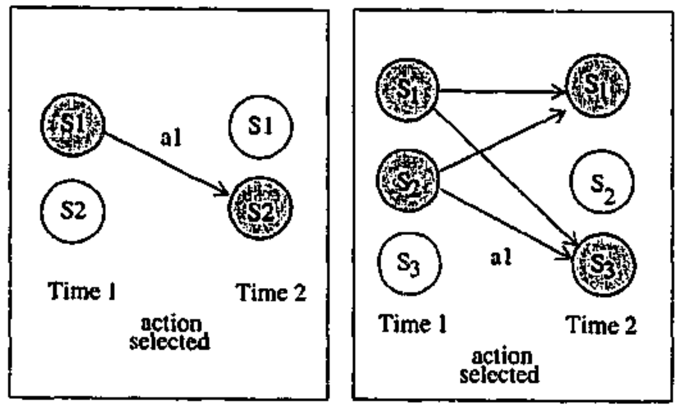
ETPs (e) are calculated using Equation 4.1. Here r will be 1 if a node's predicted group is matched in the cycle after it executed, and 0 otherwise. The learning rate, α, is a small constant. Initially e is set to zero.

$$e_{k+1} \leftarrow e_k + \alpha\,[r - e_k] \tag{4.1}$$

Note that ETPs are recency-weighted, this should allow fast adaption in non-stationary environments. Drescher (1991)'s marginal attribution machinery also uses recency weighted statistics in a similar manner.

The next problem for TRACA is determining the utility value returned from successor states. Utility estimates in TRACA (which are Q-values) are stored in nodes (which represent transitions to other groups) which in turn are contained in groups (which represent world states). Each group maintains at least one node to predict each group that has occurred as a successor (i.e to predict groups matched in the following cycle) since the group was created. If localised sensors are being used, where each state (or observation) is identified with a single unique bit of the agent's input string, successor groups can be identified unambiguously (as shown in Figure 4.9(a)) and the return is simply the highest node value within the group. However, if distributed sensors are being used, the successor state may be represented as multiple bits in the input string (as in Figure 4.9(b)), which may map to multiple groups (since in TRACA input bit positions are initially fully differentiated as unary groups).

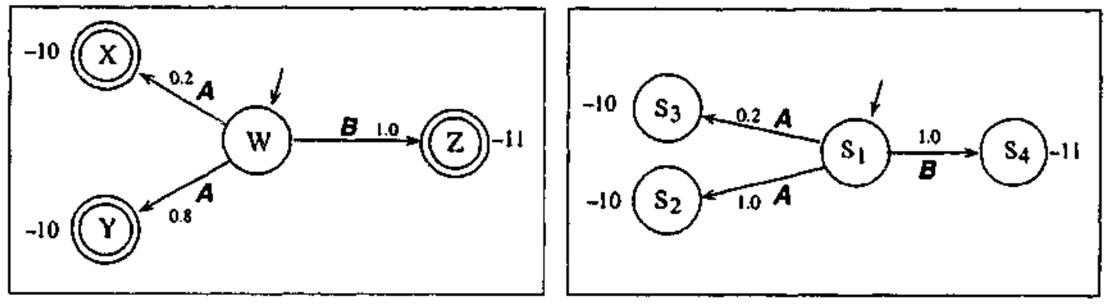
The problem now is: how to update utility value estimates in models with generalised internal states? One possible choice is to do value-iteration using the sum of each successor group's value along with the transition probability to the group. This was the approach used by McCallum (1995). An example illustrating this approach is based on the finite state automata presented in Figure 4.10(a). There are four states including three terminal states (X, Y and Z). Reaching X or Y incurs a penalty of -10 while reaching Z incurs a penalty of -11. The initial state is W and in each state the agent has two possible actions A and B (and must select one, there is no "stay" action). When action A is taken in state W, state X follows with probability 0.2 while state Y follows with probability 0.8. Without discounting, the value of action A in state W is calculated as the sum of 0.8 × -10 and 0.2 × -10, giving -10. In this scenario, action A is preferable to action B.



(a) Successive groups matched when using localised sensors (allowing only an enumerative internal model).

(b) Successive groups matched when using distributed sensors (allowing a generalised internal model).

Figure 4.9: Comparison of internal models for Q-learning.



(a) A sample finite-state automata.

(b) Transitions to a set of distributed sensor inputs for the problem in Figure 4.10(a).

Figure 4.10: A sample problem for Q-value updates.

Figure 4.10(a) demonstrates the situation using a localised sensor model in which each possible combination of features in the environment is represented by one unique bit in the agent's input string. Now consider the same finite state automata, but represented using distributed sensors. In this case, inputs are represented using a vector of state variables, $S = (s_1, s_2, s_3, s_4)$ each of which takes on value of 1 if the feature represented by the state variable is present in the state and 0 otherwise. In our new mapping of states to sensors, state W has a single feature which results in a value of 1 for sensor s_1 and state Z has the single feature represented by s_4. Both states X and Y have the feature represented by s_2 but X also has the feature represented by s_3. This use of distributed sensors produces the effect illustrated in Figure 4.10(b). Action A given sensor s_1 leads to sensor s_2 with probability 1.0, and to sensor s_3 with probability 0.2. Taking the sum of s_2 and s_3, 1.0×-10 and 0.2×-10 incorrectly gives action A the value of -12, making it seem a less desirable action than B with a value of -11.

This is just one problem which arises when distributed sensors are being used. It can be eliminated in a variety of ways through the creation of joins, however, until the state distinctions are made the value associated with action A in state W (i.e s_1) will be incorrect.

As a consequence of this, TRACA uses Q-learning to update the values of nodes rather than value-iteration (Watkins and Dayan 1992). Groups which are unsuspended and not *support suppressed* send returns to predicting nodes in groups which were not *support suppressed* in the previous timestep. The nodes that receive returns then update their Q-values using the maximum value received. If no predicted nodes are eligible to send a return, the return defaults to 0. The resulting Q-values will be approximations of the real Q-value and may vary between nodes in the same group and supporting the same action. This is unavoidable while developing specialisations and also while learning in environments with hidden-state. Similar approximations of Q-values are also used by Chrisman (1992) and McCallum (1993) for belief states and by McCallum (1995), Ring (1994) and Lin (1993) for other internal representations. The use of approximations for learning Q-values in belief spaces is also discussed in Littman, Cassandra, and Kaelbling (1995).

The calculated Q-value (a dependent value) is the node's utility estimate and is sent as support for effectors. The ETP (an independent value) is used to determine when join groups should be removed (join removal is discussed in Section 4.2.10).

Following is the Q-learning rule for non-deterministic environments (Mitchell 1997):

$$\mathcal{Q}_n(x,a) \leftarrow (1-\alpha_n)\mathcal{Q}_{n-1}(x,a) + \alpha_n \cdot (r_n + \gamma \cdot \max_{k \in A} \mathcal{Q}_{n-1}(y,k) - \mathcal{Q}_{n-1}(x,a)) \qquad (4.2)$$

where

$$\alpha_n = 1/(1 + visits_n(x,a)) \qquad (4.3)$$

Where $visits_n(x,a)$ is the number of times this state/action pair has been visited up to and including the nth iteration. However, the convergence of this rule (due to α_n) may slow adaption in non-stationary environments. This is also a problem in TRACA because general rules will be incorrect sometimes until other rules are developed which suppress it when the specialisations apply. Once the correct specialisation groups are created, the values of nodes in the subordinate general group will rise. From the perspective of this subordinate group, the reward function has changed. To quickly reflect this change TRACA uses a constant update rate in place of $(1 - \alpha_n)$ and α_n. By using a constant rate, node utility estimates will reflect changes, but may not converge. Since in practice Q-learning often succeeds with far fewer trials than theoretically required to converge this should not prevent successful learning (Littman, Cassandra, and Kaelbling 1995). The experimental results support this claim for its use in TRACA (see Chapter 5 and Chapter 7). A constant update rate is also used for ETPs. The effect of this is discussed in Section 4.2.10.

One Q-learning rule used in experiments with TRACA is the one presented in Equation 4.4 (Watkins and Dayan 1992):

$$Q_{(n+1)}(x,a) = Q_n(x,a) + b(r_n + c * max_k Q_n(y_n,k) - Q_n(x,a)) \qquad (4.4)$$

In TRACA, these Q-values are only updated for nodes which executed and were not support suppressed in the same cycle. However, it remains true that using this rule Q-values may oscillate in non-deterministic environments or for generalised internal states. In experiments this rule is refered to as the *standard learning rule*.

Alternatives for calculating Q-values

For non-deterministic environments there are alternatives to the use of a constant update rate for Q-learning.

One possible alternative is to update Q-value estimates using equations 4.2 and 4.3. In this case, to allow node Q-values to better reflect the effects of additional unsuspended superior groups, the value for $visits_n(x,a)$ used in equation 4.3 can be reset to 0 for nodes in affected parts of the network. However, it has not been necessary to use this alternative in any of the experiments with TRACA so far.

A second alternative takes advantage of TRACA's SRS rules and the associated relationship between nodes and their predictions. In this case each node includes the probability of their prediction occuring given they executed (without being support suppressed in the same cycle) and their prediction was matched in the subsequent cycle (also without being support suppressed):

$$Q(x,a,y) \leftarrow R(x,a) + \gamma Pr(y|x,a)\mathcal{U}(y) \qquad (4.5)$$

Where $\mathcal{Q}(x,a,y)$ is a node value, $R(x,a)$ is the immediate reward for executing action a when group (internal state) x is matched and $Pr(y|x,a)$ is the probability of group y being matched at t_{n+1} given that x was matched at t_n and action a was taken. $\mathcal{U}(y)$ is the utility of the group y. Under all learning rules, if a node predicts its containing group, its return is always zero.[4]

The rule in Equation 4.5 is refered to as the *SRS learning rule* and is used with TRACA in the truck driving experiment in Section 7.8. In the implementation of this, $R(x,a)$ is stored in nodes using a recency weighted average (using the learning rate as the constant update rate). $\mathcal{U}(y)$ is the maximum value of nodes in the group y. The rule is only applied to nodes which execute in a cycle during which their containing group (x) is not support suppressed. In place of a probability estimate for $Pr(y|x,a)$, the return is zero when a predicted group is not matched or is support suppressed in a cycle following the execution of the predicting node.[5] The resulting average is equivalent to keeping and applying explicit transition frequency counts.

4.2.8 Allowing Sufficient Trials for Returns

A problem arises when calculating returns from groups if negative rewards are being used. The problem arises if TRACA makes updates based on Q-values in new nodes before they have had sufficient trials to converge. It occurs with negative values because under Q-learning subsequent states (which correspond to TRACA's groups) return their maximum action value.

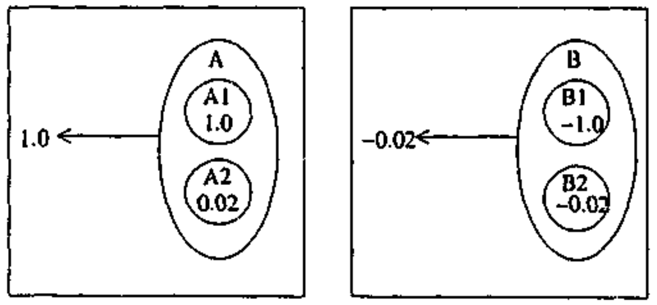


Figure 4.11: Two groups each containing two nodes. The numbers in the nodes are their utility estimates, the numbers at the end of the arrows are the returns sent by each group to predicting groups.

Consider the two groups in Figure 4.11. Group A has a node A1 which has had a number of trials and has a value of 1.0. Recently a new node A2 has been created and because it has had only a few trials, and utility values are initially 0, its current value is only 0.02. With sufficient further trials, the value of A2 will eventually approximate 1.0. However, in this scenario the incorrect estimate of A2 does not adversely affect the value of predicting nodes, as the return for the group remains at 1 because of the value of A1.

Unfortunately, a similar scenario using negative values does adversely affect the value of predicting nodes. In Figure 4.11 the group B has a node B1 with the value -1.0. A new node is created which has received only a few trials and its current value is -0.02. Given sufficient trials B2 will approximate -1.0, however, in the mean time its value will be used by Q-learning as the return from group B, because it is the maximum in the group (it is higher than the value of B1). This incorrect return leads to incorrect values in predicting nodes and may result in poor actions being selected. To combat this problem, all nodes are prevented from sending returns until they have a minimum number of trials.[6] In the experiments in subsequent chapters the minimum number of trials required before a node can send returns is equal to the number of cases required for the *test for noise* which is described in Section 4.2.11).

However, having a fixed number of minimum trials is not an ideal solution, as the number of trials required by nodes to approximate their true utility and ETP values may depend on a variety of factors, including the learning rate. Allowing a sufficient number of trials is particularly important when comparing nodes' ETP values to decide if a join group should be removed.

4.2.9 Allowing Sufficient Trials for Comparisons

A superior group will only be retained in the system if one or more of its contained nodes achieves a higher ETP than its two equivalent subordinates. Like utility estimates, ETPs are initially zero and must gradually converge to an (approximate) asymptotic or baseline value, called its steady-state. This is not an issue for unary nodes as they will have sufficient trials (at least as many as its superior nodes) before any nodes in join groups can be compared to them (however, at least one of the nodes in a unary group must have had a minimum number of trials before the group is set to unsuspended, currently the minimum number of trials required is equal to the number of cases required for the *test for noise* which is described in Section 4.2.11). To deal with this problem for nodes in join groups, the Cox-Stuart test for trend is used to test for the presence of a rising trend in the ETP of nodes. The Cox-Stuart test for trend is a simple statistical test which compares later observations with earlier observations. If a later observation is higher than the compared earlier one, this difference is recorded with a plus sign, if it is lower it is recorded with a minus sign. The frequency of these relative signs over a period of time indicates either an upward or downward trend. The probability of the occurrence of different numbers of plus or minus signs allow this test to be used for varying levels of statistical significance (Daniel 1990).

---

[4] This is used in place of the dependent ETP.

[5] Actions which lead to the same state will have the same value as actions which lead from the state to other states, plus any immediate rewards (or penalties) associated with the transition. An exception to this is where there is hidden-state in which case the construction of a temporal chain is required.

[6] When sending support for effectors the number of trials is not as important as the lowest negative value in each group is sent. See Section 4.2.13 for details.

Because ETPs start at zero, they will initially rise as nodes receive trials. Once the rising trend stops (or if the node's ETP never drops over the period required to collect sufficient observations to apply the test), the node is assumed to have reached a steady-state. Once a node reaches its steady-state, its group may be set to unsuspended and its nodes can send support for effectors. Nodes in join groups which reach their steady-state can only set their group to unsuspended if:

- the node reached its steady-state without it or any other node in the group having achieved a higher value than their equivalent subordinates (this allows the group to be removed); OR
- the node has at some point had a value higher than both its equivalent subordinates and it has had enough trials since it was first higher to determine whether it improves over these subordinates.

The problem that remains now is how to determine if a node improves over its subordinates. If there is no improvement by any nodes in a group the entire group should be removed, otherwise, the group should be retained.

4.2.10 A Problem with Measuring Improvements

Once a group is unsuspended, if it does not contain at least one node which provides an improvement over its equivalent subordinate nodes the group should be removed. Improvements are measured by comparing the ETP's of nodes.[7] However, a simple comparison between the ETP's of superior and subordinate nodes is complicated by the combined effects of a constant update rate and random ordering of the training data.

Because α is constant in both the ETP and the value estimate updates it is possible that neither of these estimates will converge instead they may oscillate around their steady-state values. These oscillations may make a subordinate node's ETP value occasionally rise above that of its equivalent superior nodes, even though the ETP value of the superior node is generally higher. This effect may occur when a rule is too general and also in noisy or stochastic environments. The effect is crudely depicted in Figure 4.12.

4.2.11 Dealing with Oscillating Values

The test used to tackle the problem of oscillating values is based on the sign test (see Daniel (1990) for a description of this well-known test), and I refer to it as the *test for noise*. The test for noise addresses the need to detect whether the values in a superior node are in fact consistently lower than an immediate subordinate node, or whether one of the values is just

---

[7] The ETPs used in this comparison are *independent* ETPs, therefore they are subject to distortion as described in Section 3.5.1 and not affected by suppression to improve predictions as described in Section 3.3.3. However, these comparisons are based on the relative reliability of predictions rather than reliability per se.

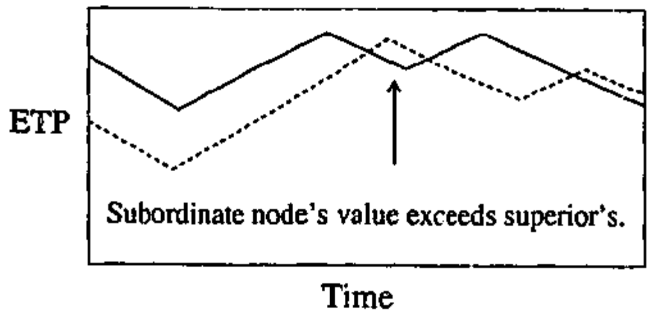


Figure 4.12: Depiction of oscillating values in a subordinate node and its equivalent superior (same effector and prediction).

on a downward or an upward oscillation. Using an average for this comparison does not help because the values of nodes change as learned structures are developed and start to drive system behaviour, and an average would be slow to reflect these changes (this effect is described in Section 5.4.1). An alternative is to use a recency weighted moving average, however, this fails because a relatively small run of events close together can quickly drive values up or down.

To deal with changes over time, it is necessary to look at events over a period of time which is sufficiently long to offset short trends and reflect longer term changes in values as learning progresses. The test for noise is used for this. It is essentially the non-parametric sign test with a fixed number of cases, however, as with the Cox-Stuart test for trend, new cases are continually added. With each new case added the oldest case stored is removed. However, unlike the test for trend, rather than comparing values at one time against values at a previous time, we compare values of a node in a superior group with the values of its equivalent nodes in its immediate subordinate groups. If the superior node is higher than both its equivalent subordinates (nodes with the same effector and prediction), it is recorded as a plus, if it is lower, it is recorded as a minus. The pluses and minuses indicate the frequency with which the superior node has a higher value than its best subordinate. Only if at least one node in a superior group is consistently higher than both its subordinate input nodes (as determined by this sign test) is a group set to *unsuspended* and retained in the system. Once retained, nodes in the group may send support to effectors. Only when all a group's nodes are consistently lower than their equivalent subordinates is the group removed.

The cases for the test for noise are collected by superior nodes as soon as the superior node's ETP estimate reaches a value higher than both its equivalent subordinates. Cases are added each time the node executes, and are continued to be collected and evaluated over the lifetime of the node (this allows groups to be removed as described in Section 4.2.10). This comparison between nodes and their subordinates can be slightly biased by increasing or decreasing the values of either by a small percentage, called an *improvement factor* (*IF*), before they are included in the test, therefore requiring a minimum improvement for superior nodes to be retained.

Like the Cox-Stuart test, the test for noise is extremely efficient to store and to calculate. For a given sample size and significance level, the detection of a trend reduces to counting the number of pluses or minuses and comparing them to the number required for significance. Alternative measures for significance used in similar systems are the Student t-Test, used by Chapman and Kaelbling (1991), and the Kolmogorov-Smirnov test (also a non-parametric test) used by McCallum (1995).[8]

4.2.12 Local Minima

Like any learning algorithm, TRACA is subject to local minima where it finds suboptimal solutions. Local minima in TRACA occur when a rule is discovered which is useful in some situations, but one or more better rules are possible. A better rule may be a single rule that is more generally applicable, or part of a default hierarchy, with a general rule and appropriate exceptions. However, in the absence of better rules, the local solution is often still useful. TRACA aims to exploit such locally optimal rules while continuing to search for a global minimum. However, once a better rule is found it is possible that both it and the locally useful rule may exist together in the system. This can lead to TRACA creating a larger rule set than some alternative approaches since TRACA does not currently attempt to specifically eliminate rules representing local minima (a possible solution to this is described in Section 8.4).

A contributing factor to the occurrence of local minima in TRACA is the *create suppression* mechanism. This mechanism is necessary to prevent duplicate join groups being formed and is also used to exclude subordinate groups from searches for new structures (this was discussed in Section 4.2.6). However, create suppression may have the undesirable side-effect of restricting the opportunities for the subordinates to participate in new joins.

To combat the possibility of create suppression contributing to local minima an *exclusion* mechanism has been developed. This mechanism allows joins to occasionally escape create suppression by its superior joins while ensuring that duplicate join groups are not created.

Take for example, a join group C with two subordinate groups, A and B. For a duplicate of C to be created, both A and B must have been matched in the same cycle, and not have been *create suppressed* by a superior group. So that A and B may be used in other joins which exclude the combination of both together, each is occasionally excluded from the create suppression imposed by C. However, only one of A and B can be excluded at any one time for this mechanism to work.

The exclusion mechanism is implemented by randomly selecting a small number of *create suppressed* groups to be excluded each cycle. A selected group sends a message up the hierarchy to its superiors requesting that it be excluded from the create suppression. Once all superior groups at the root of hierarchies the requesting group belongs to have received the

---

[8] Chapman and Kaelbling (1991) mention that they considered non-parametric tests inappropriate for their domain but do not provide any further explanation.

message they can approve or deny the request. Approval is implicit if no denial is sent and also temporarily excludes the root group from participating in new joins. A root group will only deny the request if another request has already been approved (received). Requesting groups that are excluded from the suppression may participate in new joins without the risk of a duplicate of an existing join being created.

4.2.13 Effector Selection

Effectors are selected based on their ***effective support*** which is calculated using the support sent by nodes. Nodes send as support their current utility value estimate. This support is sent by nodes which belong to matched groups which are unsuspended and not support suppressed. However, not all nodes in a group get to send support. In all cases, only nodes with the highest positive value and lowest negative value of all nodes in their group are allowed to send support.

In the experiments presented in later chapters, two methods are used to calculate an effector's effective support based on the support received by nodes. These are the *best supporter* and *average support* methods. The *best supporter* method calculates the effective support for each effector as the sum of the highest positive support and the lowest negative support received. The *average support* determines the effective support for each effector as the sum of the average positive support received and the average negative support received.

Ideally, the *best supporter* method of selecting effectors is the most desirable, since once learning is complete we would like to always select the best action. However, often actions must be selected before learning is complete. In TRACA, this means some rules (nodes) may be sending support in an inappropriate context because suitable suppressing superiors (specialisations) have not yet been created. In this case, the support for poor action choices may be of sufficient magnitude for an action to be selected inappropriately regardless of support sent by other better rules. Hopefully, given enough experience suitable superior join groups will be created and (using support suppression) prevent such adverse affects on system behaviour. However, in some cases, a complete rule set may not be formed, either because of insufficient training experience, or a problem which is difficult to represent. In such situations, an *average supporter* method of action selection may demonstrate better performance. In TRACA many rules (nodes) may be eligible to send support at any one time, using an average allows a voting mechanism which reduces the "voice" of individual rules. However, such a voting mechanism may have undesirable effects if the majority of rules sending support are inappropriate to the situation. This could occur in environments with hidden-state (such as those presented in Chapter 7).

Another issue related to effector selection is exploration. During learning it is often necessary to try a variety of actions to explore the environment. There are many possible schemes for such exploration, some of which were discussed in Section 2.14. The experiments presented in following chapters use two simple exploration schemes. The first is the *uniform method* which selects the effector with the highest support according to a probability p. With probability $1 - p$ it selects from all possible effectors with a uniform random probability. The second is the *non-uniform* method. This is a variation on the roulette-wheel approach described in Goldberg (1989). Goldberg's approach sums the support for all effectors and selects actions probabilistically based on the proportion each effector's support contributes to the sum. TRACA uses the roulette wheel only for exploratory actions. TRACA's non-uniform approach selects the effector with the highest support with probability p. However, with probability $1 - p$ an effector is selected using the roulette-wheel approach.

When selecting exploratory actions using the roulette-wheel, if any effector obtains positive effective support any other effectors with negative effective support have their support replaced with a small positive effective support. Effectors are then selected using the roulette wheel method according to their effective support. The roulette wheel allocates to each effector a probability of being selected directly proportional to their contribution to the sum of effective support across all effectors. If all effectors have negative effective support the probability of each being selected is inversely proportional to the absolute value of their effective support.

When using the roulette-wheel to select exploratory actions it is necessary to ensure that no actions are unduly excluded from frequent selection. Therefore, effectors which have a very low effective support ($-0.0001 < \textit{effective-support} < 0.0001$) may have their support replaced to improve their probability of selection. There are two schemes which are used for this. The first simply allocates a small constant value as the effective support (however, this value will be problem dependent). The second scheme allocates effective support calculated as the maximum positive support received divided by the number of effectors. If no effectors have positive support, the effective support is calculated as the minimum negative support received divided by the number of effectors. Unless otherwise stated, experiments in subsequent chapters use the second scheme. If no effectors have support, an effector is selected with uniform random probability.

Note, that every effector may receive both positive and negative support from a number of nodes at every step. The effective support of each effector can be seen as either an indicator of the desirability of selecting that effector or of the agent's confidence in its suggested action.

4.2.14 Hypothetical Look-ahead

A final feature of the basic system is its ability to perform hypothetical-lookahead. Hypothetical lookahead is not a new idea, and its implementation in TRACA closely follows the description in Holland (1990) for Learning Classifier Systems which was used in experiments by Riolo (1991). In TRACA, each node has a "virtual" strength variable, which is used to store Q-values propagated back by look-ahead. However, unlike Riolo (1991)'s implementation, TRACA does not require explicit creation of rules to form chains, as TRACA's rules are chained directly.

A look-ahead cycle begins with groups matched by the current input. Nodes belonging to matched groups which are unsuspended, propagate a message matching their predictions. Any unsuspended groups predicted as a result of this message pass back a *virtual return* based on its *virtual utility*. A group's *virtual utility* will be either its average reward (if non zero) or the maximum virtual return received by any nodes contained in the group (average rewards can be either based on actual rewards received upon transitions to the group in real trials or can be explicitly "injected" by an experimenter into appropriate groups, this is done in the experiment in Section 5.7). Hypothetical look-ahead can be repeated from the most recently hypothetically matched groups to an arbitrary depth in the internal model. This continues for a pre-determined number of virtual timesteps (i.e to a pre-defined distance) during which the virtual utility of nodes is passed back to the hypothetical predictor nodes of the sending group in the same manner actual values and rewards are passed. The look-ahead cycle can then be repeated as often as required to propagate the values of goal states back to the groups matched by the current real-world state.

It is impossible for TRACA to calculate the Q-value of a group without some real-world experience, so the additional dependent ETP variable is maintained for look-ahead. This variable estimates the transition rate for each node to its prediction. Like the independent ETP, it is calculated each time a node executes (in a non-look-ahead cycle). However, unlike the independent ETP, it is updated with 1 only if the node is not support suppressed when it executes and the node's predicted group is matched and not support suppressed in the following cycle, otherwise it is updated with 0. The hypothetical Q-value (virtual utility) for each node is the product of the maximum hypothetical return it receives during look-ahead cycles and the dependent ETP. The learning rate for hypothetical updates is 1.0.

Once all the required look-ahead cycles have completed, look-ahead terminates and action selection proceeds with actions being selected using the virtual utility of nodes matched by actual inputs. In general it is difficult to know to what depth look-ahead should extend. However, in the experiments with TRACA (in Section 5.7) a sufficient depth can be easily determined based on the size of the experimental grid world.

4.3 Diagrams of the Basic System

Unified Modelling Language (UML) diagrams of TRACA's basic system are presented below including a class diagram (Figure 4.13) and sequence diagram (Figure 4.14). Associations on the class diagram indicate the primary direction of messages between instances of classes. In the case of nodes, the association roles are shown to distinguish messages between nodes and their container group and nodes and their predicted group. Association roles are also shown for groups to indicate that the `sendsMessagesTo` association is directed at subordinate groups . The `sendMessagesTo` association encapsulates the two types of suppression messages (the *create suppress* and the *support suppress* messages).

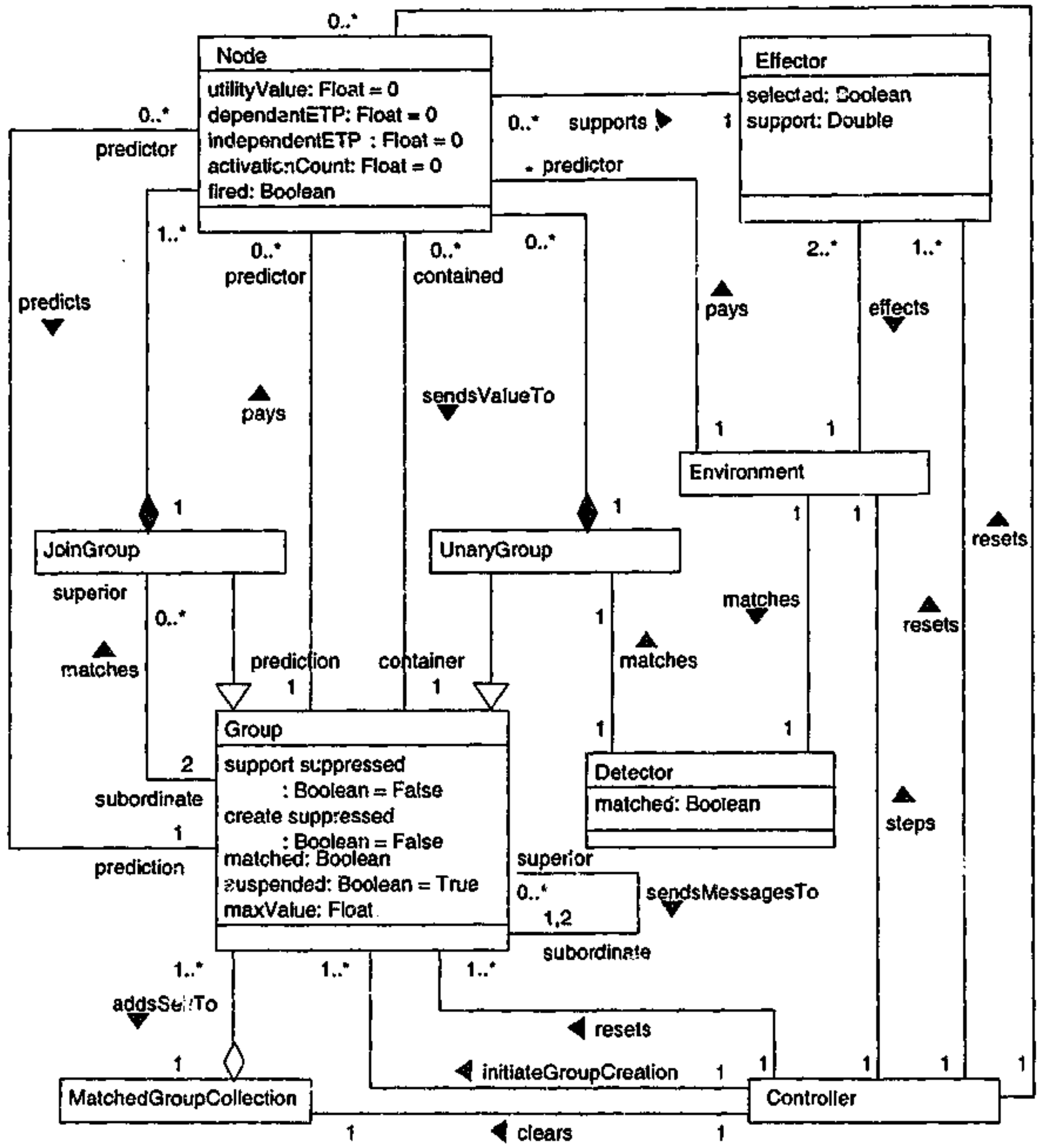
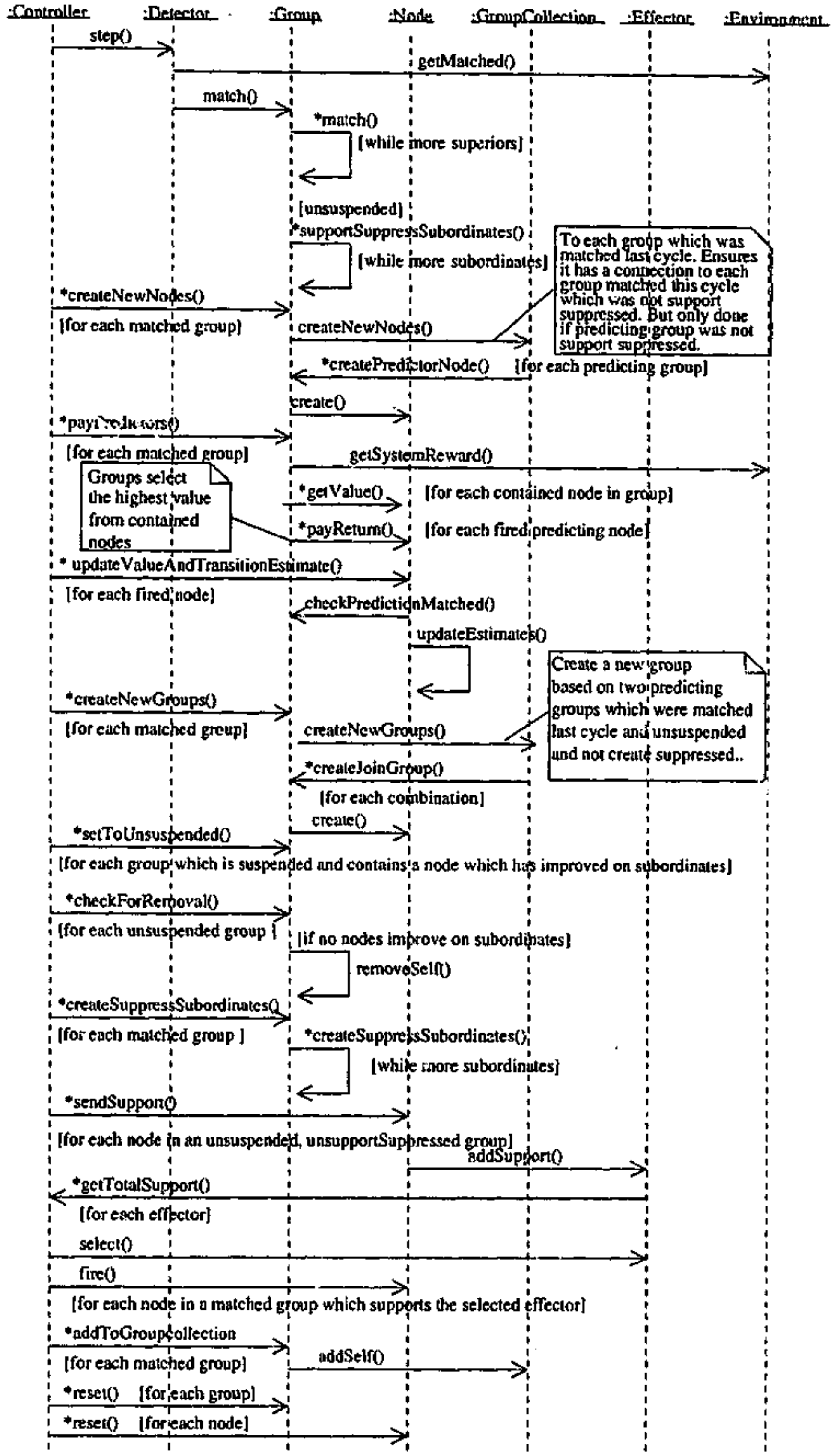


Figure 4.13: Class diagram showing class relationships for the basic system

In the sequence diagram (Figure 4.14) messages begin with detector nodes receiving their inputs from the environment at the start of the major cycle. The steps of the major cycle are timed and controlled by the `system controller`. Each major cycle corresponds to a single step in the environment. Detectors in turn send input messages on to the unary groups, which pass those messages up to their superior join groups. Nodes send their utility value estimates to their containing group so the group can send the highest value to predicting nodes if their predictions are correct. Nodes which make correct predictions also receive any instantaneous reinforcement received from the environment as a result of their actions. While the controller maintains a list of all nodes and groups, various other lists may also be maintained for efficiency. One of these lists is a list of groups matched in the previous cycle.

The sequence diagram shows the interactions between instances of the various classes. Message names and loop labels indicate the recipients of messages (which may apply to all



To each group which was matched last cycle. Ensures it has a connection to each group matched this cycle which was not support suppressed. But only done if predicting group was not support suppressed.

Groups select the highest value from contained nodes

Create a new group based on two predicting groups which were matched last cycle and unsuspended and not create suppressed..

Figure 4.14: Sequence diagram for the system's major cycle

members of various collections). The `groupCollection` is the set of all groups that were matched in the previous cycle. Following the sequence diagram is a textual description of significant steps within the major cycle. Some of these steps use stored (backed-up) values from the previous cycle. Following the sequence diagram is textual description of the major cycle.

4.4 Steps in the Basic System

The major steps of TRACA's basic system for input generalisation are presented below. Descriptive comments are included using *italics*, values of state variables are indicated as `fixed width` and verbs are indicated as **bold**. Skeleton pseudo-code for the major cycle of the basic system is provided in Appendix A.

Steps of the Basic System:

The system starts with a controller, a set of effectors, a set of detectors, and a set of unary groups. There is one unary group for each detector. Unary groups initially contain no nodes. As join groups are created, each is initially suspended until at least one of its nodes demonstrates an improvement over its two equivalent subordinate nodes (see Section 4.2.9).

The `support suppressed` *variable is used to both prevent nodes sending support for effectors and to prevent groups connecting or being connected to other groups. The* `create suppressed` *variable is used to prevent duplicate joins by preventing groups being used as subordinates for new joins. The* `create suppressed` *variable of subordinate groups can be set by either a suspended or unsuspended superior. The* `support suppressed` *variable can only be set by an unsuspended superior.*

1. An input string is received by the system. Detectors determine if they are `matched` based on the value in their bit-position.

   Detectors are matched only if they have a 1 in their corresponding input bit position.

2. Detectors notify their immediate superior groups whether they are `matched` or `unmatched`. Each notified group passes a notification to its superiors indicating whether it is `matched` or `unmatched` until all groups have received a notification. A join group is only `matched` if both its subordinate groups are `matched`, otherwise it passes on the notification that it is `unmatched`.

3. Groups which are `matched` in this cycle **pay** their predicting nodes a `return` based on the maximum returns associated with nodes in the group.

4. Nodes which executed in the previous cycle **update** reinforcement `utility estimates` and `transition estimates` (ETPs). Nodes in groups which were not `support suppressed` in the previous cycle update their utility estimates based on the immediate reward and the `returns` they received from their prediction.

   Groups which controlled behaviour in the previous cycle update their values based on the values of groups which will control behaviour this cycle (see Section 4.2.7).

5. Groups that are `matched` this cycle and are **unsuspended** send a message to their subordinates which sets them to `support suppressed`. This message is passed down the network by the subordinates until all subordinates are `support suppressed`.

   Support suppression prevents subordinate nodes sending support for effectors. This ensures only groups at the top of matched hierarchies control system behaviour.

6. Groups which are `matched` this cycle **create nodes** to predict them for any groups which were `matched` in the previous cycle and did not contain a node to predict them.

   This chains hierarchies together temporally.

7. Groups which are `matched` this cycle, are not `support suppressed` and have more than one predicting group which was not `create suppressed` in the previous cycle, is unsuspended and contains nodes which executed in the previous cycle, select two of these predicting groups. They **create** a new join group for the selected two groups which is a composite of the two and which contains a node predicting the `matched` group. The new node supports the effector selected in the previous cycle. (*Groups which have one or more predicting nodes which executed in the previous cycle whose ETP exceeds a minimum threshold skip this step if a threshold is set.*)

   Create new join groups by selecting two groups that controlled system behaviour in the previous cycle and creating a new join group with these as subordinates. Only groups which are controlling behaviour this cycle create new groups, so we are creating and connecting groups at the top of the respective hierarchies.

8. Suspended groups which contain a node which has had sufficient trials for its ETP to converge to its steady-state value set themselves to `unsuspended`.

   Unsuspended groups are given initial trials before they are evaluated for retention or removal from the system (see Section 4.2.9).

9. Groups which are **unsuspended** and do not contain at least one node which has demonstrated an improvement over its equivalent subordinates (nodes in the immediate subordinate groups with the same prediction and effector) **remove** themselves along with all their superiors.

   Groups which were being trialled may be removed once those trials are over if they have not demonstrated usefulness. Also groups whose contained nodes had demonstrated an improvement in the past may later be removed if its nodes do not continue to demonstrate an improvement. This allows for adaption to changes in the environment *and removal of groups incorrectly retained due to uncharacteristic runs of events (see Section 4.2.11).*

10. Groups which are `matched` this cycle send a message to their subordinate groups setting them to `create suppressed`. This is passed down the hierarchy until all the group's subordinates are `create suppressed`. Groups which were `create suppressed` in the previous cycle set themselves to not `create suppressed` prior to these messages being sent.

    Ensure groups which are **not** *at the top of controlling hierarchies in this cycle are not used as subordinates when creating new join groups in the next cycle.*

11. Nodes in *matched*, unsuspended groups which are not `support suppressed` **fire** sending `support` for their associated effectors.

    Nodes in groups at the top of matched hierarchies control system behaviour(see Section 4.2.13)

12. The system selects an effector based on the `support` sent by nodes this cycle.

13. Nodes in matched groups whose action was selected **execute**.

14. **Repeat**

4.5 Chapter Summary

This chapter has described the operation of TRACA's input generalisation algorithm. It also described how TRACA achieves efficiency in search by excluding irrelevant inputs and by using suppression in conjunction with node groups to avoid complex equivalent structures. Excluding some inputs from rule conditions also allows efficiency in representation and computation as statistics only need to be computed and stored by rules directly affected by the values of bit positions in the agent's input vector. These efficiency gains are achieved by requiring additional input bits in the input vector for some problems. As is demonstrated in the experiments in Chapter 5 and Section 7.8, node groups contribute further to efficient search and representation by allowing the reuse of created structure.

Chapter 5

Experiments using the Basic System

5.1 Introduction

This chapter examines TRACA's performance using the basic system, as described in Chapter 4, without temporal structures. The basic system is intended primarily for input generalisation and state discrimination. Input generalisation is similar to the traditional machine learning problems of classification and concept learning. The concept learning problems selected from the literature for experiments with TRACA allow comparison with a number of other systems.[1] In addition to this comparative analysis, these problems are also used in an empirical analysis to determine the effects of changing various parameters within TRACA. Classical concept learning tasks do not require sequences of actions to solve them, so this chapter also examines a robot navigation domain in which chains of rules can be developed to connect structures which discriminate states. As a final experiment, a navigation domain is used to demonstrate the potential benefits of hypothetical look-ahead planning.

5.2 Generalisation Tasks

This section analyses TRACA's performance on three input generalisation tasks. All three tasks have irrelevant attributes and one has noise present in the training data. These are supervised learning tasks which not only allow comparison of TRACA's input generalisation with other systems, but are also used in Section 5.4 to evaluate TRACA's behaviour under parameter changes. In selecting suitable tasks for these analyses, TRACA was trialled on a range of well known problems selected from the machine learning literature. The state spaces of these problems varied in size from several attributes (features) up to 60 attributes. Over this range of problems (and using a single set of parameters across them all) TRACA was

---

[1] Problems such as learning boolean multiplexers and parity problems where there is little opportunity for generalisation have been avoided in these comparisons as it has been suggested that they are inappropriate tests for the types of problems TRACA is intended for (Lin 1993; Fahlman 1988a).

able to achieve an accuracy within several percent of the best performing algorithms, often with less training. Interestingly, on many of these problems, only slightly lower predictive accuracy (and some cases, higher predictive accuracy) was achieved without the use of join groups (join creation was turned off) (Mitchell 2002).

Out of the range of problems trialled, among the most challenging was a set of three artificial tasks called the Monk problem. The internal structure required for these tasks is relatively clear and from the experiments in Mitchell (2002) it appears their solutions require more structure than many of the other standard classification and concept learning tasks in the literature. Furthermore, they include the task on which TRACA's predictive accuracy (and the accuracy of a number of other algorithms) was lowest.[2] Consequently, it is the Monk problem that has been used for the following empirical investigations into TRACA's performance. For results on the complete set of machine learning tasks trialled see Mitchell (2002).

5.2.1 The Monk's Problem

The Monk problem is an artificial problem incorporating three different tasks each with its own concept.[3] Each task is a binary classification task involving examples with 6 attributes taking the following values:

- Attribute 1 (A_1): values {1,2,3};
- Attribute 2 (A_2): values {1,2,3};
- Attribute 3 (A_3): values {1,2};
- Attribute 4 (A_4): values {1,2,3};
- Attribute 5 (A_5): values {1,2,3,4} and;
- Attribute 6 (A_6): values {1,2}.

Each example has a value for each attribute and is one state in the total possible space of 432 states. In the artificial domain of this problem each example represents either a friendly or unfriendly robot. For each of the three tasks, membership of the friendly class is determined by the following logical description:

- Monk 1: $A_1 = A_2$ OR $A_5 = 1$.
- Monk 2: $A_n = 1$ for exactly two choices of n.
- Monk 3: ($A_5 = 3$ AND $A_4 = 1$) OR ($A_5 \neq 4$ AND $A_2 \neq 3$).

The attributes indicate features of the robot. For example; head and body shape, whether the robot is carrying a sword and whether or not it is smiling. Each task involves presenting a subset of the possible examples to the learning agent along with (or followed by, in TRACA's case) the information on whether the example is an instance of a friendly or unfriendly robot. From its experience with this subset of the total possible examples, the agent must try to learn how to classify the remaining examples, without being provided explicit information on the class the example belongs to. The third task includes 5% misclassifications (noise) in the training examples.

The Monk 1 and 3 tasks are in a standard disjunctive normal form. However, the combination of attributes for Monk 2 is complicated to describe in disjunctive or conjunctive normal form (Thrun et al. 1991). This property makes learning Monk 2 difficult for TRACA as is explained in Section 5.3.3.

5.2.2 Experimental Methodology

These experiments are primarily concerned with the predictive accuracy of TRACA when compared to other similar systems. However, also of interest is the amount of training experience required to achieve the recorded accuracy and the amount of structure used to represent the solution.

Thrun et al. (1991) contains results obtained from the application of a wide range of learning systems on the Monk problem. Of these results only those obtained using incremental systems were used in comparisons with TRACA. These systems include ID5R, ID5R-hat, Back-propagation neural networks and Cascade correlation networks. In Thrun et al. (1991) all the systems were trained on a fixed set of training examples for each problem and in each case the test set was the full set of examples. Obtaining training and test data in this way is unusual and in general random selection and cross-validation methods are prefered (Cohen 1995). However, these results were obtained by researchers proficient with the various systems and so as to take advantage of their expertise their results are relied on as presented. The number of training examples in the fixed training sets for each task are (Thrun et al. 1991):

- Monk 1: 124.
- Monk 2: 169.
- Monk 3: 122.

For consistency the same test data and fixed training set is used in the experiments with TRACA. The training as provided is in ascending order based on the attribute values. To mitigate any effects of this ordering on the experiments with TRACA nine randomly ordered

---

[2] The reason the artificial Monk tasks are more challenging may possibly be explained by criticisms that machine learning practitioners use problems that are too easy. There have also been questions raised about whether these tasks are representative of common-real world problems (Holte 1993; Cohen 1995).

[3] The name is derived from the Corsendonk Priory Monks who invented the problem as a result of hosting the 2nd European Summer School on Machine Learning.

copies of each training set were created. During training on each task examples are drawn from the resulting ten training files in succession and the training cycle is repeated as necessary for the required number of epochs. All results obtained by TRACA are averages of 20 runs on the training data, and each run is started using a different random seed.

5.2.3 Validating the Default Training Set

To validate the results obtained by the experiments in following sections (see Section 5.2.7) which test TRACA using the default training set, an additional set of experiments was run in which 20 different training and test sets were generated for each task. Each of these training sets was created by randomly selecting a set of training examples from the full set of examples. The remaining examples are then used as the test set. The training sets generated for each task each had the same number of training examples as in the original fixed training set. This experimental design is similar to one used by Holte (1993) in his comparisons of learning algorithms. The results obtained using these validation experiments were compared to the results obtained using the fixed training sets which are presented in Section 5.2.7. The comparisons were aimed at revealing statistical differences at the 0.05 significance level. Some of the data did not pass the Wilk-Shapiro test for normality, so in those cases both the student t-Test the Mann-Whitney test were used in the comparisons (National Institute of Standards and Technology 2001). For Monk 1 there were no statistical differences between the results obtained by the validation experiment and the experiment using the fixed training set. The average accuracy on the test data (i.e the percentage of test examples correctly classified) for these two experiments were within 0.6 percent of each other. Similarly, there was no statistically significant difference between the predictive accuracies obtained for the two experiments on Monk 3. The average accuracies for the two experiments on Monk 3 were within 0.1 percent of each other. Only for Monk 2 did the t-Test indicate a statistically significant difference (at the 0.05) level between the results obtained by the two experiments, however, this was not reflected by the Mann-Whitney test which indicated no statistically significant difference. The average accuracy obtained for Monk 2 in the validation experiment was 64.8 percent, 5.2 percent lower than the average result obtained in the experiment using the fixed training set.

5.2.4 Input Representation

TRACA's input representation scheme (based on a fixed length bit vector and input strings) is designed for on-line learning environments consistent with the description of the system's operation in Section 4.2. However, the same representation scheme can be applied, somewhat unusually, to supervised learning tasks such as the Monk's tasks. In these tasks, the presentation and classification of each example is treated as a single trial in the agent's environment. Each trial consists of only two timesteps and the agent has as many possible actions as there are classes. The input string for the first timestep represents the features or attributes of the example to be classified. The agent then selects an action to indicate the class of the example represented by the first input string. The content of the second input string then indicates whether or not the agent's classification was correct and a corresponding reward or penalty is provided. At this point the trial ends and the next one may commence independent of the first apart from any internal changes within the agent as a result of learning.

Each example is represented in the first input string of trials using a binary sub-string for each attribute. Within a sub-string there is a bit position for each of the possible values of the feature represented by the sub-string. Following the method for representing features described in Section 4.2.6 (which is not the most compact, but is efficient) at most one bit of a sub-string will have a value of 1 at any time, all other bits are zero. For the Monk tasks, a value of 1 in the left most bit position indicates a value of 1 for the attribute while a 1 in the right most bit position indicates the highest possible value of the attribute. These strings are concatenated to create an input string 17 bit positions long. Two extra bit positions are then added to contain the information about the correct classification in the second timestep of each trial. Of these two, the leftmost position will contain 1 if TRACA's classification (action selection) was correct, otherwise, the rightmost position will contain 1. In the first timestep both the classification bits will contain zero while in the second time-step all the feature bits will contain zero. The resulting bit vector is shown in Figure 5.1 along with an example input string.

Since TRACA is a reinforcement learning system, without appropriate reinforcement it will develop structures to predict percepts but not attempt to make correct classifications. To direct TRACA's behaviour appropriately a reward of 100 is provided for correct classifications and a penalty of -100 for incorrect classifications.

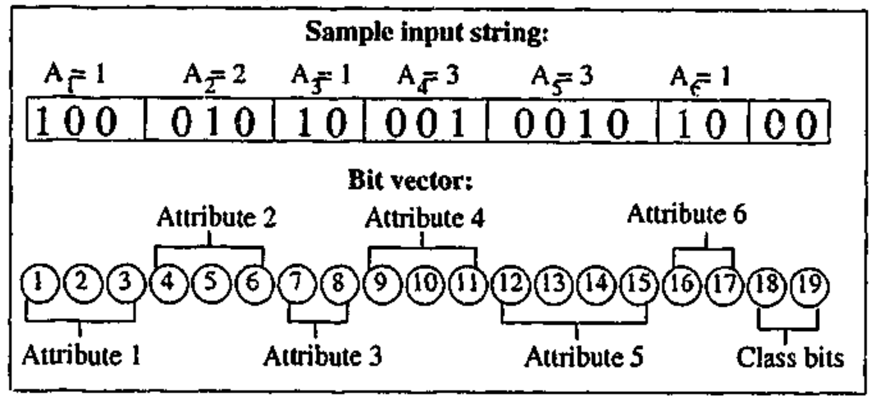


Figure 5.1: A sample input string and the bit vector format for the Monk problem.

5.2.5 Input Generalisation Parameter Settings

The number of cases used for the Cox-Stuart test for trend (see Section 4.2.9) and the test for noise (see Section 4.2.11) was 10 and 20 respectively with a two-tailed significance of 0.05. The exclusion rate for joins was 1 in each 20 times they were matched (see Section 4.2.12 for details on this). The learning rate was 0.1 for all three problems. An exploration rate of 0.25 was used (1 in 4 actions were selected randomly during training). A constant value of 0.5 was assigned to effectors which received support within the minimum threshold values (see Section 4.2.13).

TRACA was run on each problem 20 times, so all results are averages. Since the training sets for each run were identical, each run was started with a different random seed. The only parameter to vary across the three tasks was the number of training epochs. For the results presented below, these were selected based on a small set of initial trials to allow ample training time for the system. For Monk 1 TRACA was trained on the training set for 30 epochs (30 times) before testing. For Monk 2, 50 training epochs were provided before testing and for Monk 3, 40.

5.2.6 Evaluation Criteria

TRACA is evaluated primarily on three criteria. These are: (i) predictive accuracy; (ii) the amount of search conducted in finding a solution; and (iii) the size of the final solution network. Predictive accuracy is calculated as the percentage of correct classifications on the task's test set once training is completed. Search is measured using the number of join groups created and removed during learning and the total number of join groups present in the system on completion of training. The size of the final solution is measured by the number of unsuspended join groups (join groups which are driving system behaviour) present on completion of training. Note that the number of unary groups always remains fixed at the number of bit positions in the input string.

5.2.7 Basic Results

The following briefly summarises performance on each Monk task before comparing TRACA's performance to that of other learning programs.

On Monk 1 (Table 5.1) TRACA achieved an average of 96.7 percent predictive accuracy on the test data. The average number of groups created and removed during the training epochs was 313.8. This gives some indication of how many different join combinations were trialled. However, some combinations may have been trialled more than once as TRACA does not prevent the same group being created more than once, it only prevents duplicates existing at the same time. On completion of training there was, on average, 69.3 join groups in the network. Of these groups, an average of 10.9 were unsuspended join groups - groups whose nodes were actually driving system behaviour and which could be used as subordinate components for other joins.

	Predictive Accuracy	Number of Removed Groups	Total Number of Groups after Training	Number of Unsuspended Groups
Average	96.7	313.8	69.3	10.9
Std. dev.	3.5	28.1	10.0	4.0
Max	100.0	379.0	93.0	20.0
Min	88.7	262.0	53.0	5.0

Table 5.1: Performance - Monk 1.

For Monk 2 (Table 5.2), TRACA achieved an average of 70.1 percent on the test data. The average number of join groups created and removed during training was 1091.7, with an average of 117 existing on completion of learning. Of the existing join groups, an average of 28.8 groups were unsuspended and therefore driving system behaviour.

	Predictive Accuracy	Number of Removed Groups	Total Number of Groups after Training	Number of Unsuspended Groups
Average	70.1	1091.7	117.0	28.8
Std. dev.	2.8	46.0	14.3	4.9
Max	76.6	1197.0	149.0	39.0
Min	66.4	1020.0	99.0	22.0

Table 5.2: Performance - Monk 2.

On Monk 3 (Table 5.3) an average of 92.8 percent accuracy was achieved on the test data. The average number of join groups created and removed during learning was 421.6 with 108.2 in the system on completion of learning of which 24.9 were unsuspended.

	Predictive Accuracy	Number of Removed Groups	Total Number of Groups after Training	Number of Unsuspended Groups
Average	92.8	421.6	108.2	24.9
Std. dev.	3.7	29.7	14.8	5.9
Max	98.2	485.0	127.0	36.0
Min	83.8	377.0	77.0	12.0

Table 5.3: Performance - Monk 3.

5.2.8 Comparing Predictive Accuracy

The results presented in Table 5.4 for comparison are from several other learning algorithms selected from Thrun et al. (1991). Other results from Thrun et al. (1991) obtained by non-incremental algorithms have been excluded. The compared results include Back-propagation neural networks with weight decay, Cascade correlation using QuickProp, ID5R, IDR5-hat and IDL. IDL and IDR5 are incremental induction algorithms for constructing decision trees. Two separate results are reported in Thrun et al. (1991) for ID5R. These are included as ID5R*a* for the results obtained by W. Van de Welde and ID5R*b* for the results gained by J. Krueziger, R Hamann, and W. Wenzel (see Table 5.4).

TRACA's results are the averages of 20 runs on each problem. The three rows in Table 5.4 after the first contain results obtained by W. Van de Welde. These are averaged over 10 runs for Monk 1 and 5 runs for Monk 2. He did not run experiments on Monk 3 because of the noise. However, Van de Welde's results are the among the most reliable, as some of the other results compared to are from a single run. In many cases the results were obtained by the algorithm's inventor. For example, the results for cascade-correlation were obtained by S. Fahlman (Fahlman and Lebiere 1990).

Table 5.4 compares the percentage predictive accuracy of TRACA to the other algorithms. For the first Monk problem, TRACA's result was up to 15 percent higher than the results for both ID5R*a* and ID5R*b*, and 3.3 percent lower than the two neural network approaches. TRACA's results for the second problem were much lower (nearly 30 percent) than the results for both neural networks but higher than the results for the other compared algorithms. For the third problem, due to the noise in the data, only three results are available from the compared approaches, Back-propagation, Cascade-correlation and ID5R*b*. Here TRACA's result was 2.5 percent lower than the result for ID5R*b* and 4.4 percent lower than the results for Back-propagation and Cascade Correlation.

	Monk 1	Monk 2	Monk 3
TRACA	96.7	70.1	92.8
IDL	97.2	66.2	
ID5Ra	81.7	61.8	
ID5R-hat	90.3	65.7	
ID5Rb	79.8	69.2	95.3
Backprop. with weight decay	100.0	100.0	97.2
Cascade correlation	100.0	100.0	97.2

Table 5.4: Comparison of percentage predictive accuracy for Monk problems.

5.2.9 Comparing Training Time

Table 5.5 compares TRACA's training time with the other algorithms. The comparison is based on the number of presentations of the training set. One complete presentation is 1 training epoch. The large number of training examples used by TRACA to obtain the

experimental results in Table 5.4 demonstrate that TRACA's learned solutions are stable across time. The structures required to represent the solution were often found early in training and successfully retained until training is complete (based on manual inspections of the final networks). Section 5.3 discusses this in more detail. An indication of the relative training times required by TRACA and other algorithms can be gained by looking at the training epochs used by different algorithms on the Monk tasks. These are presented in Table 5.5. If the number of training examples used to achieve a result is unavailable this is indicated with n/a.

TRACA results were produced using 30, 50 and 40 training epochs for Monk 1, 2 and 3 respectively. Van de Welde reports his algorithms as being trained using 500 examples randomly selected from the training set. This is approximated in table 5.5 as 4 epochs. The Back-propagation neural network used 390 epochs for Monk 1, 90 for Monk 2 and 190 for Monk 3 while Cascade-correlation used 95 epochs for Monk 1, 82 for Monk 2 and 259 for Monk 3.

	Monk 1	Monk 2	Monk 3
TRACA	30	50	40
IDL	4	4	
ID5Ra	4	4	
ID5R-hat	4	4	
ID5Rb	n/a	n/a	n/a
Backprop. with weight decay	390	90	190
Cascade correlation	95	82	259

Table 5.5: Comparison of training times required for Monk problems.

The results in Table 5.5 suggest that TRACA requires substantially less training experience than the two neural network approaches. On Monk 1 TRACA uses less than 1 tenth of the training required by the Back-propagation neural network and only 1 third of the training required by Cascade-correlation. Similar ratios are true for Monk 3. In Monk 2 the amount of training provided to TRACA was around 2 thirds of that provided to Cascade correlation. This indicates that the higher predictive accuracy of the two neural network approaches is achieved at the cost of many more training examples.

To better assess how TRACA compares to the other algorithms in terms of training times, a series of experiments were run on Monk 1 and Monk 3. These experiments varied the number training epochs provided to TRACA before testing. All other parameters were unchanged from Section 5.2.2 and each result is again an average of 20 runs. Comparisons are based on the 0.05 significance level using the t-Test and also the Mann-Whitney test if the

Wilk-Shapiro test indicated the data was not normal (National Institute of Standards and Technology 2001).

The predictive accuracy of TRACA when using 5, 7, 10, 15, 30 and 40 learning epochs is presented in Table 5.6. These results indicate that TRACA can perform quite well on Monk 3 achieving 91.3 percent accuracy using only 5 learning epochs. With 7 epochs TRACA's performance on Monk 3 almost equals the best decision tree result on this task. However, this performance declines slightly as the number of epochs increases to 10 and 15. T-test results indicate this decrease is significant at the 0.05 percent level and it is possibly due to overfitting. However, the t-Test (but not the Mann-Whitney test) indicates a statistically significant increase between the results for 15 epochs and 30 epochs, while there is no statistically significant difference between the results for 30 and 40 epochs.

For Monk 1 TRACA's accuracy is quite poor with a smaller number of epochs. At both 5 and 7 epochs TRACA's accuracy is around 79.4 percent and it gradually improves as more training epochs are provided reaching 86.4 percent with 10 epochs and 90.1 percent with 15 epochs. The increase in accuracy using 15 epochs is a statistically significant improvement (at the 0.05 level) on the accuracy using only 5 epochs of training. However, 15 epochs is substantially more training experience than the 4 epochs required by the compared decision tree approaches. There is a further increase in predictive accuracy when the number of training epochs is increased from 15 to 30 epochs and the t-Test indicates that this difference is also statistically significant (the Mann-Whitney test does not). There is no statistically significant difference in the predictive accuracy results between 30 and 40 epochs. It also appears that for increased numbers of training epochs beyond 40 the learning curve is quite flat. With 100 epochs of training the predictive accuracy for Monk 1, 2 and 3 is 96.4, 72.1 and 90.2 respectively.

Epochs	5	7	10	15	30	40	100
Monk 1	79.3	79.4	86.4	90.1	96.7	97.8	96.4
Monk 3	91.3	95.1	94.3	89.6	93.7	92.8	90.3

Table 5.6: TRACA's predictive accuracy when varying the training time for Monk 1 and 3.

5.2.10 Comparing Solution Size

Table 5.7 compares the number of unsuspended joins created by TRACA with the size of the trees constructed by the decision tree approaches and the number of hidden-nodes used by the neural network approaches. In the table, the decision tree results include the number of nodes followed by the number of leaves. Nodes in decision trees are roughly equivalent to join groups in TRACA. Where multiple decision tree results are available, the figures in Table 5.7 are for a typical run.

	Monk 1	Monk 2	Monk 3
TRACA	10	29	25
IDL	(36, 26)	(170, 107)	
ID5Ra	(64, 40)	(165, 95)	
ID5R-hat	(49, 32)	(131, 82)	
ID5Rb	(**34**, 52)	(**64**, 99)	(14, 28)
Backprop. with weight decay	3	2	2
Cascade correlation	1	1	3

Table 5.7: Comparison of structures created for the Monk tasks. Decision tree sizes are reported as (number of nodes, number of leaves). The smallest number of nodes for each task is indicated in bold.

Direct comparison of the different learning systems is again difficult. The two neural networks both use at most only a few hidden nodes, however, these each have a large number of connections to input nodes. When compared to the decision trees, TRACA's network looks very efficient. For Monk 1, TRACA creates an average of around 10 join groups, whereas the smallest decision trees required around 34 nodes. For Monk 2, TRACA created on average around 29 joins while the ID5Rb decision trees required around 64 nodes, and the remaining decision tree approaches had well over 100 nodes. It is only on Monk 3 that decision trees appear to be smaller, and this is based on the single result for ID5Rb which had 14 nodes compared to TRACA's 25 join groups.

5.2.11 Summary of Comparisons

The positives about TRACA's performance on the Monk tasks appear to be a smaller and more efficient representation when compared to decision trees and fewer required training examples than the two neural network approaches. The negatives are that decision trees require even fewer training examples than TRACA, while neural networks appear to provide a more compact representation. The smaller representation of neural networks is achieved at the cost of many more training examples while the inefficient representation of decision trees is likely to become an impediment to situated learning when scaling to larger state spaces.

5.3 Qualitative Analysis of TRACA's Generalisation Results

The following sections analyse TRACA's results from Section 5.2.7 on each of the Monk tasks. In particular TRACA's constructed representation for each task is discussed along with any available insights or explanations of its behaviour. This is done without comparison to the compared algorithms due to the very little available analysis available for those systems on the Monk tasks (as noted by Henery (1994)). Some insight in the behaviour of decision tree algorithms on Monk 2 is provided by W. Van de Welde, who mentions that the concepts in Monk 2 appear to be too difficult for decision tree based approaches.

5.3.1 Comparing TRACA's Performance without Joins

Before explaining TRACA's predictive performance, results are presented for a set of experiments on the Monk tasks in which join group creation by TRACA is turned off. TRACA's performance in these cases is based entirely on nodes contained in the default set of unary groups which are created to correspond with each bit position in the input string. All other experimental conditions were unchanged from those described in Section 5.2.2.

The results of these experiments are presented in Table 5.8. The top row of this table repeats the results presented in Section 5.2.7 showing TRACA's predictive accuracy on each of the Monk tasks when joins are created. The bottom row of Table 5.8 presents the new results obtained on each task when joins are not created during learning. The results for the two sets of experiments (with and without joins) on each task were compared using both t-Tests and, if the data was not normal according to the Wilk-Shapiro test, a Mann-Whitney test was also applied. For the Monk 1 task, the experiments without joins achieved only 72.8 percent predictive accuracy on the test set, this was 23.9 percent lower than the 96.7 percent accuracy obtained in the experiments in which joins were created. On Monk 2, the experiment without joins achieved 67 percent accuracy, only 3.1 percent lower than the experiments using joins. The t-Test indicated a statistically significant difference, however, the data was not normal and the Mann-Whitney test indicated no statistically significant difference. Finally, on Monk 3, there was only a 0.1 percent difference between the results obtained for the two experiments. No significant statistical differences were detected between the results of the two experiments.[4]

	Monk 1	Monk 2	Monk 3
Using Joins	96.7	70.1	92.8
Without Joins	72.8	67.0	92.9

Table 5.8: Comparison of predictive performance by TRACA on the Monk tasks with and without the creation of join groups (join creation is turned off).

5.3.2 Analysis of Monk 1 Performance

To represent the concepts for the Monk 1 problem we would like the system to create joins to represent the conditions when A_1 and A_2 both have a 1 in the same relative bit position of their substrings. However, we do not want the system to create join structures unnecessarily, if a group is being predicted correctly then further structures to predict it should not be created. One method to detect when a structure is being correctly predicted is to look at the ETP of the nodes predicting it; if the ETP of one of these nodes has an asymptote of 1.0, then the node is always making correct predictions. The creation of further joins to predict a group should not be necessary in situations when the group is being accurately predicted and it is possible to configure TRACA (but this was not done in these experiments) so that it will not create new joins in these situations (this can be achieved by specifying a threshold value close to 1.0 which when exceeded by a node's dependent ETP prevents join group creation by the node's predicted group). In the Monk 1 problem, the nodes associated with the unary group representing $A_5 = 1$ will always be correct. In this case, any join groups created which use the unary group representing $A_5 = 1$ as a subordinate will be eventually removed, since it is impossible for any nodes in the join to achieve a higher ETP value than their equivalent subordinates.

In the Monk 1 task, attributes A_3, A_4 and A_6 are irrelevant and join structures created using these attributes should eventually be removed (and typically are). To represent the situations when the input case is friendly, TRACA needs to create join groups for the combinations where $A_1 = A_2$. However, in situations where $A_1 \neq A_2$ and $A_5 \neq 1$ TRACA needs to be able to use the remaining bit positions to match groups which will allow it to make the correct response. This can be done using unary groups, however, given that TRACA's objective is to discover useful structures that can be later built on, ideally TRACA should develop joins to represent situations where $A_1 \neq A_2$ (note that this is not a minimal description nor is it intended to be).

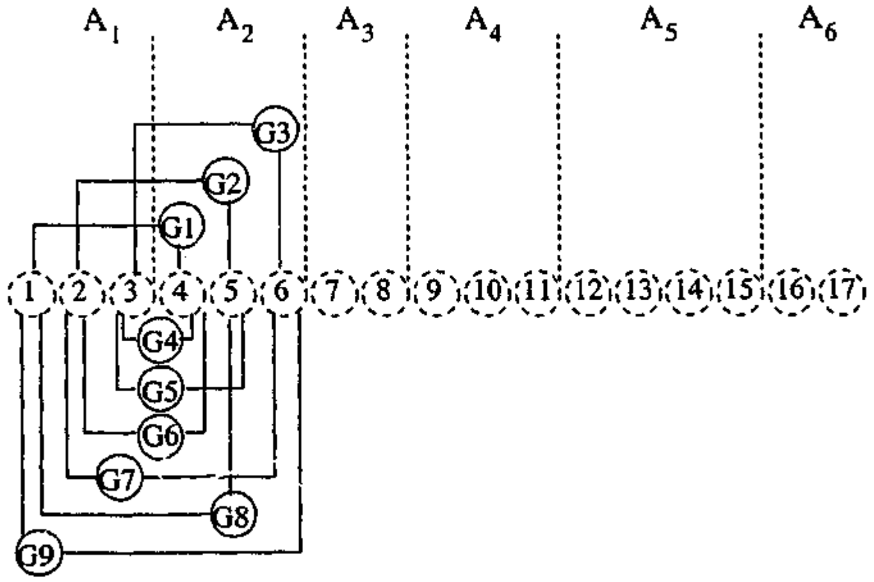


Figure 5.2: Join groups required to represent the true concepts for Monk 1 (dashed lines show the borders of each attribute's bit positions).

---

[4] Experiments in which join creation was turned off were also conducted on a range of other tasks in Mitchell (2002). On all these tasks there were also very small differences in the predictive performance of TRACA with and without join creation turned off. Monk 1 and Monk 2 were the most obvious exceptions to this.

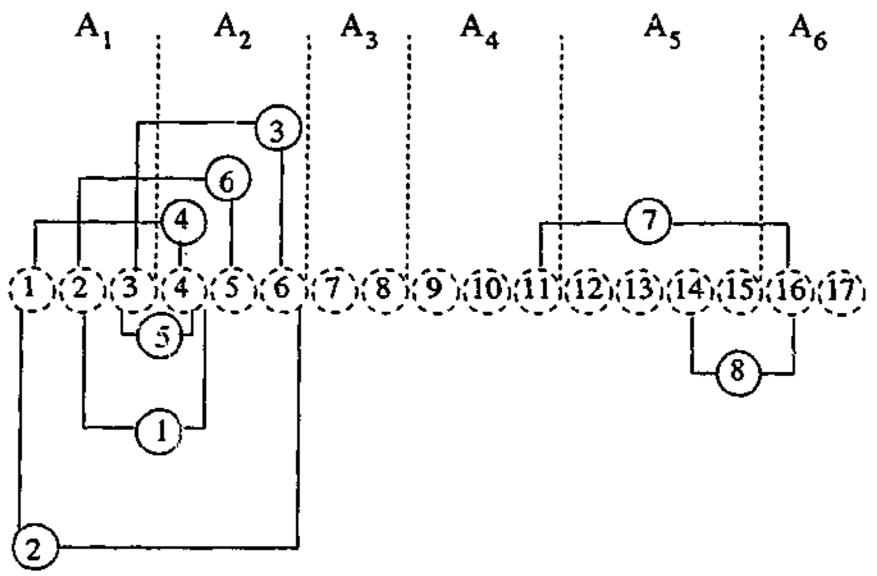
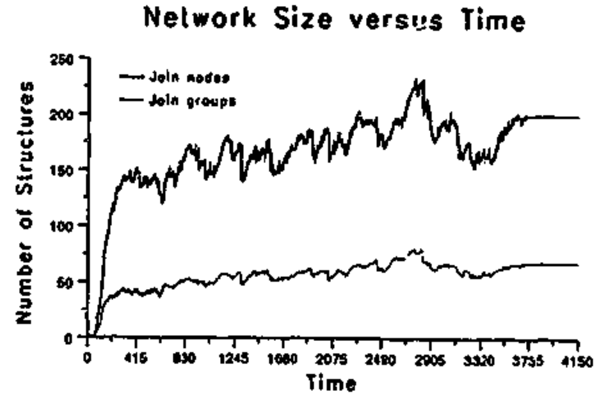


Figure 5.3: Unsuspended join groups actually created to represent the concepts for Monk 1 in one run in which 100 percent test accuracy was achieved (groups are labelled in order of creation).

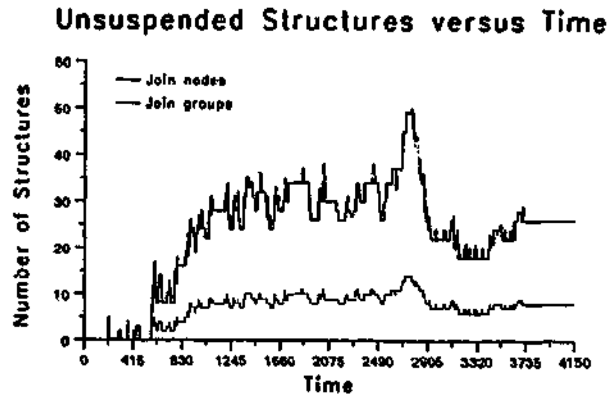
The ideal structures to represent the Monk 1 concepts are presented in Figure 5.2. The joins labelled G1 through G3 represent friendly cases. The joins labelled G4 through G9 represent situations where the case is unfriendly, but only if $A_5 \neq 1$. Joins of groups G4 to G9 with the unary group for bit position 12, which would represent the logical **not** conditions, are excluded from the ideal description. This is because they will never be retained in the system because no node in these joins will improve on the ETP of the nodes in the subordinate unary group since the subordinate nodes have an asymptote of 1.0. The fact that there are no joins connecting the unary group for bit position 12 to the groups G4 through to G8 would suggest that TRACA cannot accurately represent this problem, since groups G4 to G8 are not support suppressed when $A_5 = 1$. In cases such as this, the fact that the nodes in the group representing $A_5 = 1$ are always correct (therefore their ETP values are close to 1.0) can be used to resolve any ambiguity.

The best performance TRACA obtained on Monk 1 was 100 percent. The unsuspended join groups developed during one run in which 100 percent test accuracy was obtained is presented in Figure 5.3. We see that the conditions for identifying the friendly cases are all represented, however, not all the join groups necessary for unfriendly cases appear (in particular, join groups G5, G7 and G8 from Figure 5.2). In actual fact, the remaining join groups often have been created, but by the completion of the learning trials, nodes in these groups had not yet reached a value which would allow them to be evaluated for removal or retention in the system. This is due to the fact that the joins for the friendly cases, combined with the utility values of nodes in unary groups, are driving system behaviour such that the nodes in joins representing the unfriendly cases rarely get executed. This indicates that the use of joins to represent all possible unfriendly cases is unnecessary to achieve 100 percent classification accuracy and the nodes in unary groups are often sufficient.

We also see that some joins are formed on irrelevant attributes (groups 7 and 8). This occurs because at different stages during learning various combinations may appear more useful than they really are. In this case, the joins do not prevent TRACA achieving 100 percent accuracy on the test data due to the values of nodes in other groups, however, in general this is a difficulty for TRACA while learning Monk 1. Joins of irrelevant attributes often appear to perform quite well especially when many of the correct structures have been discovered. Most of these joins are removed once a sequence of examples is experienced that reveals their true value, however, because their execution frequently coincides with correct effector selections some are retained for lengthy periods before removal.



(a) Network size over time.



(b) Unsuspended structures over time.

Figure 5.4: Network growth during one run on Monk 1. Training continues for the first 3720 timesteps (30 epochs) and during this time random actions are selected with probability 0.25. Once training is completed, learning and random action selections are turned off.

Figure 5.4(a) shows network growth during one run on Monk 1. These peak with around 230 nodes in 75 groups after 2800 training examples finishing at 200 and 68 respectively on completion of training at time 3720. Figure 5.4(b) shows the number of unsuspended joins during the same run. This curve peaks at around timestep 2800 with approximately 50 nodes in 13 join groups.

5.3.3 Analysis of Monk 2 Performance

The Monk 2 problem requires more join groups for its solution than either of the Monk 1 or Monk 3 problems. Its solution requires joins for each possible pairing of unary groups which represent the value 1 for each attribute (the left most bit position will be matched for each attribute that has a 1 in its input substring). There are 15 such combinations that must be made, and for each of these TRACA must create superior groups to exclude the situations where any of the attributes other than the join's pair have the value 1. Therefore, we have a minimum of 15 × 4 = 60 join groups that need to be discovered.

The join structures required for one pair of attributes with the value 1 is shown in Figure 5.5, however, there are many more complex variations which are equivalent. The structures in Figure 5.5 also demonstrate how TRACA may reuse groups, with G1 used as a subordinate structure for all the groups from G2 to G4, which combined correctly represent the structure required for one pair of attributes.

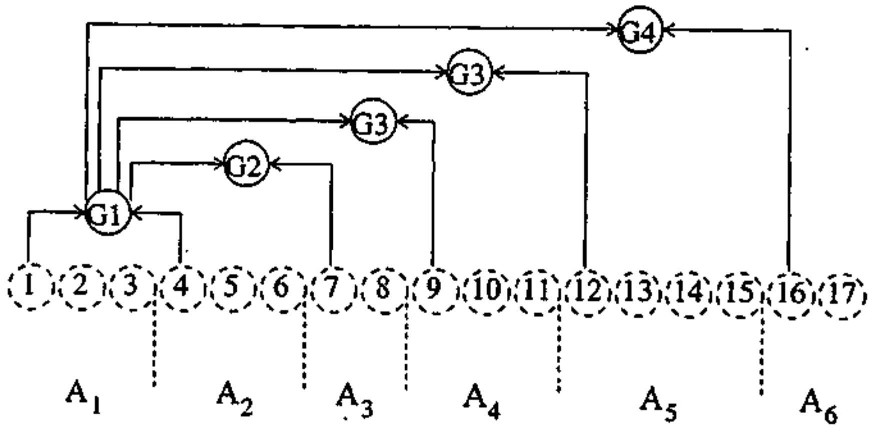
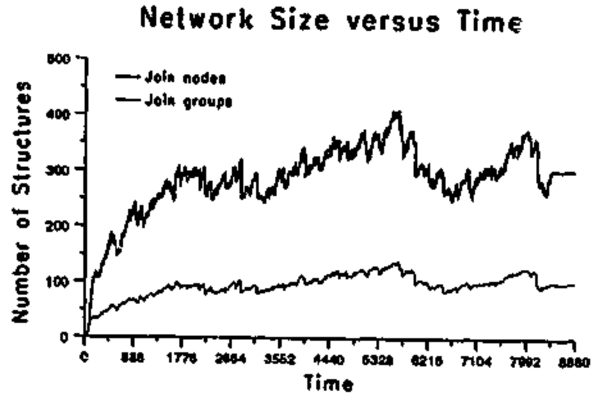


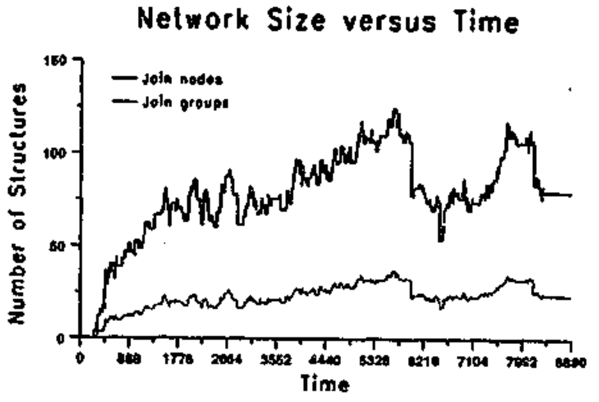
Figure 5.5: Join groups required for just one pair of attributes to solve the Monk 2 problem (dashed lines show the borders of each attribute's bit positions).

During the trials with TRACA a complete solution for this problem was never found. TRACA is capable of learning some of the required structures; an extract of the joins discovered during one run is presented in Figure 5.7(a). One difficulty in solving this problem is that joins are formed that are too specialised (see Figure 5.7(b) for an example). While these specialised joins are useful in the absence of more general joins, they add to the complexity of the representation. Specialised joins also add to the total number of joins in the system and attempts by the system to build on them result in more complex structures which in turn leads to fewer opportunities to discover the more general rules. These problems, along with the relative infrequency of the useful joins being correct, prevent the value of nodes in useful groups from rising, which would normally occur as system performance improves (as predictions are correct more often). This allows an increased presence of joins of marginal use, which are retained in the system because they appear useful when their values are compared to the values of their subordinates (see Figure 5.7(c)).

The presence of these joins allows more over-specific joins further reducing opportunities for the creation of other, more useful, joins.



(a) Network size over time.



(b) Unsuspended structures over time.

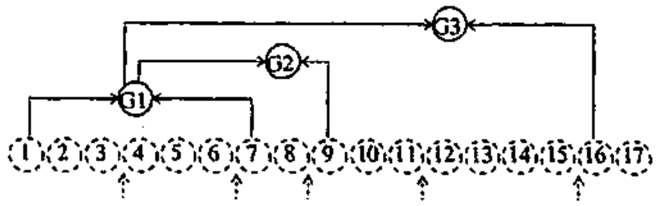
Figure 5.6: Network growth during one run on Monk 2. Training continues for the first 8450 timesteps (50 epochs) with random actions selected with probability 0.25. Once training is completed, learning and random action selections are turned off.

Clearly the Monk 2 task causes TRACA difficulty. The evidence provided by the poor performance of other learning systems on this task (with the exception of neural networks) suggests this problem is inherently hard. Indeed, in Thrun et al. (1991) it is noted that this problem is complex to describe in disjunctive normal form (which is similar to the structures TRACA creates). Van De Welde also later notes that this problem is difficult for decision tree based methods. The problem is one in which the relevance of the information of attributes is difficult to detect in isolation of other attributes.[5]

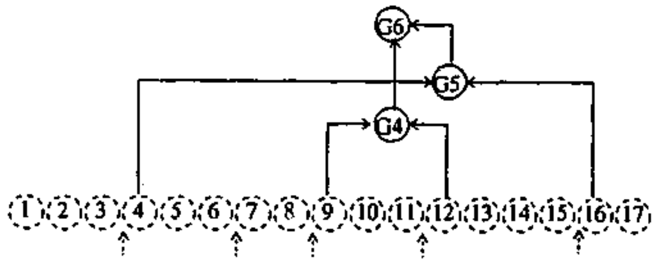
Figure 5.6 shows network growth during one run on Monk 2. Figure 5.6(a) shows the total number of join nodes and join groups over time. This peaks with around 400 nodes in 140 groups after 5800 training examples finishing at 302 and 101 respectively on completion of training at time 8450. Figure 5.6(b) shows the number of unsuspended joins during the same learning run. This peaks around timestep 5800 with approximately 125 nodes in 70 join groups.

---

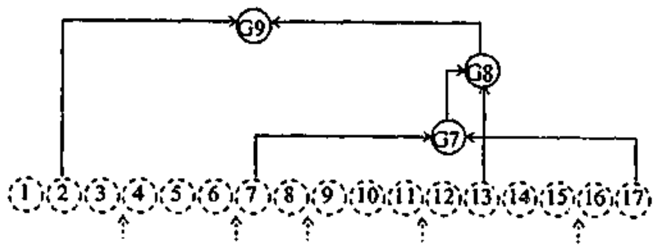
[5] Chapman and Kaelbling (1991) note that this type of problem also occurs with their G-algorithm and suggest that it may be overcome in some cases by using appropriately orthogonal features.



(a) Join groups developed for one pair of attributes.



(b) An overly specific join.



(c) Joins of marginal usefulness.

Figure 5.7: Joins formed in one run on Monk 2 (dashed arrows show the borders of each attribute's bit positions).

5.3.4 Analysis of Monk 3 Performance

One difficulty in solving the Monk 3 task is that it has noise present due to the fact that some of the examples in the training set are incorrectly classified. TRACA's structures for the ideal solution to the Monk 3 problem are presented in Figure 5.8. Figure 5.9 shows some of the joins developed during a sample run which also appear in the ideal solution. TRACA's created join structures correctly represent a number of correct predictive relationships in the data. These joins reflect the fact that regardless of the noise, many different hypotheses describe the Monk 3 training data very well.

The presence of good alternative hypotheses in the Monk 3 training data become apparent when the structures created during one sample run are investigated. Some joins of input string bit positions discovered by the system during the selected run learning that contained nodes which were always correct during training were: (6, 1), (6, 10), (6, 12), (10, 15) and (6, 8). These joins are not shown in Figure 5.9 as they could be excluded from a minimal description of the data.

At the end of the sample run, eight other joins which did not represent the declarative description provided in Section 5.2.1 were also present as unsuspended groups. These were all joins where one or more nodes in the join group demonstrated a predictive improvement over equivalent subordinate nodes, again for clarity they have been omitted from Figure 5.9. In a strict assessment of TRACA's learned representation these groups would ideally not be present, as they are not required for a minimal description of the data. However, it is likely that nodes in unary and join groups which are not part of the minimal description are useful for classifying cases when the joins required to complete the minimal description have not been discovered.

As it turns out, during the sample run (as shown in Figure 5.9), TRACA discovered nearly all of the joins required to represent the declarative description with the exception of the group represented as G3 in Figure 5.8.

Figure 5.10 shows network growth during one learning run on Monk 3. Figure 5.10(a) shows the total number of join nodes and join groups over time. This peaks with around 300 nodes in 100 groups after 3186 training examples finishing at 229 and 95 respectively on completion of training at time 4880. Figure 5.10(b) shows the number of unsuspended joins during the same learning run. This peaks at around timestep 4500 with approximately 80 nodes in 25 join groups.

5.3.5 Summary of Performance Analysis

TRACA reliably creates structures which capture the relationships in the data for the Monk 1 task. On this task few structures are created which are based on the irrelevant attributes. Furthermore, join structure is necessary for good predictive performance on this task as

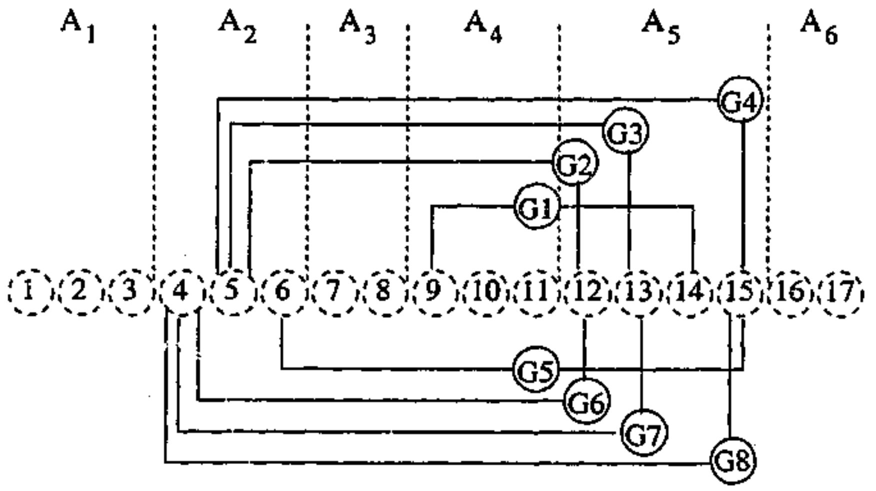


Figure 5.8: Join groups required to represent the true concepts for Monk 3 (dashed lines show the borders of each attribute's bit positions).

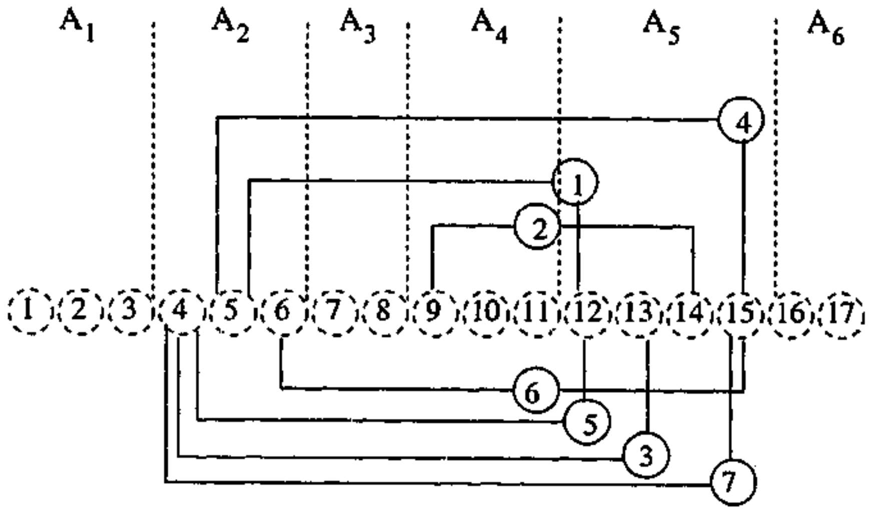
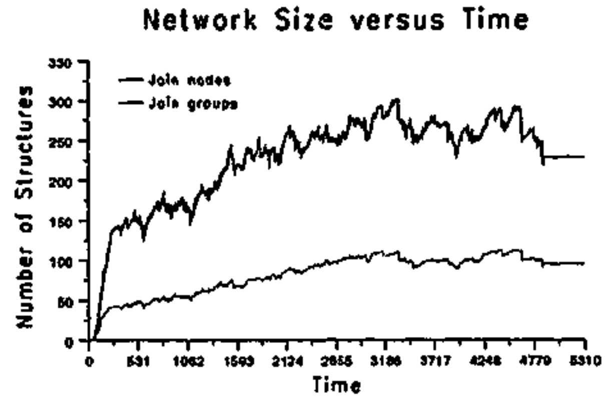
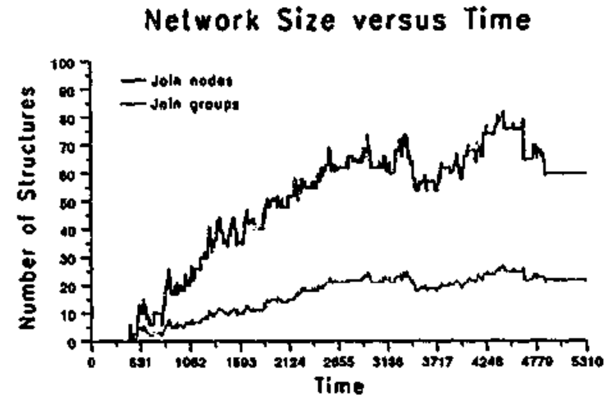


Figure 5.9: Some of the unsuspended join groups actually created in one run to represent the concepts for Monk 3 (labelled in order of creation).



(a) Network size over time.



(b) Unsuspended structures over time.

Figure 5.10: Network growth during one run on Monk 3. Training continues for the first 4880 timesteps (40 epochs) with random actions selected with probability 0.25. Once training is completed, learning and random action selections are turned off.

indicated by the relatively poor performance obtained in experiments in which join creation was turned off.

It is difficult to draw any conclusions on TRACA's performance on the Monk 2 task, other than the fact that TRACA has difficulty solving this task. This difficulty is most likely due to the combination of two factors; one is that so many joins are required to represent this task and the other is that the relationships between attributes are difficult to detect.

Finally, there are the results for Monk 3. On this task, TRACA finds a number of predictive relationships which explain subsets of the data quite well. These can be viewed either positively as alternative solutions, or negatively as redundant given the presence of other predicting nodes and groups. On this task TRACA also retains some joins containing nodes that provide an improvement over their equivalent subordinates, but are not necessary for the solution. It is difficult to prune these, as without prior knowledge of the environment, the system cannot determine whether or not they may be useful building blocks for other more reliable structures.

5.4 Effects of Parameter Changes on Generalisation Performance

In all the experiments presented so far (in which join creation was turned on) the same parameters were used, except for Section 5.2.9 in which the number of training examples (epochs) was varied. Now the effects of varying other parameters are investigated.

Of all the parameters, the learning rate is one of the most interesting. Because nodes' ETPs are recency-weighted, a high learning rate allows some nodes to approach their steady-state values quickly, while for others it causes wild fluctuations in values. For example, in the structures for Monk 1 presented in Figure 5.2, nodes in groups G1 to G4 will make reliable predictions and approach an asymptote of 1.0 which is reached quickly using a high learning rate. However, nodes in groups G4 to G9 do not have an asymptote of 1.0 as their predictions will only be correct depending on whether the unary group for bit position 12 is matched or not (attribute 5 in Section 5.2.1). Since groups G4 to G9 do not encapsulate sufficient conditions to make reliable predictions, their values will oscillate around a baseline value somewhat less than 1.0.

Another interesting parameter is the exploration rate. In experiments presented in this chapter so far, the exploration rate is fixed and exploration stops when training trials complete. However, it is possible that for many problems, such as the truck driving task in Section 7.8, better results can be obtained if exploration is reduced over time (as found by McCallum (1995)).

Also investigated is the effect of varying the number of cases used in the Cox-Stuart test for trend and the test for noise. Finally, a comparison is made with a run of the system based on the ability of groups to provide utile improvements rather than perceptual improvements.

Monk 1 is used exclusively for the following experiments as it is one task for which using joins leads to a substantial improvement in performance. In the following statistical analysis each distribution was tested for normality using the Wilk-Shapiro test. A variance stabilising transformation was also used in which percentages, p are converted to a score x using $x = asin(\sqrt{p/100})$ before being tested for normality (Box, Hunter, and Stuart 1978). All data is compared using parametric techniques such as the t-Test. For data that tests indicate is not normal, non-parametric techniques are also used, specifically the Mann-Whitney and Kruskal-Wallis tests. Discrepancies in these two types of test are reported where they occur. Statistically significant differences at the 0.05 level are refered to as *significant*.

5.4.1 Effects of Varying the Exploration Rate

Table 5.9 shows the effects of varying the exploration rate for the Monk 1 task. A rate of 1.0 indicates that every action was randomly selected, a rate of 0.1 indicates that 1 in 10 actions were randomly selected and a rate of 0 indicates that the effector with the highest support was always selected. Other parameters are unchanged from Section 5.2.

The following discussion divides the results presented in Table 5.9 into two groups; results obtained with explicit exploration and results obtained without explicit exploration. The first group includes the results for exploration rates from 1.0 to 0.1 while the second group includes the results for the exploration rate of 0.

Exploration Rate	1.00	0.50	0.33	0.25	0.10	0.00
Test Accuracy	85.8	90.0	94.0	96.7	96.2	96.2
Std. dev.	4.0	5.2	7.3	3.5	5.6	3.7
Max	94.9	97.7	100.0	100.0	100.0	100.0
Min	77.8	79.4	76.6	88.7	80.8	85.9
Unsuspended	22.0	17.0	13.9	10.9	9.4	14.1
Current	97.0	84.6	74.7	69.3	64.3	75.1
Removed	271.0	298.9	322.6	313.8	325.9	289.9

Table 5.9: Performance on Monk 1 when varying exploration rate (averages of 20 runs).

The reason for this division is that there are several interesting effects at the extreme values of 1.0 and 0. The first effect is observed in the number of unsuspended joins. This starts quite high at the extreme value of 1.0 with 22 joins. As the exploration rate is reduced there are a series of significant decreases (according to the t-Test, but not the Mann-Whitney test in all but the first case where the data was normal) in the number of unsuspended joins from 22.0 to 10.9, which bottoms out at 9.4 with the exploration rate of 0.1. But at the extreme exploration rate of 0, there is a significant increase up to 14.1.

The second effect observed is the number of current joins (both suspended and unsuspended) in the system on completion of learning. This follows a series of significant decreases as the exploration rate is reduced from a high of 97.0 (with the exploration rate of 1.0) through to a low of 64.3 (with the exploration rate of 0.1). In this case, with the extreme exploration rate of 0, there is a significant increase in the number of current joins to 75.1.

\ third effect is noticeable with the number of joins generated and removed during learning. This gradually increases from a low of 271.0 with the extreme exploration rate of 1.0 with two significant increases by the time the exploration rate reaches 0.33. It peaks at 325.9 with the exploration rate of 0.1 before a significant decrease to 289.9 with the extreme exploration rate of 0.

A final, slightly different, effect is evident in the changes to predictive accuracy as the exploration rate is reduced. The lowest predictive accuracy is 85.8, at the exploration rate of 1.0. Following this there is a series of significant increases in predictive accuracy until the peak of 96.7 with an exploration rate of 0.25. This peak is followed by a significant drop to 96.2 with the exploration rate of 0.1 which remains unchanged with the extreme exploration rate of 0.

An explanation of the anomalies associated with the exploration rate of 0 (or when approaching 0 in the case of predictive accuracy) is defered for now to follow the description of the more general trends experienced in the results with the other exploration rates. These trends include a gradually increasing number of removed joins and a gradually decreasing number of current and unsuspended joins.

The result of a manual comparison of sample structures created during two different runs helps explain these trends. The exploration rate for the first run was 1.0 and for the second run was 0, all other parameters were the same. The two runs resulted in similar learned structure with 21 and 22 unsuspended joins created respectively. There were also no apparent differences in the number of joins created using relevant or irrelevant attributes. However, the two runs had significantly different predictive accuracies. The run with the exploration rate of 1.0 achieved 86.1 percent predictive accuracy, while the run with an exploration rate of 0 achieved a predictive accuracy of 100 percent.

The differences in the performance of these two runs was due to the different relative ETP and utility values of the nodes within the joins. This is evident when these values are investigated manually for each run on completion of learning. Two nodes selected from a join group which represented the true concept (as was shown in Figure 5.2) both had utility values close to 100. However, the same nodes in the run with an exploration rate of 1.0 had utility values around -50. There were similar differences in the ETP's of nodes. In the run with an exploration rate of 0, ETP's of nodes in groups representing both relevant and irrelevant attributes were often close to either 1.0 or 0, depending on the node's prediction whereas in the run with the exploration rate of 1.0, the ETPs of nodes were often around 0.5, regardless of the node's prediction.

This example demonstrates how the very act of learning itself can affect the ETP and utility estimates of nodes. A low exploration rate allows the system, as it learns, to drive the ETP's of nodes in groups representing relevant attributes to extreme values. This type of behaviour was mentioned in Section 3.5.1 which highlighted its possible detrimental effects in the absence of a suppression mechanism. However, on problems with irrelevant attributes, such as Monk 1, this behaviour improves TRACA's performance by reducing the number of unsuspended joins created and allowing less useful joins to be identified easily and removed. This allows more opportunity for other joins to be created. The process of creating new joins to predict an individual unary group can eventually be stopped once the unary group is always reliably predicted.[6]

The anomalous effects described earlier which occur with an exploration rate of zero, are explained by the fact that a small amount of exploration is required to assess unnecessary joins independently of the joins required for a correct solution. As the required joins begin to drive system behaviour they protect some unnecessary joins from removal by preventing their nodes from executing incorrectly. This explains the larger number of unsuspended joins. It also explains the smaller number of joins created during learning; the retention of joins which would otherwise be removed reduces the opportunities for other joins to be created.

These results show that the amount of search conducted and the amount of structure created by TRACA is dramatically affected by exploration. Using higher exploration rates for problems which contain irrelevant attributes increases the difficulty of distinguishing relevant attributes from irrelevant attributes, resulting in more joins being retained. On the other hand, a very low (close to zero) exploration rate can result in some unsuspended joins being protected from removal during training. However, the differences in predictive accuracy appear to be primarily due to differences in the ETP and estimated utility values rather than the number and type of unsuspended joins created during learning.

5.4.2 Effects of Varying the Learning Rate

When the learning rate is increased, utility and transition estimates of nodes which make non-deterministic predictions oscillate more wildly around their baseline value (due to TRACA's recency weighting). This disruption to convergence may cause join groups to be removed which might otherwise have been retained in the system. Such an effect is evident in the results presented in Table 5.10, which shows the effects of varying the learning rate for Monk 1 from 0.05 to 0.7. As the learning rate increases from 0.05 to 0.6, there is a significant decrease in the number of unsuspended joins and a significant increase in the number of joins removed during learning (other parameters are again unchanged from Section 5.2).

However, while the standard deviations for predictive accuracy generally increase as the learning rate increases there were no significant statistical differences in the predictive

---

[6] How quickly this occurs depends on how quickly predicting nodes' ETPs reach the threshold value. Reaching this depends in turn on the learning rate.

Learning Rate	0.05	0.1	0.3	0.5	0.6	0.7
Test Accuracy	95.5	96.7	95.4	95.4	94.2	92.4
Std. dev.	3.5	3.5	5.0	6.0	4.7	7.3
Max	100.0	100.0	100.0	100.0	99.8	100.0
Min	87.7	88.7	81.9	82.6	83.8	77.3
Unsuspended	11.4	10.9	8.1	7.0	6.5	7.3
Current	71.7	69.3	57.7	54.1	53.4	53.5
Removed	281.2	313.8	399.6	427.7	430.9	427.7

Table 5.10: Performance on Monk 1 when varying the learning rate (averages of 20 runs).

accuracy across the different learning rates (according to the Kruskal-Wallis test). Manual inspections of the structures created during runs revealed that many of the unsuspended joins present on the completion of training were those required to represent the true concepts and that these were retained once discovered. This is particularly true for the groups representing G1, G2 and G3 in Figure 5.2, all of which contain nodes whose ETPs approach an asymptote of 1.0. However, groups representing G4 to G9 in Figure 5.2 were often missing. This suggests that the consistent presence of join groups representing G1 to G4 with unary groups is sufficient in this task to prevent a significant difference in predictive accuracy as the number of unsuspended joins decreases.

5.4.3 Effects of Varying the Number of Statistical Cases

The two statistical tests used in TRACA are the test for noise (Section 4.2.11) and the Cox-Stuart test for trend (Section 4.2.9). Results for varying both of these across 40, 20, 10 and 4 cases on Monk 1 are presented in Tables 5.11 and 5.12 below. In both cases, other parameters are unchanged from Section 5.2.

Table 5.11 presents the results of varying the number of cases used for the test for noise (Section 4.2.9). It shows that the predictive accuracy rises from 92.8 with 40 cases peaking at 10 cases with a dramatic decrease to 71.8 when only 4 cases are used.[7]

T-tests indicated a statistical difference between the predictive accuracy obtained with 40 cases when compared to the accuracies obtained with 20 and 10 cases. However, the distribution for predictive accuracy with 40 cases was not normal and the Mann-Whitney test indicated no difference. Excluding the results for only 4 cases, there were clear statistical differences between the number of groups removed for each case count, but not for the number of current groups nor the number of unsuspended groups on completion of learning.

Table 5.12 presents the results of experiments which varied the cases used in the Cox-Stuart test for trend. As the number of cases decreased from 40 to 10, predictive accuracy gradually

---

[7] The predictive accuracy of 98.4 using 10 cases reflects the fact that optimisation of parameters was not performed for each of the initial experiments in Section 5.2 which used 20 cases.

Case count	40	20	10	4
Test Accuracy	92.8	96.7	98.4	71.8
Std. dev.	5.5	3.5	2.1	2.1
Max	100.0	100.0	100.0	75.0
Min	83.1	88.7	91.7	68.1
Unsuspended	11.2	10.9	10.8	1.1
Current	71.3	69.3	64.5	36.6
Removed	240.9	313.8	399.9	478.1

Table 5.11: Performance on Monk 1 when varying the number of cases for the test for noise (averages of 20 runs).

increased from 95.3 percent to 96.7. When the number of cases is reduced to 4 the predictive accuracy drops, however, it was not until the number of cases was reduced to 2, at which point predictive accuracy dropped to 80.3, that there was a significant change. Similarly, there were no significant differences in the number of unsuspended groups and the number of groups present in the system on completion of training until the number of cases reduces to 4. In these cases, the t-Test indicated a significant difference, however, the data for 4 cases was not normally distributed, and the Mann-Whitney test indicated no significant difference. It is not until the number of cases drops to 2 that both tests indicate (not surprisingly) a significant difference. On the other hand, there were statistical differences in the number of groups removed with each change in the number of cases used. This indicates that as the number of cases used in the test for trend is reduced, the number of joins created and removed during learning increases.

Case count	40	20	10	4	2
Test Accuracy	95.3	95.5	96.7	92.3	80.3
Std. dev.	5.3	4.5	3.5	3.8	6.7
Max	100.0	100.0	100.0	100.0	93.1
Min	83.6	82.4	88.7	84.3	67.8
Unsuspended	10.5	11.5	10.9	6.1	5.1
Current	66.3	67.5	69.3	43.2	29.9
Removed	264.3	280.3	313.8	1261.2	2339.7

Table 5.12: Performance on Monk 1 when varying the number of cases for the Cox-Stuart test for trend (averages of 20 runs).

One may have expected predictive accuracy to change as the number of statistical cases varies. For example, allowing more Cox-Stuart cases allows more time for nodes to reach an asymptotic or baseline value. Similarly, allowing more cases for the test for noise may have been expected to make groups less susceptible to short runs of events. In fact, the results provide no evidence that varying the number of cases in either of these statistical tests affects predictive accuracy, with the exception of very low numbers, such as 2 and 4. However, varying the number of cases for both these statistics can have significant effects on the amount of search conducted (i.e joins created and removed) during learning.

5.4.4 Changing the Action Selection Strategy

In the experiments so far, the *average support* action selection strategy has been used. Using this strategy, all unsuspended, un-support suppressed nodes send support and the effector with the highest average support has its action executed. An alternative is the *best supporter* strategy, which selects an effector based on the sum of two individual nodes. The node which sends the highest support and the node which sends the lowest support.

Both these support strategies allow for conflicting rules due to an inadequate rule set due to either in-experience in the environment or other difficulties in representing the correct concepts. For example, the system may contain a node which predicts that picking up precious opals found on the ground will lead to a high reward. However, another rule may indicate that approaching dangerous animals should be avoided. In the novel situation of an opal sited next to a large crocodile the two rules may both send values for the action *approach-object*, one sending positive support and the other negative support. Using the sum of these two values as the support for the action reflects the ambiguity inherent in the two rules. This is the *best supporter* strategy which is a type of "winner pair of rules take all". The *average support* strategy is similar, except that all rules may influence the support for an effector. A risk with this strategy is that the support of many rules may act as noise confusing the decision. This seems particularly likely in TRACA when irrelevant attributes are present, since all unary groups develop nodes which send support, regardless of their usefulness (there is no test to eliminate them).

A comparison of the predictive accuracy using both strategies for the three Monk tasks is presented in Figure 5.13. The results indicate very little difference in performance for the two different methods. Only on Monk 1 was it found to be statistically significant. The predictive accuracy using the best supporter method was lower on Monk 1 (3.4 percent) and only slightly higher on Monk 2 and Monk 3 (0.3 and 0.5 respectively). It is likely that this lower performance on Monk 1 using the best supporter strategy is due to the fact that the join groups representing unfriendly cases are not set to unsuspended, but a detailed investigation into this has been not conducted.

	Monk 1	Monk 2	Monk 3
Test Accuracy for best supporter	93.3	70.4	93.2
Test Accuracy for average support	96.7	70.1	92.8

Table 5.13: Performance on Monk problems when changing the action selection strategy (averages of 20 runs).

5.4.5 Making Utile rather than Perceptual Comparisons

The results presented so far have all been based on TRACA retaining groups which demonstrate their ability to predict percepts (better ETPs). However, it is also possible for TRACA to retain groups based on their utility. Making this switch would change TRACA from a system which makes comparisons based on perceptual improvements into one which makes comparisons based on utile improvements (McCallum 1995). In the simplest case, this involves replacing the ETP values used for the Cox-Stuart test for trend (see Section 4.2.9) and the test for noise (described in Section 4.2.11) with the utility estimate of nodes. The results of a version of the program with these changes is presented in Table 5.14. These results were produced under the same experimental conditions as the Monk results presented in Section 5.2.7.

	Monk 1	Monk 2	Monk 3
Test Accuracy	94.9	66.6	94.5
Std. dev.	5.2	2.4	2.9
Max	100.0	70.1	98.2
Min	86.1	60.9	88.2
Unsuspended	8.6	20.1	27.4
Removed	246.2	1042.0	334.8

Table 5.14: Performance on Monk problems based on utile improvements.

For Monk 1 there was no statistically significant difference in predictive performance when learning was based on utile improvements. The predictive accuracy using utile improvements was only 1.8 percent lower than the result obtained using perceptual improvements. For Monk 2 and 3 there were significant differences in accuracy. The accuracy for Monk 2 when making utile distinctions was 3.5 percent lower than the perceptual accuracy, while the utile based accuracy for Monk 3 was 1.7 percent higher than its perceptual equivalent. There were also significant differences in the number of structures created and removed during learning between the utile and perceptual methods. The utile method created slightly less structure during learning for all three monk problems (for comparison, see Tables 5.1, 5.2 and 5.3). Significant differences were also found between the number of unsuspended structures created on completion of training for Monk 1 and Monk 2 (but not Monk 3). The utile method created an average of 8.6 unsuspended joins versus 10.9 for Monk 1 and for Monk 2 it created 20.1 versus 28.1. The number of unsuspended joins created for Monk 3 was slightly higher using utilities, with 27.4 join groups versus 24.9 when making perceptual improvements.

These results suggest that TRACA can be successfully used to create and retain joins based on their ability to predict utility as well as their ability to predict percepts. On Monk 1 and 2 the utile method created less structure and achieved lower predictive accuracies, while on Monk 3 it created more structure and achieved a higher predictive accuracy.

5.5 Discussion of Generalisation

TRACA's performance has been compared to that of a number of other incremental learning algorithms on the Monk problem. On Monk 1 and Monk 3, TRACA's predictive accuracy was within 4.5 percent of the best performing algorithm. Monk 2 presents serious difficulties for most of the compared algorithms and also for TRACA. On Monk 2, TRACA's accuracy was well below the two best performing algorithms, but was higher than the other compared algorithms. TRACA's created structure has also been analysed for each task. For Monk 1 this analysis revealed that the correct structures were regularly created to efficiently represent the problem, however, frequently some structure required for a complete solution was missing. Furthermore, structures incorporating irrelevant attributes were occasionally incorrectly retained. Analysis of Monk 2 revealed that TRACA had difficulty finding the most efficient representation, often creating structures which provided predictive improvements but whose presence complicated the solution and increased the search space. Finally, on Monk 3, TRACA often found useful relationships that were present in the data even those that were not required for a minimal description of the data.

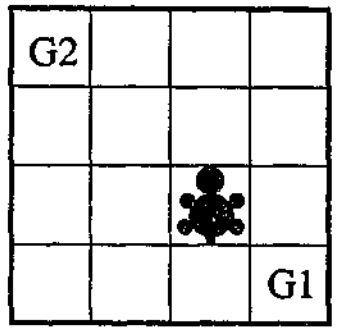
TRACA was also trialled using a variety of different parameters for Monk 1. While TRACA appears robust to parameter changes they did affect performance. The results indicate that tuning of parameters can be used to improve predictive accuracy, reduce the number of joins created during learning and reduce the size of TRACA's final learned network.

Two additional experiments were run on all three Monk tasks. One to change the action selection strategy the other to retain structure based on utile rather than perceptual distinctions during learning. Changing the action selection strategy had little effect on TRACA's performance on the three Monk tasks. TRACA's high accuracy when making utile distinctions demonstrates that TRACA is capable of retaining joins based on utile improvements resulting in a similar predictive accuracy as when assessments are based on perceptual improvements.

5.6 Rule Chaining

The Monk tasks test the ability of a system to perform input generalisation, but not require complex chaining of rules from one set of join groups to another. This section demonstrates chaining using a grid problem in which a number of bits in the input string may contain a 1 for each location in the grid. This task does not require input generalisation, but does require the discrimination of a number of positions in the grid using join groups.

The grid in Figure 5.11(a) represents a 4x4 aperiodic grid. The agent may move around the grid by taking one of four actions, Move-South, Move-North, Move-East and Move-West. Attempting to move into the grid boundary results in the agent's position remaining unchanged. The grid has one goal state at any one time which when reached by the agent results in a positive reward and ends the learning trial. Trials commence when the agent is placed in a start position which is selected with uniform random probability. All states other than the goal state have a zero reward.



(a) The agent in a start position and the two goals on the 4x4 grid.

10000	10100	10010	10110
01000	01100	01010	01110
11000	11100	11010	11110
00100	00010	00110	00001

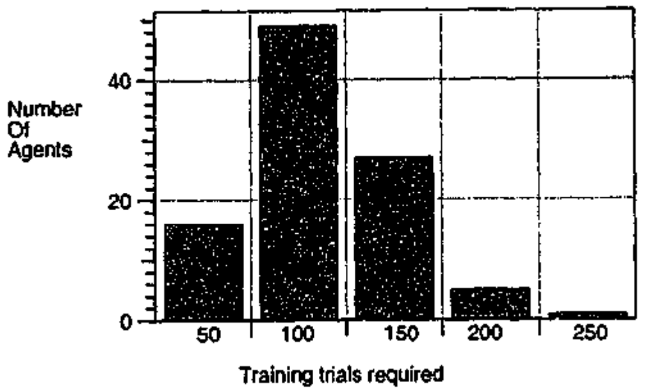
(b) The input strings for states on the grid.

Figure 5.11: The 4x4 grid problem.

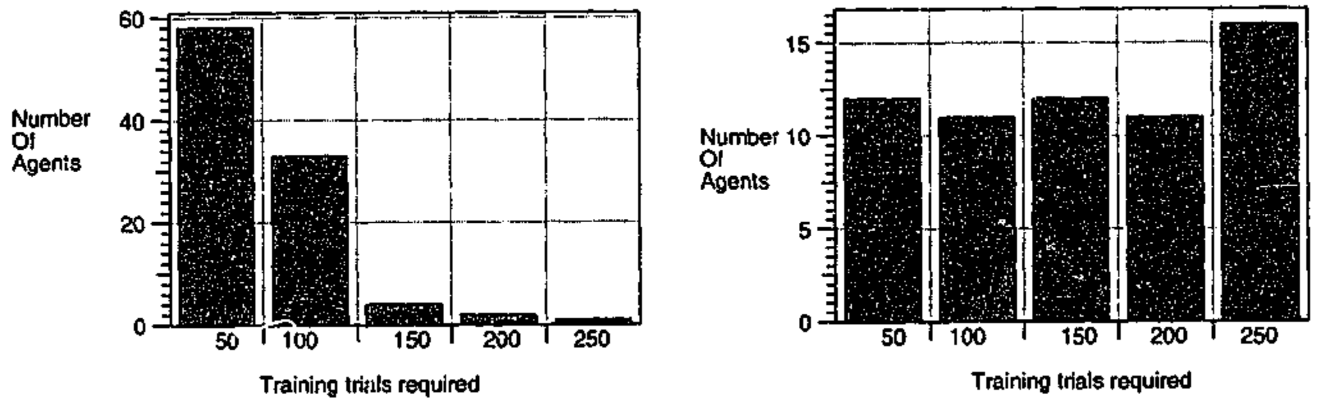
5.6.1 Experimental Design

In the initial experiments, the goal is located at the bottom right hand corner. In each trial, TRACA is placed at a random location on the grid (other than the goal location) and allowed to navigate to the goal. The learning rate is 0.2 and actions are selected probabilistically 1 in 3 times (using the roulette wheel approach) based on the *best supporter* method of action selection. The number of cases for the Cox-Stuart test and the test for noise was 10 and 20 respectively. Once the goal is reached or a maximum number of steps is taken (> 1000) the trial ends. Initially the agent is allowed a training episode of 50 trials. After which the agent is tested for 100 trials in the grid with random action selections turned off. If during any of the test trials the agent does not reach the goal within the maximum allowed moves the agent is given another training episode of 50 trials and tested again. This process is repeated up to 5 times. If during testing the goal is reached within the time limit on every trial, training is stopped and the agent is deemed to have learned the grid. The average number of moves to goal over the 100 test trials is used as an assessment of the agent's performance.

Once the agent has learned to successfully reach the first goal, the goal is moved to the top left hand corner. The agent then continues learning (using the network developed for the first goal) to reach the second goal. Again, the agent is given a series of 50 learning trials (up to 5 times) and testing is taken as the average of 100 trials once the second goal has been successfully learned. The top left corner was selected as the second goal because reaching it required more training (from scratch) than was required to reach the goal at the bottom right hand corner.



(a) Training trials required by 98 successful agents to learn the first goal.



(b) Training trials required by 98 successful agents to learn second goal extending the network learned for the first goal.

(c) Training trials required by the 62 successful agents to learn the second goal from scratch.

Figure 5.12: Results on the 4x4 grid.

5.6.2 Results on the Grid

On the first goal 98 successful agents recorded an average number of moves to goal of 3.21 with a standard deviation 0.14. This is very close to the average distance of 3.2. The number of training trials required for the agents is presented in Figure 5.12(a). Exactly half the successful agents (49) required 2 training episodes of 50 trials each. 27 agents required 3 training episodes, 16 required just 1, 5 required 4 and 1 required 5. The average number of unsuspended joins created was 15.1 with a standard deviation of 2.8.

On the second goal, training times were significantly reduced, with 58 of the agents learning the task on the first training episode of 50 trials. Another 33 of the agents required 2 training episodes, with only 4, 2 and 1 agents requiring training episodes of 3, 4 and 5 respectively. The average number of moves to the goal was slightly higher than for the first goal at 3.46 with a standard deviation of 0.29, a difference in performance of around 8 percent. The average number of unsuspended joins on completion of training was 18.2 with a standard deviation of 2.4. Another 100 agents were then trained on the second goal from scratch. Of these agents, 38 failed to learn to reach the goal within the allowed training time, training times for the remaining 62 successful agents are shown in Figure 5.12(c).

In summary, after being trained on one goal, the agents learned the second goal with more success and with fewer trials than were required to learn it from scratch. Furthermore, the second goal required only slightly more structure to be added to the network developed for the first goal (manual inspections of selected networks developed before and after the second goal indicated that most of the original structure is retained). The structures developed by TRACA for one task were successfully reused for another similar task. While this result is not unexpected it does suggest that the independence of vertices in TRACA's network avoids problems of catastrophic forgetting and interference that tend to affect many connectionist approaches (Fahlman 1988a; McCloskey and Cohen 1989; French 1999).

5.6.3 Adding Effector Noise

Another experiment was run with 25 agents, but with probability 0.1 the action selected by the agent would be changed to one of the four possible actions (selected with uniform random probability). Learning and test trials were conducted as they were above, except that for each goal agents were provided with up to 5 episodes of 200 trials of learning experience rather than 50. For the first goal in the grid the average number of steps to goal was 11.60 (standard deviation 10.64) with an average of 11.2 joins created (standard deviation 2.6). For the second goal, fewer steps were required to reach the goals and a number of additional joins were created. The average number of steps to the second goal was 3.88 (standard deviation 1.13) with an average of 20.6 joins (standard deviation 2.0). As in Section 5.6.2, the better performance on the second goal, when compared to the number of steps required to reach the first goal, appears to be due to the presence of the structures developed during the learning trials for the first goal. These provide a basis on which additional structure is added to achieve the gains on the second goal.

5.6.4 Problems with Rule Chaining

There is a problem with TRACA's rule chaining which adds difficulty to the learning of the 4x4 grid task and may prevent TRACA always taking the shortest path.

This problem is not unique to TRACA and is a by-product of its use of a default hierarchy representation, a similar problem also arises within Learning Classifier systems (Yates and Fairley 1993). The problem becomes apparent when we look at what structures are sending support for effectors in some states.

Take the example structures developed by TRACA for the 4x4 grid task presented in Figure 5.13. The nodes in groups G4 and G5 represent the states with input strings "10100" and "01010" respectively. These are two structures which ideally should be specific to these states. Ideally, G4 and G5 would respectively record the transitions and the associated values

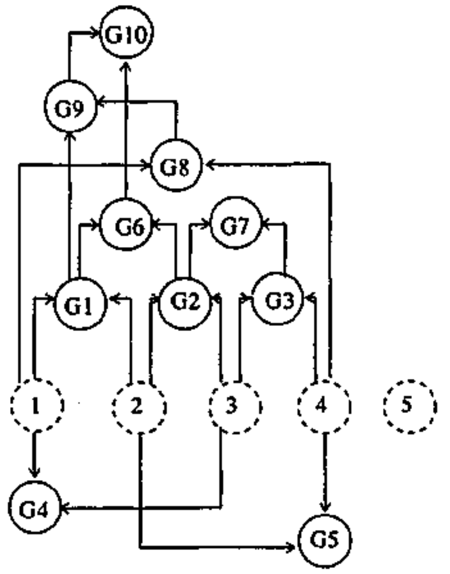


Figure 5.13: Sample of structures developed by TRACA for the 4x4 grid. Unary groups are indicated as broken circles and join groups as unbroken circles. The arrows indicate the subordinates of join groups.

for transitions for each of these states only. However, nodes in these two groups also update ETP's and values when in the state "11110" among others. Consequently, the value estimates of nodes in groups G4 and G5 are distorted and are not only inaccurate for the states "10100" and "01010", but may also introduce noise into the selection of effectors in other states where they are matched.

In many cases, TRACA can prevent this distortion by suppressing groups such as G4 and G5 when more specific groups are matched. For example, G10 is specific to state "11110" and suppressing all other groups in this situation would prevent those other groups from sending support for effectors when in the state "11110", and consequently from updating their dependent ETP's and value estimates on transitions from "11110".

The suppression mechanism indeed works on the subordinates of G10 (such as G8 and G1), it only fails on groups that are not subordinates – other groups in joins which subsume or are equivalent to other joins. Yates and Fairley (1993) define subsuming rules in relation to classifier systems as those rules which are implied as being active when the subsumed rule is active, but whose activation does not imply the subsumed rule is active. Classifier rules correspond to TRACA's nodes, and rule activation corresponds to nodes executing. Subsuming rules in classifier systems translate to nodes in more general join groups in TRACA which are not subordinates of more specific joins.

While in many cases superior joins can be created, it is possible that subsuming or equivalent rules will exist and interfere with the correct policy. Nodes in the subsuming groups may have higher ETP values than nodes in the suppressing superior, causing it to be removed. Furthermore, if the predicting node in the subsuming group makes reliable predictions, the

predicted group will not even create the new superior join group (if a threshold for dependent ETP's is used as described in Section 5.3.2).

However, join groups only discover and represent relationships between groups representing the current state. It is possible that by using information from temporally previous strings (states) we can incorporate additional context information to create temporal chains which overcome the problem of subsuming rules. Using this approach, the updating of ETPs and values for such temporal chains would be based not only on current state input, but on the input from temporally prior states also. Temporal chains are described in Chapter 6.

5.7 Hypothetical Look-ahead

In Section 5.6 a 4x4 grid was used to learn two goals. The problem with learning this grid was that once the agent had learned to get to the second goal, changing back to the first goal requires re-learning the policy for that goal over a number of training trials. In this section a similar 4x4 grid is used to demonstrate the use of hypothetical look-ahead to find paths to goals by propagating values back from the current goal without the need for actual trials to update the policy.

5.7.1 Experimental Design

The 4x4 grid for look-ahead is presented in Figure 5.14. Like the grid in Section 5.6 it is aperiodic and attempts to move into a boundary leave the agent's position unchanged. However, in this new grid the two goal states each have a single unique bit position which takes a value of 1 only when in the corresponding goal state. Furthermore, in a set of initial experiments it was found that the amount of training sufficient to reach the two goals as done in Section 5.6 (up to 250 trials for each goal) was not sufficient for successful look-ahead (due to the effects of subsuming rules which reduce as more structure is introduced).

000001	101000	100100	101100
010000	011000	010100	011100
110000	111000	110100	111100
001000	000100	001100	000010

Figure 5.14: The 4x4 grid for look-ahead and its input strings for each location.

In this experiment we are not so interested in the learning of the model as the associating of a policy with the model. Consequently, each agent is allowed a fixed period of 5000 timesteps for training in the grid which, based on the experiments in Section 5.6, should be more than sufficient for a good model to be discovered. During this training actions are selected with uniform random probability while the learning rate is 0.1. As no rewards are provided during training, and learning (other than policy learning) is turned off on completion of training, the order of the goals in this problem can be arbitrary.

Once training is complete, the *virtual utility* value (see Section 4.2.14) for the bit representing the location at the top left hand corner is set to 100. This is the first goal and the value set is the value propagated by hypothetical look-ahead. With this value set and learning turned off, the agent is given 100 trials in the environment. Each trial commences with the agent being placed at a (uniform) randomly selected location in the grid other than the current goal location. In the first timestep of the trial look-ahead is executed for 10 look-ahead cycles with each cycle extending across a distance of up to 10 consecutive states (which is more than adequate). No further look-ahead is undertaken during the remainder of the trial. In subsequent timesteps the agent must use the virtual utility values propagated from the initial look-ahead to navigate to the goal. The trial completes when the agent reaches the goal or if the agent does not reach the goal within 100 timesteps. If the agent does not reach the goal in the 100 timesteps, it is recorded as as failure. The number of timesteps for these trials is reduced from that in the experiments in Section 5.6 as the agent already knows the model and the aim of hypothetical look-ahead is to propagate values through the model which enable the agent to get to the current goal quickly. In light of this, 100 timesteps is very generous, and as it turns out far fewer timesteps are required in practice. After each trial the hypothetical values generated by the previous look-ahead process are reset to zero.

Once an agent has successfully completed the 100 trials, the goal is moved. The virtual utility of the first goal is reset to zero and the virtual utility of the second goal, in the bottom right hand corner, is set to 100. Another 100 trials are then conducted for the second goal in the same manner as for the first.

5.7.2 Results

25 agents were trialled on each of the goals. All were successful at reaching the first goal. One failed on the second goal and was excluded from the results for that goal. Of the successful agents the average number of steps to the first goal was 3.35 (standard deviation 0.3) and to the second goal 3.24 (standard deviation 0.19). The average number of unsuspended join groups was 25.3 (standard deviation 3.1). The fact that the average number of steps to goal is sightly higher than expected (3.20) is possibly due to inaccuracies in the dependent ETP's. Despite this, the use of look-ahead with an appropriate model successfully avoids the need for actual experience in the grid to propagate values back from goals when they are changed. Using hypothetical look-ahead in this way avoids the need to duplicate the state-space to allow learning two policies for the two tasks (by incorporating a

extra bit in the input string to indicate the current task) and therefore the need to learn a model for this larger state space.

5.8 Chapter Summary

This chapter has evaluated TRACA's learning abilities and described some areas in which it has difficulties. The experimental results indicate that TRACA can successfully learn in the presence of noise and can develop efficient generalised representations for problems with irrelevant attributes. However, TRACA has difficulty with classification problems where the usefulness of bits in the input string are not apparent in binary combinations (such as Monk 2). This is a problem shared by some other systems as described in Chapter 8. The experiments also demonstrated that parameters can be tuned to improve TRACA's performance (improve accuracy and reduce the amount of created structure). However, TRACA does appear to be robust to parameter settings and successful learning is possible without optimising parameters. Experiments with Monk 1 in which the exploration rate was varied indicate that even though TRACA does not make explicit use of utilities to reduce the amount of structure it creates, the use of rewards and penalties during training can lead to a smaller learned network.

Following the input generalisation experiments were experiments requiring rule chaining for distributed sensors. These experiments demonstrated two types of reuse. The first type of reuse is that subordinates of one group can be used as subordinates for a number of others. This type of reuse was also demonstrated by the sample structures presented for Monk 2. The second type of reuse is that structures learned by TRACA for one task could be successfully reused for another. This reuse lead to a reduction in the learning of the second task compared to learning conducted from scratch. The final experiment demonstrated how a model constructed by TRACA's input generalisation mechanism could be used to achieve multiple tasks by supporting multiple policies. In the final experiment a model was learned in the absence of reinforcement feedback. Context sensitive hypothetical look-ahead was then used to propagate virtual utility values throughout the model appropriate to the current task. This provides large potential savings in the size of learned world model and the training experience required to learn it. While TRACA was successful on both the simple chaining task and the look-ahead task, one drawback with TRACA is that the presence of subsuming rules can cause difficulties which may extend the time needed to learn the predictive model. This is a perceptual-aliasing problem affecting some internal structures, which, along with other sources of perceptual aliasing, may be addressed by TRACA's temporal structures described in the next chapter.

Chapter 6

Adding Temporal Chains

6.1 Representing Hidden-state

In the previous two chapters the problem addressed was input generalisation, which is necessary to efficiently represent useful world states and to allow learning to be applied to novel situations. However, learning with these generalised structures was still based on the assumption of the Markov property (see Section 2.3). This chapter addresses how TRACA uses temporal chains to tackle the problem of hidden-state which arises when, from the perspective of the agent, the Markov assumption does not hold. Using TRACA, hidden-state arises when the information necessary for a unique state identification is not present in the immediately available input vector. This problem is not unique to TRACA, nor is it unusual, since any agent's sensory apparatus can only ever provide limited information, usually local.

When using distributed sensors (see Section 2.5) hidden-state effects may occur even if the Markov assumption holds. This is possible if the internal state representation for states is constructed on an inappropriate selection of features so as to allow multiple world states to be mapped inappropriately to the same internal state. In this case, it is possible to correct the inappropriate mapping by making a better selection of features to base state identifications on. However, since it is often impractical to use all features in state identification, the selection of features poses a search problem in itself. In this respect, the input generalisation problem and hidden-state problem are intertwined; until a useful set of features is found, either historical or immediately available, our agent cannot represent the underlying Markov model. In many cases, the learning agent cannot be certain if a state is being incorrectly identified because some feature relevant to the distinction (and present in the input vector) is not being used in the state identification, or if more memory is required.

Finally, there is the complication of stochastic environments. An observation and action pair which repeatedly results in multiple different outcomes could be due to inadequate immediately available sensory information, basing state identification on an inappropriate set of features, or stochastic processes within the environment itself. This makes it difficult to

distinguish the presence of hidden state from a truly stochastic process in the underlying Markov model. If unconstrained, attempts to make this distinction (given partial observability) may result in a never ending search for an appropriate amount of memory (k) or an appropriate set of features.

In summary, the three primary difficulties identified in uncovering and representing hidden-state in large state spaces are:

1. Selecting correct features to identify states;
2. Incorporating memory when these features seem insufficient; and
3. Identifying stochastic processes where no set of features or amount of memory will be of assistance.

TRACA's input generalisation addresses the first of these difficulties, while TRACA's short-term memory mechanism, based on temporal chains, addresses the second two. The next section identifies different types of hidden state regions and some corresponding problems when using distributed sensors. Following this section is a high-level description of the search strategy used by TRACA to uncover hidden-state. This leads on to a description of the implementation of TRACA's search strategy which specifically looks at the three issues of when a search for a new useful temporal chain is started, how it progresses and how an unsuccessful search is identified and terminated. After this description several potential difficulties with TRACA's search strategy are identified along with the techniques developed to address them. Finally an example is presented which demonstrates in detail how TRACA builds temporal chains to represent solutions to problems with hidden-state. This example helps elucidate the algorithm used by TRACA, the major steps of which are listed once the example has been presented.

6.2 Hidden-state Region Types

There are two primary types of hidden-state region that may occur. Two terms, *homogeneous region* and *heterogeneous region*, are introduced to describe each of these regions. Each of these region types requires a different technique for creating temporal chains to represent them. Two example environments are used to exemplify these two different types of region. The example environments are both simple mazes with an east and a west branch. Within each maze, the two branches are identical except for the presence of a door at the end of each east branch. Each maze has boundaries which are impassable and locations within each are divided into discrete states with an input string (observation) representing each state's features. Possible features are the presence of windows, doors and surrounding walls. In both mazes, a value of 1 in each of the first four bits (in order from the left) is used to indicate walls to the north, south, east and west respectively. The value of 1 in the fifth bit indicates a door and the remaining two bits are used to indicate a window to the west and east respectively. An agent navigating in this simplistic environment may select an action which moves it from its current state to one of its neighbour states, unless the action leads the agent into a wall, in which case the agent's state will remain unchanged. Possible actions are Move-South, Move-North, Move-East and Move-West. Before selecting each action, the agent receives the input string which indicates the features of its current state.

The first maze is depicted in Figure 6.1. The branches of this maze provide an example of a heterogeneous region of hidden-state. For the east branch this region begins in the north-east corner. An agent moving south from that corner will experience the same observation until it arrives at the south-east corner. It contains hidden-state because each observation in the sequence of observations received while traversing the region is *aliased* (i.e shared) by some other state in the state space, in this case, by the states of the west branch. It is called heterogeneous because each state within the shared region of observations has a different input string (observation) due to the features (windows in this case) present in each state.

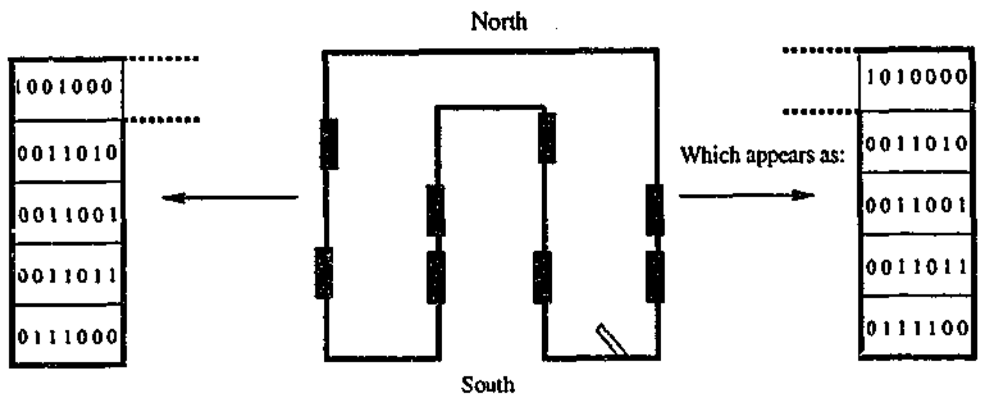


Figure 6.1: Hidden-state regions (corridors) with a heterogeneous input pattern.

The maze in Figure 6.2 illustrates the second type of hidden-state. This maze contains a homogeneous region of hidden-state. This region is demonstrated by the set of states traversed by an agent commencing at the north-east corner and continuing south until it reaches the south-east corner. Again, the region contains hidden-state because a sequence of observations received while traversing the region is shared by some other region in the state space, in this maze by the west branch. However, in a homogeneous region such as this, there is an additional dimension to the hidden-state. The same observation appears for multiple states within the hidden-state region.

This distinction between homogeneous regions and heterogeneous regions is critical, as homogeneous regions need to be treated differently during learning. These differences are discussed in detail in Section 6.5.1. However, there is one further problem with these two mazes; since distributed sensors are being used, each of these region types may appear to be a case of the other type. Whether or not this occurs depends on the individual features and combinations of features used to construct the agent's memory.

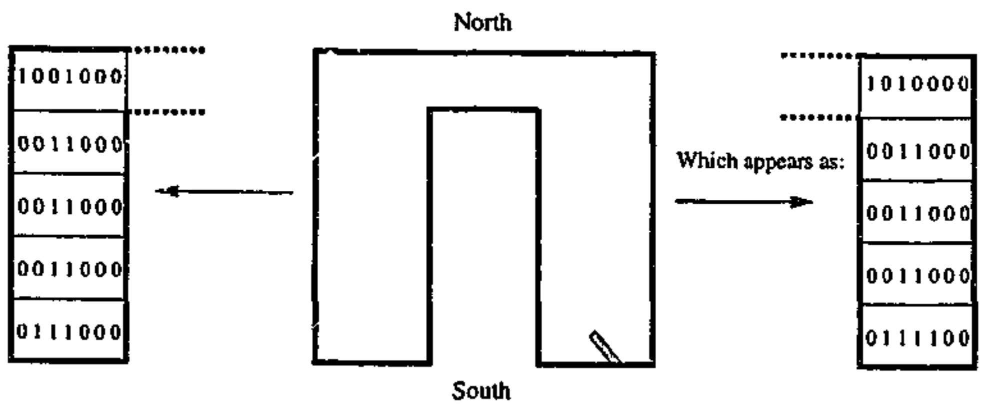


Figure 6.2: Hidden-state regions (corridors) with a homogeneous input pattern.

To avoid the complications associated with distributed sensors when constructing temporal chains the following sections which describe temporal chains focus only on problems which use a localised sensor scheme. Using a localised scheme each possible observation (set of features) is represented by a single unique bit in the agent's input string.

6.3 The Search Strategy

At a high-level TRACA represents memory for hidden-state problems using *temporal chains.* Each link in a temporal chain is a group containing a rule which extends across two time-steps. The temporal chain construction process consists of three tasks: *chain creation*, *chain extension* and *search restriction* (i.e chain removal). Temporal chain creation occurs when a condition arises which indicates the agent's environment may contain hidden-state. This condition arises when a group is incorrectly predicted by a node which is not support suppressed when it executes and the predicted group is not support suppressed in the following cycle.

Consider the example presented in Table 6.1 which represents a tape reading task with three different tapes. Each tape has three different letter positions which are the different states in this learning task. The agent is given a series of n trials on the three tapes with all tapes being used with equal probability. However, the agent receives no information on which tape it is reading during a trial other than the observations associated with the tape's sequence of letters. Each trial consists of three timesteps (t_1 to t_3) commencing with an initial state and observation at t_1. For this task, a localised sensor scheme is used in which each observation has a unique bit in the input string. At each timestep the agent executes the read action (its only possible action for this task) after which it moves one position along the tape, changing state and receiving a new observation. This process is repeated until the end of the tape is reached (at t_3) at which point the trial ends.

The tape reading task is used in the following subsections to examine TRACA's processes for constructing temporal chains with the result that its internal model correctly represents the true Markov model underlying the observations it receives. The tape reading task is therefore unlike most other tasks in this chapter and the next, where our interest in temporal chains is not only in their ability to correctly represent transitions in the agent's environment, but also in their ability to improve TRACA's action selection policy.

Tapes 1, 2 and 3 start with the letters D, E and F respectively. All three have A as the second letter, however, tapes 2 and 3 have a Y as the final letter in their sequence, while tape 1 has an X as its final letter. This difference in the final letter of one of the sequences, combined with the hidden-state problem resulting from the shared letter in the middle of each sequence, makes correct representation of the transitions in this task impossible when using just unary groups and nodes (which are created by TRACA automatically for any task). To correctly represent the transitions in this tape reading task short-term memory is necessary, which in TRACA's case requires the creation of temporal chains.

In general, chains (once created) are not formed instantaneously in their entirety, but are developed incrementally over a number of trials as additional memory is added. This process of *chain extension* adds prior observations and action pairs to the temporal chain until the chain makes reliable predictions (chain extension and measures of reliability are discussed in more detail in Section 6.4.2). Once the process of chain extension is finished for an individual chain, the chain is refered to as *final*.

In reality it is possible that no amount of history will improve the predictions of the temporal chain. This problem may arise if there are stochastic processes in the environment. For example, if B represents standing on the sidewalk and A the passing of a car, no amount of history (within the percepts of the agent) will help predict A. It is in situations like this that search restriction is necessary. One method of restricting search (already mentioned in Chapter 2) is to maintain a fixed sized history window the pre-determined size of which prevents endless searches. However, if the window size is too small, necessary hidden-state will not be uncovered. Similarly, if it is too large, resources may be wasted (this was discussed in Section 2.10.1). TRACA uses an approach which allows the maximum length of temporal chains (i.e the window size) to vary in different areas of the search space. This sizing of this window is part of TRACA's process of *search restriction* which is based upon estimates of the benefits of constructing a temporal chain to represent a particular aliased region in the environment. These benefits are calculated using the estimated returns associated with the internal state predicted by the temporal chain under construction which is discussed further in Section 6.4.3.

Regardless of the size of the history window, there are multiple possible paths that can be followed when trying to uncover hidden-state. The following section describes how the search for an appropriate path is conducted in the tape reading task.

	t_1	t_2	t_3
Tape 1 Observation sequence	D	A	X
Tape 2 Observation sequence	E	A	Y
Tape 3 Observation sequence	F	A	Y

Table 6.1: Observations in order of appearance for each of the tape sequences. The first observation in a trial is one of the letters D, E or F all of which are followed by A and then either X or Y, which can be predicted by remembering the first letter of each sequence.

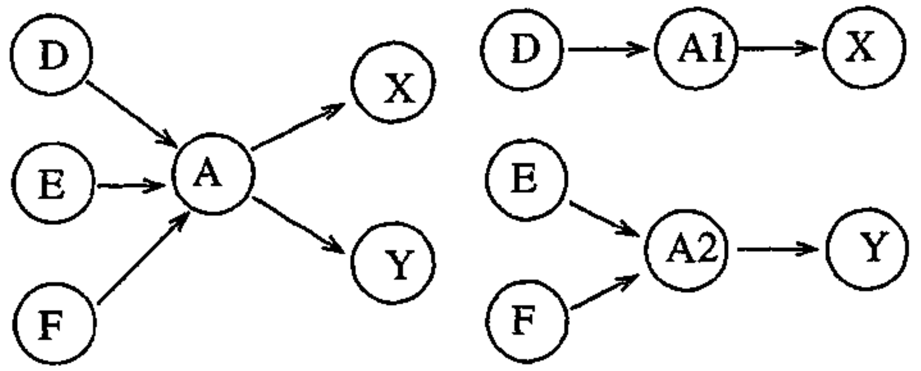
6.3.1 Problem Search Space

The search domain of any hidden-state task may be viewed as a directed graph of states and actions. Given a particular state, the graph forms a tree, the root is the state observation to be predicted with vertices corresponding to possible prior state observations and edges corresponding to actions leading from one state to another. Each branch in the tree represents paths from the state represented by the leaf to the state represented by the root, or alternatively, paths from the branch to the root.

The size of such a tree may increase exponentially as it is expanded down all possible paths. Each sequence of observations and actions leading to the root are paths in this *search tree* structure. The particular state we are constructing a temporal chain to predict is the *chain prediction* and the tree root. During temporal chain extension, the temporal chain created so far represents a current *search path*, which continues to be extended in depth down the tree until either the hidden-state is uncovered, in which case the search is *complete*, or the search is terminated by restrictions on search size. TRACA's algorithm for creating temporal chains involves a search through this space of observation and action sequences. For each search tree, the group representing the tree root is the *chain prediction*.

A search path is a path which extends from the root down one branch of the tree towards a leaf. However, during trials paths are followed in the opposite direction; from a leaf to the root. These paths are called *execution paths*. Accordingly, each temporal chain has its own execution path which is the sequence of observations and actions experienced while following the chain. The group encountered at the start of a chain's execution path is called the *first group* while the group encountered at the end of a chain's execution path, just before the chain prediction, is the *terminal group*. Nodes within the terminal group are *terminal nodes* and the terminal node which predicts the temporal chain prediction is the chain's *primary node*.

When a domain contains hidden-state, the search tree includes paths which do not form valid execution paths in the graph to the root. This is because the graphs (or sub-graphs) appear connected until shared vertices are distinguished as separate environment states. Our example presented in Table 6.1 appears to be a connected graph (Figure 6.3(a)), whereas it is in fact a disconnected graph (Figure 6.3(b)).



(a) A connected graph representation of the domain presented in Table 6.1.

(b) Once hidden state is distinguished the connected graph in (a) appears as a disconnected graph. In this case, the vertex A has been split into two vertices A1 and A2 in two separate graphs.

Figure 6.3: Two graphs before and after hidden-state is uncovered.

The shared vertex is not distinguished as we are searching our tree, so search paths can be formed representing execution paths (walks) which can never actually occur in the problem domain. Figure 6.4 shows the valid and invalid paths that can be formed in a tree structure which has been expanded from the root node down, it also shows the directions of search and of chain execution. The temporal chain discovered for the valid path would have a *first group* corresponding to the observation D and a *terminal group* corresponding to the observation A. The *chain prediction* for this chain is the unary group representing the observation X.

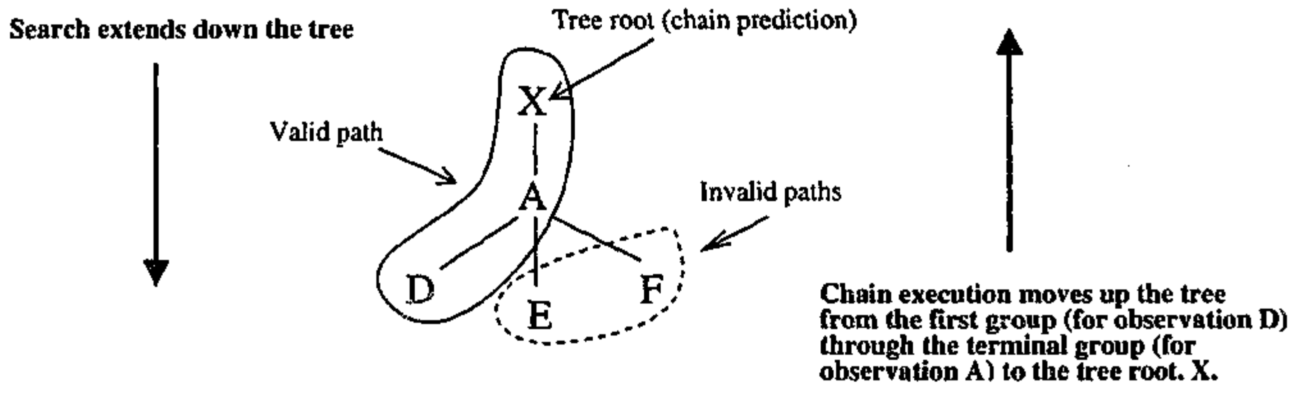

Search extends down the tree
Tree root (chain prediction)
Valid path
X
A
Invalid paths
D
E
F
Chain execution moves up the tree from the first group (for observation D) through the terminal group (for observation A) to the tree root. X.

Figure 6.4: Valid and invalid search spaces in the solution tree for X in the problem from Table 6.1.

6.3.2 Implementing the Search Strategy

TRACA's search strategy aims to discover a valid search path which leads to the root and it is perhaps best described as a depth-limited iterative depth-first search. That is to say, that starting from the root, the search will extend down an arbitrary branch to the leaf. If an invalid path is followed, the most recent extension is removed and another created down a different branch. The depth is limited by the value of the chain prediction and if the search extends too far, the entire path created so far is removed and another commenced. Note, that because the search depends on the execution path being followed, which in turn depends on the order of the trial sequences which are typically random, it is unlikely that the branches of the tree will be explored in any systematic order.[1]

Searches are conducted during actual trials, where each trial provides an opportunity for at least one extension for each temporal chain (searches may execute concurrently for different temporal chains). As such, temporal chain creation requires a number of walks (and possibly a variety of walks) being taken through our graph. In each walk, once the first group of an incomplete chain is reached, the chain can be extended to the most recently visited vertex. This is equivalent to extending it down one branch of the search tree and in doing so this new extension becomes the new first group of the chain. If we continue up this new chain execution path and find that we do not follow a valid path to the root, the path has been extended to form an invalid path and needs to be back-tracked so the extension is removed. In fact, successful retention of an extension requires that during the walk in which the extension was added the chain's execution path is followed. If the execution path is deviated from the system cannot determine if the execution path leads to the root and the extension will be discarded (details of this process are presented in Section 6.4).

In situations where the most recent extension is discarded, we must wait for another trial during which the solution path can be extended again, hopefully, (due to exploratory actions) to a sibling node of the removed node which forms a valid execution path. This is the *iterative* component and it is not systematic, no record is kept of which paths and siblings have been tried.[2] Given this, if a valid extension is found we proceed in a depth-first manner but, due to the randomness of the agent's experience, there is no guarantee that any particular branch will be followed first. The maximum depth (execution path length) a chain will extend to is restricted based on the reinforcement associated with the state corresponding to the root (this is discussed in more detail in Section 6.4.3).

6.3.3 A Possible Alternative Approach

There are a variety of ways of addressing the hidden-state problem. This section justifies the design decisions taken by considering a possible alternative design.

---

[1] This prevents the use of a systematic search method such as depth-first iterative deepening (Korf 1987).

[2] In fact in many cases it maybe desirable to select for extension at any time the group which most frequently precedes the subordinate group of the current first node in the temporal chain (see Section 6.5.3).

If our root is X the creation of an execution path to include either E or F in Figure 6.4 would be invalid since with our single read action, there is no walk from E or F which leads to X. However, these paths are valid execution paths for tree with a root of Y which does help solve our overall problem. This ability of a temporal chain's execution path to be invalid for one tree (or prediction) but valid for another, leaves open the possibility of an alternative approach to the one implemented in TRACA. This alternative involves simply extending down any search path and then determining which root a chain ends up predicting (i.e which tree the search belongs to). However, this alternative has two major drawbacks. The first is the necessity to maintain nodes in our terminal group for each possible root group (in this case, simply X and Y, but in more complex environments the number of structures can be quite large) only to have to remove (or redundantly retain) the nodes which predict groups which never occur at the end of the chain's execution path because they belong to another tree. The second drawback is that since we do not know what the final root will be during chain extension, we can only construct one temporal chain at a time or it is possible that duplicate, redundant temporal chains may form. This restriction of only extending one temporal chain at a time for the set of trees in regions with hidden-state is expensive in terms of the number of trials required in the environment which could be used to create multiple temporal chains for the regions simultaneously.

In comparison, the approach taken by TRACA of building temporal chains for a specific root allows the simultaneous creation of multiple temporal chains, minimising unnecessary internal structure (i.e nodes) while maximising the benefits of actual experience in the real environment.

6.3.4 Parallel Rule Operation

TRACA's design for temporal structures takes advantage of TRACA's parallel operation to address a number of issues. The first is the large number of nodes required if every link group in each temporal chain contains nodes for every possible prediction and action available from the state the link group represents. The possibility that some hidden-state regions may require multiple temporal chains for different action sequences only exacerbates this problem.

The second is the amount of experience (i.e training time) required. If each link group in a temporal chain was to act in place of its subordinate group, the nodes in these links must be given a number of initial trials before they can be used to send support. Consequently, additional experience may be necessary with the link group to re-learn transitions and reinforcement values that are already represented in the subordinate groups. An example of this is a temporal chain which extends down the length of a corridor. During early trials nodes in unary groups in the corridor will have learned that to move into walls results in a collision and a penalty. To re-learn this for each of the link groups requires additional experience and exploration, taking actions (and incurring the associated penalties) to repeat that learning for the link groups. TRACA's parallel rule execution avoids this need for

re-learning by allowing unary groups to send support to effectors in conjunction with their superior link groups thereby directing TRACA's behaviour away from such collisions.

An exception to this are the subordinates of terminal groups, these are support suppressed by their superior terminal group. However, in this case the superior terminal group may contain multiple nodes to predict the multiple possible groups that may occur simultaneously with the chain prediction at the end of the chain's execution path when using a distributed sensor scheme. This allows the terminal group to represent multiple possible predictions at the end of a chain's execution path. In addition to this, once a chain is final, the terminal group may contain nodes which support actions (effectors) other that supported by the primary node. This allows a single chain to predict different outcomes for different actions at the end of a hidden-state region. Effectively the chain now represents multiple execution paths, however, each additional path is identical up until the terminal group.

6.4 Implementing the Approach

Here we look at the three tasks of temporal chain creation, temporal chain extension and search restriction in more detail using another tape reading problem. Like the problem in Section 6.3 there are three tapes each with a sequence of letters which can be read in order. However, in this new task there is an additional letter in each sequence. As before, the initial observations for tapes 1, 2 and 3 are D, E and F respectively, followed by the observation B and then by the added observation of A. For tape 1 (which has an initial observation of D) the final observation is X, and for tapes 2 and 3 (with initial observations of E and F) it is Y. These three tapes are shown in Table 6.2.

	t_1	t_2	t_3	t_4
Tape 1 Observation sequence	D	B	A	X
Tape 2 Observation sequence	E	B	A	Y
Tape 3 Observation sequence	F	B	A	Y

Table 6.2: The sequences of observations for the three new tapes. In each trial, the system starts at t_1 and then moves through t_2 and and t_3 with each read action. The final observation at t_4 can be predicted by remembering the initial observation.

For each of these observations there is a unary group containing nodes which record the transition histories (ETPs) to other observations based on counts of the relative frequencies of the different transitions. After a number of trials the ETPs will be approximately 1.0 for the transitions from D, E and F to B, 1.0 from B to A, 0.333 from A to X and 0.667 from A to Y (see Table 6.3).

t_1	t_2	t_3	t_4
D (1.0)	B (1.0)	A (0.333)	X
E (1.0)	B (1.0)	A (0.667)	Y
F (1.0)	B (1.0)	A (0.667)	Y

Table 6.3: The approximate transition histories (ETPs) calculated by unary nodes (in groups representing each observation) to subsequent observations. As X occurs only a third of the time A has a value of 0.333, and as Y occurs two thirds of the time it has a value of 0.667.

6.4.1 Temporal Chain Creation

Temporal chain creation is triggered by the ETP values maintained by nodes in unary and join groups. In our example, these conditions are demonstrated by the scenario where the system has moved through the sequence with the initial observation D passing B and A and has just received observation X:

Chain creation rule:

If A's unary group is unsuspended, not support suppressed and the dependent ETP from A to X given the read action is less than 1 (it is 0.333) **then** create a temporal chain for predicting X given the re-occurrence of the sequence of observations and actions experienced since the observation which occurred prior to A (i.e the sequence experienced from B to X inclusive).

This rule creates a chain, however, the chain is not yet sufficient to correctly predict X and must be extended in a later trial in which it again experiences the sequence of observations and actions leading from D to X. ETP's are calculated as they are in the basic system (see Section 4.2.7). Dependent ETP's are calculated as follows:

$$e_{k+1} \leftarrow e_k + \alpha\,[r - e_k] \tag{6.1}$$

This rule is applied each time a node executes and its group is not support suppressed by a superior group. α is an update rate (typically the learning rate) and r is 1 if the predicted group was matched in the timestep after executing and 0 if not. As for the creation of join groups (see Section 4.2.2), groups which are predicted by a node with a dependent ETP above a threshold can be considered reliably predicted. These groups do not create new chains.

In general chains are represented as: $< O_1, a_1 >, < O_2, a_2 > ... < O_n, a_n > \rightarrow O_{n+1}$, where O_n is the observation at timestep n and a_n is the action taken at timestep n. Each of the bracket expressions <> is one *link* in the chain and each link is implemented by a temporal group. The commas between each of the bracketed expressions indicate the sequence of observations and actions (i.e links) in the chain and the → indicates the chain prediction.

As a result of this rule the temporal chain $< B, r >, < A, r > \rightarrow X$ is created, where r is the agent's *read* action. Internally to TRACA, a temporal chain's prediction is one of: a unary group, a join group or the first group of a final temporal chain; all of which can create a chain if they are not support suppressed by a superior group in the timestep in which they predicted. Temporal groups contain a single node called the *link node*, whose associated action is the *link action*. For terminal groups the primary node is the *link node* and its action is the link action. A link node is matched at timestep t if at timestep t the chain execution path has been followed up to the link node. The link node *executes* if it is matched at timestep t and at that time its supported action is selected as the system action. A temporal chain *executes* each time the complete chain execution path is followed. Once a chain executes, the primary terminal node will update an ETP to the chain prediction called the *chain ETP*. Each chain link will also maintain an ETP for the next link in the chain, given that the chain's execution path has been followed up to that link. Note that the unary group representing A is now a subordinate of the temporal chain's terminal group (similar to subordinates of join groups) and will be support suppressed each time the chain's execution path is followed up to the terminal group, at which point the chain is *matched*. Each time a chain is matched, the terminal group sends a *create suppress* and a *support suppress* message to its subordinate group (see Section 4.2).[3] Note, that for heterogeneous temporal chains terminal nodes can be created for other predictions before the temporal chain is completed, but only when the chain's execution path has been followed and the primary node's prediction is correct.

6.4.2 Temporal Chain Extension

The newly formed temporal chain $< B, r >, < A, r > \rightarrow X$ is now allowed a number of trials to determine its ETP. In our example the transition history will approach 0.333, and because this is less than a threshold value (typically close to 1.0) the temporal chain needs to be extended. Assuming we started in D and have moved through the sequence to B then the temporal chain can be extended using the following rule:

Chain extension rule:

If the first link in the temporal chain is matched given the current observation of the system **then** extend the chain using the previous observation for the new link.[4]

The temporal chain is now of the form: $< D, r >, < B, r >, < A, r > \rightarrow X$. This chain is sufficient to solve the problem of correctly predicting X, and the creation of a similar temporal chain can be used to predict Y.

---

[3] Temporal groups do not send these messages to their subordinates.

[4] This assumes a localised sensor scheme, under a distributed scheme, a group is selected randomly for the extension from those matched and not support suppressed in the previous timestep.

However, if the sequence of states in which we extended the temporal chain started with E rather than D, the chain would represent $< E, r >, < B, r >, < A, r > \rightarrow X$, a sequence which would never occur. Therefore each time a temporal chain is extended, if the sequence (walk) in which it was extended does not pass the temporal chain's prediction, the most recent link is discarded.[5]

6.4.3 Search Restriction

Search restriction is concerned with the removal of chains. There are two main situations in which created temporal chains should be removed. The first situation is to prevent duplicates in complex environments. The second is when the cost of maintaining and extending the temporal chain outweighs the benefits. This could occur if the state we are trying to predict has low utility, or if transitions to the state we are trying to predict are truly probabilistic, in which case building temporal chains to try and improve predictions is futile.

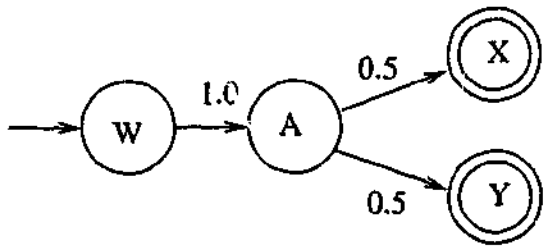


Figure 6.5: A probabilistic finite-state automata where the transition from A to X or Y is entirely stochastic.

The environment depicted in Figure 6.5 is an environment where there is a probabilistic transition, given a single possible action, a, and memory of the initial state does not assist in predicting the end state. In this case, we want to prevent a temporal chain such as $< W, a >, < A, a > \rightarrow X$ being retained in the system. One way of ensuring this is to require that one or more terminal nodes provide an improvement over their equivalent subordinate in order to be retained. This is done based on the ETPs of the nodes in the terminal group (except for homogeneous chains as is explained shortly).

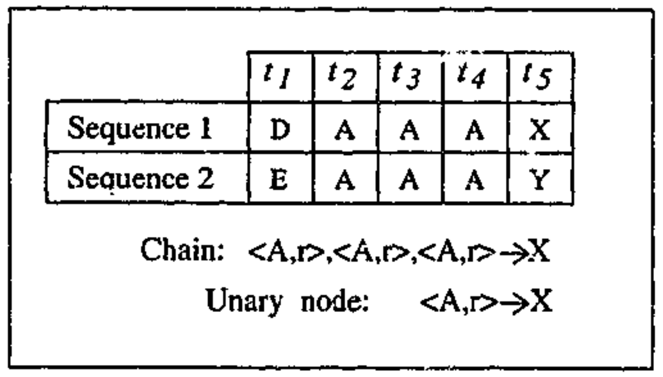
Like ETP values in other nodes, ETP values in terminal nodes can oscillate. So the test for noise (see Section 4.2.11) is used to compare the value of each terminal node with their equivalent subordinate node (the node in their immediately subordinate group with the same action and prediction). This test is the same one as used for unary nodes described in Section 4.2.9. However, each time a chain is extended, both the Cox-Stuart test and the test for for noise are re-initialised.

---

[5] In fact for tasks with homogeneous hidden-state regions this is more complex. A chain sequence may not have its prediction matched after each extension, this is discussed in detail in Section 6.2.

Chain extension is triggered by another set of tests. Once a terminal node has a higher ETP value than its equivalent subordinate, cases are collected for the test for noise and when sufficient cases are available the comparison is made. If the terminal node's ETP is not higher than its equivalent subordinate and the Cox-Stuart test for trend indicates no upward trend, the chain is extended. Should a terminal node demonstrate an improvement over its equivalent subordinate (using the test in Section 4.2.9), the temporal chain will be set to *final*. No further extensions are made to a final temporal chain, and the nodes in the chain (temporal and terminal) are eligible to send support to effectors as appropriate (when matched). The first group in a final chain may also create predictors for itself in other groups when its subordinate is matched.

Temporal chains representing homogeneous regions of hidden-state (homogeneous regions were introduced in Section 6.2) present a different problem. These chains will demonstrate an improvement over their primary terminal node's subordinate just by being extended. This occurs because the specificity of the temporal chain increases with each extension, even though the temporal chain may not have sufficient length to prevent any perceptual aliasing (see Figure 6.6). In these cases, the temporal chain is retained if its primary terminal node's ETP exceeds the ETP value of its equivalent subordinate and a threshold value (usually close to 1.0) called the *temporal threshold* (TT).[6] ETP comparisons used in statistical tests can be biased to favour retaining chains and joins by increasing node ETPs by a small factor using an *improvement factor* (*IF*) up to a maximum of 1.0 (see Section 4.2.11). Nodes whose ETP's exceed their subordinates ETP's apply the *IF*, to their ETP value (but not their subordinates) when recording it for use in the test for noise. In the experiments in Chapter 7 the *IF* is applied by nodes in terminal and join groups.



	t_1	t_2	t_3	t_4	t_5
Sequence 1	D	A	A	A	X
Sequence 2	E	A	A	A	Y

Chain: <A,r>,<A,r>,<A,r>->X
Unary node: <A,r>->X

Figure 6.6: In this figure, the temporal chain does not help solve the letter prediction problem because it has not been extended to include D. However, the primary node in this chain will appear a better predictor than its equivalent subordinate node because it makes an incorrect prediction of X less often ($\frac{1}{2}$ versus $\frac{1}{6}$).

Now that we can determine when a temporal chain should be retained, the remaining problem is to determine how many links should be added while trialling an unfinished

---

[6] The use of threshold values such as this is less than ideal, as their values will depend on the environment.

temporal chain. It is possible that the sequence of state transitions which the temporal chain must extend across before being retained is so long that it does not provide any benefit to the system. For example, building a temporal chain to represent a long corridor may be worthwhile if the reward for traversing that corridor is high, however, the effort may be wasted if the reward is low and the corridor is perhaps better avoided.[7] Similarly, we do not want to waste resources attempting to create temporal chains to deterministically predict states for which transitions are unavoidably probabilistic in the underlying Markov model.

The extension of temporal chains is restricted by the returns associated with the predictions of terminal nodes. Each predicted group maintains an estimate of its average return and the average immediate reward received when it is matched. These are recency weighted moving averages which are calculated as follows:

$$a_{k+1} \leftarrow a_k + \alpha[v - a_k] \tag{6.2}$$

where a is the average being calculated and v is the latest figure to be included in the average and α the update rate. The maximum of these values is the terminal value t which is stored in the terminal group and is calculated as:

$$t = R + \gamma r \tag{6.3}$$

where R is the immediate reward, r the average return and γ the discount factor.

As such, the terminal value may be zero, positive or negative. This value is passed back by the terminal group along the temporal chain and is discounted at each link. If it is a positive value and it falls below a positive threshold at any link in the temporal chain, and the temporal chain is not final, the temporal chain will be removed. If it is a negative value, and it rises above a negative threshold value, the temporal chain will also be removed. In the current implementation of TRACA, the setting of the negative and positive threshold values requires a-priori knowledge of the reward landscape.

The next restriction is necessary to prevent duplicates. This restriction is different for final and non-final chains. Final chains which are matched prevent new chains being created in the next timestep (this is because final chains can represent multiple actions as is explained shortly). Non-final chains prevent chains being created in the timestep in which their execution path was followed to completion and their prediction is currently matched (the fact that the chain prediction must be matched allows other chains to be formed concurrently for other predictions). Broadcast messages are used in these two cases to prevent the creation of chains by other groups. However, for problems which contain homogeneous hidden-state regions, these broadcast messages are insufficient. In the case of homogeneous chains, it is also necessary to prevent temporal chains forming if a chain's execution path is being followed and the chain's prediction is passed before all the chain's link groups have been matched in sequence. Another broadcast message is used to achieve this. The additional

---

[7] This is a simplification, we may also want to (and can) represent temporal chains which predict large penalties so we can avoid dangerous situations.

difficulties that arise in relation to creating chains for homogeneous regions of hidden-state are discussed in detail in Section 6.5.1.

However, all these controls for preventing duplicates may fail if the environment is non-deterministic or if a chain's execution path is not the path consistently followed to reach the chain's prediction. This is likely to occur when exploratory actions are being taken. Since this is extremely difficult to detect and avoid, temporal chains created in these situations are initially allowed to form. In fact, they may validly end up representing an alternate path to the original chain's prediction. In these cases, duplicates are prevented as each chain is set to *final*. Chains are only set to final when they execute and their prediction is matched. When a group is predicted by a node in an existing final chain, a broadcast message is sent to all groups indicating that any other chains which executed and are due to be set to final should be removed.[8]

A further efficiency can be gained by re-using a single chain to predict outcomes from the different actions possible at the end of a hidden-state region. Rather than create a separate chain across the region for each possible outcome, once a chain is final additional nodes can be created in the terminal group to predict the results of actions other than the primary node's action.[9] Because there may be multiple groups following the matching of a completed chain, the number of chains created in each timestep is limited to one, since that single chain can be used to predict the multiple possible outcomes.

6.4.4 Summary of Chain Operation

Chains are created when a group is unreliably predicted by a node in a unary or join group. Each new chain is created with two initial link groups (the terminal group and one other). The terminal group is created as a superior of the unary group containing the unary node which was making unreliable predictions. Since the primary terminal node has the same action and prediction as the unary node, the unary node is the terminal node's equivalent subordinate. The chain is then extended until the primary terminal node reliably predicts its prediction at which point it is set to final. Reliability is determined differently for heterogeneous and homogeneous chains. A heterogeneous chain will be set to final if one of its terminal nodes improves on its equivalent subordinate (determined by the same tests as used for join nodes). A homogeneous chain is set to final if the primary terminal node's ETP exceeds a threshold value. Heterogeneous chains may be removed if the nodes in the terminal group do not continue to provide an improvement over their equivalent subordinates. Similarly, homogeneous chains may be removed if their value drops below the threshold.[10] Chains may be extended until they are final, and before each extension the chain path must

---

[8] This is really all that is necessary to prevent duplicate chains, but on its own it is not very efficient.

[9] Not only is it more efficient to leave the creation of these nodes until a chain is final, it is also difficult to determine which groups occur when a chain executes until a chain is final and extension is complete.

[10] The test for homogeneous chains is somewhat inadequate as in non-deterministic environments runs of events may cause short-term fluctuations in ETP values. The Cox-Stuart test and the test for noise could be used here to more reliably determine when these chains should be retained and moved.

be followed a number of times to determine whether the current extension is sufficient to make reliable predictions. Other nodes may be created in the terminal group to predict groups that occur (deterministically or non-deterministically) in addition to the chain prediction when the current chain execution path is followed to completion. Duplicate chains are prevented using two mechanisms. The first mechanism is the use of broadcast messages sent when final chains are matched or unfinal chains execute. The second mechanism removes chains when they are set to final if an equivalent chain already exists.

Figure 6.7 presents the chain $< L1, a >, < L2, a >, < L3, a >, < L4, a > \rightarrow P1$ which has 4 links (including the terminal link). Each link, other than the terminal link, contains a single node for the action associated with the chain's execution path at that point. The terminal group contains three nodes labelled 1, 2 and 3 which are all matched once the chain is matched, this occurs at t_4 in Figure 6.7. The chain's primary node is labelled 1 and it predicts the group P1 given that the action y is selected (this corresponds to the chain's execution path). Node 2 predicts the group P2 given the selection of action x while node 3 predicts group P3 given the action z. Nodes 2 and 3 could only have been created after the chain was set to *final*.[11] Figure 6.7 includes long broken dividing lines which indicate a sequence of timesteps during which the chain's execution path is followed. Chain execution commences at t_1 and completes at t_5 with the matching of the primary node's predicted group, P1. Within each timestep the *match* and *fire* steps of TRACA's major cycle are shown (see Section 4.2) using short broken dividing lines. The third step, *execute* is not labelled, but occurs for each unary and join group node which fired and whose supported effector was selected at t_n and whose prediction was matched at t_{n+1}. A chain is *matched* once its execution path is followed up to the terminal link. Figure 6.7 indicates the point at which the chain is matched. It is at this point that a broadcast message is sent which prevents groups matched at t_5 from creating new chains.

6.5 Difficulties in Building Temporal Chains

This section describes in detail some of the difficulties in building temporal chains. It includes a discussion of the necessity of including actions in temporal chains and the selection of paths for extending chains. This is particularly important when there is the possibility of irrelevant attributes. Unless subordinate groups for extensions are selected carefully, extensions of chains may be made on groups representing the irrelevant attributes and the extension will either need to be removed or another chain created. The use of an additional statistic is described which can be used to avoid this problem.

---

[11] Having multiple nodes in a terminal group takes advantage of the fact that Q-learning will return the maximum value of all nodes in a group.

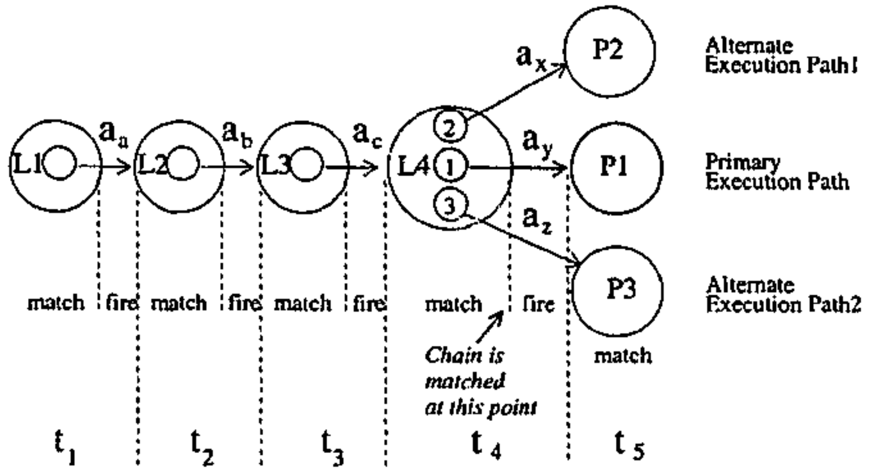


Figure 6.7: A chain with three nodes (labelled 1, 2 and 3) in the terminal link group (labelled L4) for the three different actions (labelled x, y and z) which have been taken after the chain was matched during different trials. The timesteps shown as t_1 to t_5 are each divided to indicate the *match* and *fire* steps of TRACA's major-cycle.

6.5.1 Representing Homogeneous Regions

There are two major problems associated with constructing Markov-k chains to represent homogeneous regions of hidden-state.

The first problem is detecting when the chain provides an improvement over existing structures. In the case of chains representing heterogeneous regions this can be done by comparing the terminal node's ETP to the ETP of the node in its subordinate group which has the same action and predicts the chain prediction (the equivalent subordinate). If, after a suitable number of trials, the terminal node provides an improvement over the subordinate the chain can be retained, otherwise it should be removed.

However, for a chain representing a homogeneous region, real improvements cannot be detected so easily. This is because the greater specificity of a chain extending across a homogeneous region can make it appear more useful than it really is. The terminal node in any chain which extends over two or more observations will have a higher ETP than a subordinate node whose group includes only one observation. As mentioned in the previous section, TRACA's solution in the case of homogeneous chains is to require that the ETP of the primary terminal node exceed a threshold value.

The second problem is due to the process of chain extension. Chains which are due to be extended add a link to include additional history when an execution path is followed to the first group. The extension is then retained only if the agent continues to traverse the chain's path and that traversal leads to the chain prediction.

For heterogeneous regions and for homogeneous regions extending over only two aliased states this presents few problems. However, for homogeneous regions extending over more than two aliased states there are additional complexities which only arise during the extension of chains (and not once a chain is correctly completed). The problem is that the search-path of partially constructed chains may match multiple different sub-regions of the aliased region it is being build to represent. In this scenario it is possible that a partially constructed chain never leads to the chain prediction when its path is followed.

	t_1	t_2	t_3	t_4	t_5
Tape 1 Observation sequence	D	A	A	A	X
Tape 2 Observation sequence	E	A	A	A	Y

Table 6.4: A tape reading problem with two possible sequences of observations containing homogeneous hidden-state.

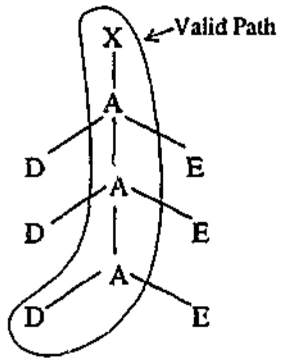
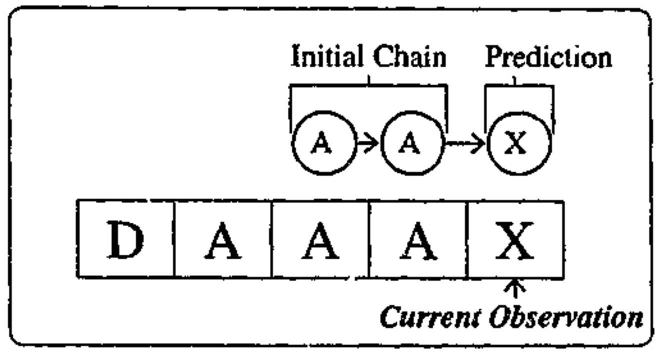


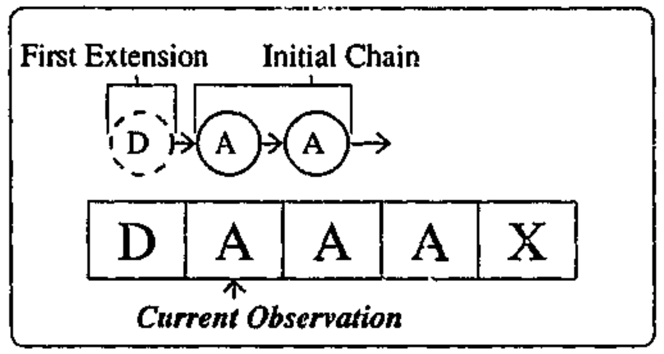
Figure 6.8: The search tree for the observation X in Table 6.4 showing the path required to be searched to construct a temporal chain to reliably predict X.

To demonstrate this point, consider the tape reading problem presented in Table 6.4. There are two possible sequences of observations, one beginning with D and terminating with X the other beginning with E and terminating with Y. The initial states D and E are equi-probable and as in the earlier tape reading tasks the agent has a single read action. In both sequences, the region between the initial letter and the terminating letter is a homogeneous region of hidden-state which consists of three consecutive A's.

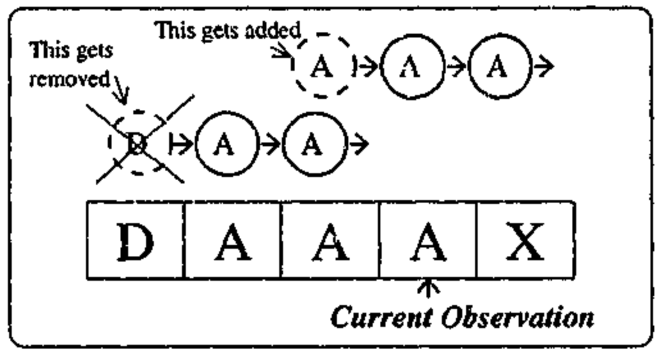
Figure 6.9 demonstrates various stages in the construction of a chain to correctly predict X. The first stage is the initial construction of the chain. In this stage, the chain covers the search path containing the state prior to X and its immediate predecessor state (see Figure 6.9(a)). In the second stage, the agent experiences another presentation of the sequence beginning with D. The first A in this sequence matches the first group, so the chain is extended to include D (Figure 6.9(b)). However, continuing the sequence with the subsequent two presentations of A leads to the removal of the extension to D. This removal is due to the discovery that following the newly extended chain path does not lead to the chain prediction (observation X) but rather the observation A. However, the occurrence of this third A again matches the first group of the chain (since the chain's path is no longer being followed) and a new extension is created to include the prior observation, which in this case was the second A



(a) Initial creation of the chain for the D sequence. This occurs at the end the presentation of one sequence from D to X.



(b) Extension of the chain for the D sequence. This occurs once the sub-sequence of D followed by A has been experienced.



(c) Removal of the first extension and an attempt to create another. These both occur in the same timestep of a later presentation of the sequence starting with D and ending with X. The removal occurs because X is not predicted or passed. The extension ensures X will be predicted in a later presentation of this sequence.

Figure 6.9: Three different stages of chain construction and extension for a sequence to represent the path from D to X.

in the sequence (see Figure 6.9(c)). The final observation experienced is an X, not an A, so the chain's path has not been followed and this most recent extension is also removed.

Without a modification to the rules for extending chains, the current chain will never be extended correctly and the aliased regions never distinguished. The necessary modification is to allow a chain extension to be retained not only if the chain prediction occurs subsequent to following the chain execution path, but also if the observation associated with the prediction is experienced in place of an observation expected when the chain execution path is being followed. This allows the extension made in Figure 6.9(c) to be retained and the final extension to D to be made and retained in a subsequent trial.

In the experiments in Chapter 7 homogeneous temporal chains are flagged when the first two links have the same group as a subordinate. This is sufficient for the experiments presented in Chapter 7. However, in general it may be necessary to flag a chain as homogeneous if the subordinates of any two links in the chain are ever matched in the same timestep.

6.5.2 The Importance of Actions

The discussion of temporal chains so far has focused primarily on how TRACA represents hidden-state problems with a single action. However, in general we are interested in how that representation can be used to improve an agent's ability to select appropriate actions in diverse environments. One such environment is the simple homogeneous maze environment in Figure 6.2 from Section 6.2. In this environment the agent can select one of four possible actions to move it north, south, east or west, except if the action leads it into a wall in which case its position remains unchanged. This task is now used to illustrate some of the complications associated with building chains when multiple actions are possible.

One of the decisions made when designing TRACA was whether or not temporal chains should include sequences of observations (as done in Ring (1994)'s Temporal Transition Hierarchies) or observation/action pairs (as done in TRACA and McCallum (1995)'s U-Tree, among others). The difficultly in this decision is that for many spatial navigation tasks, such as our homogeneous corridor navigation task in Figure 6.2, it is possible to build temporal chains which reliably predict the end of the corridor without regard for a single specific set of actions taken to reach it. Such a chain would in fact match multiple execution paths to an end state from a start state as long as the paths were all the same length (the length of the chain) and terminated at the same state. Two such paths are depicted in Figure 6.10.

A chain which is independent of a specific sequence of actions offers the advantages of allowing a single chain to match multiple paths. This is potentially useful when considering exploration. An agent which takes one or more exploratory actions each time it traverses a corridor may be capable of building one chain which leads to the chain's prediction rather than separate temporal chains for each different execution path. The disadvantage with this is that the action values for links in the chain will be inaccurately updated unless one path is consistently followed. However, even inaccurate estimates may be useful, for example, to drive the system towards a positive reward at the end of a corridor (or away from a negative reward). The real problem for temporal chains which are developed independent of a specific sequence of actions is that to be successful the type and number of exploratory actions must be restricted. This negates the advantage of action independent chains. Figure 6.11 shows examples of paths of the same length which do not lead to the state we want to predict at the end of the corridor. If the number and type of exploratory actions are not restricted, a large number of temporal chains may still need to be created to represent the different path lengths, and we no longer have a small number of temporal chains. Note, when information on actions is excluded from temporal chains the resulting representation is similar to that created by the indexed memory methods described in Section 2.11.

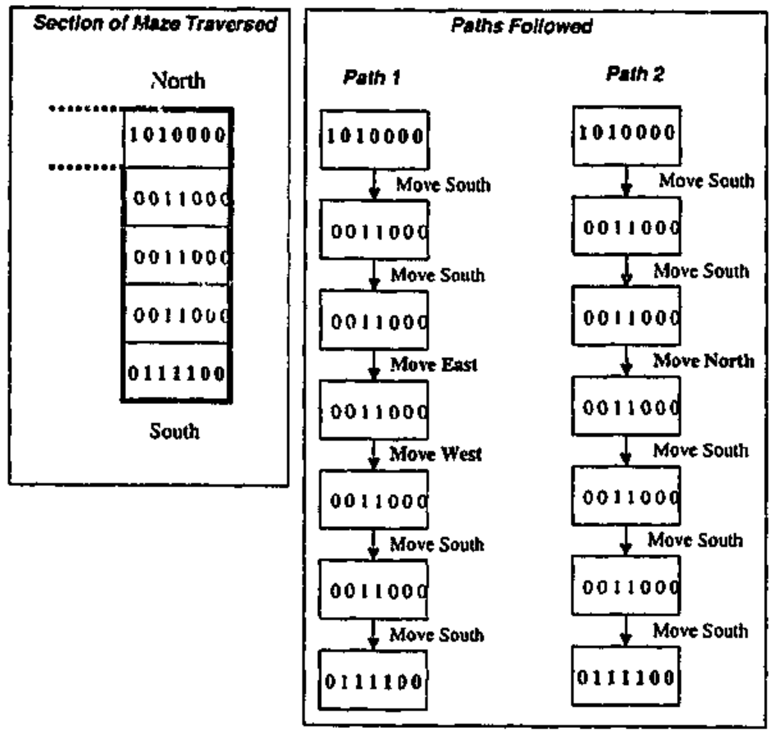


Figure 6.10: Two possible paths of the same length which lead to the same final state.

On the other hand, having temporal chains based on sequences of observation/action pairs, will allow more accurate transition and value estimates, but will require a chain for each different path the agent may take. Furthermore, if the agent is taking a large number of exploratory actions, completing a chain to represent even one of these paths may take an enormous number of trials, unless the agent consistently follows one path.

Consequently, TRACA constructs temporal chains based on sequences of observation/action pairs (providing highly reliable value estimates) and attempts to represent only a small set of commonly used paths, allowing some flexibility. This flexibility is particularly important for stochastic domains. For example, if a robot which traverses a long corridor experiences variations in the distances travelled for each of its motor movements, this will require chains

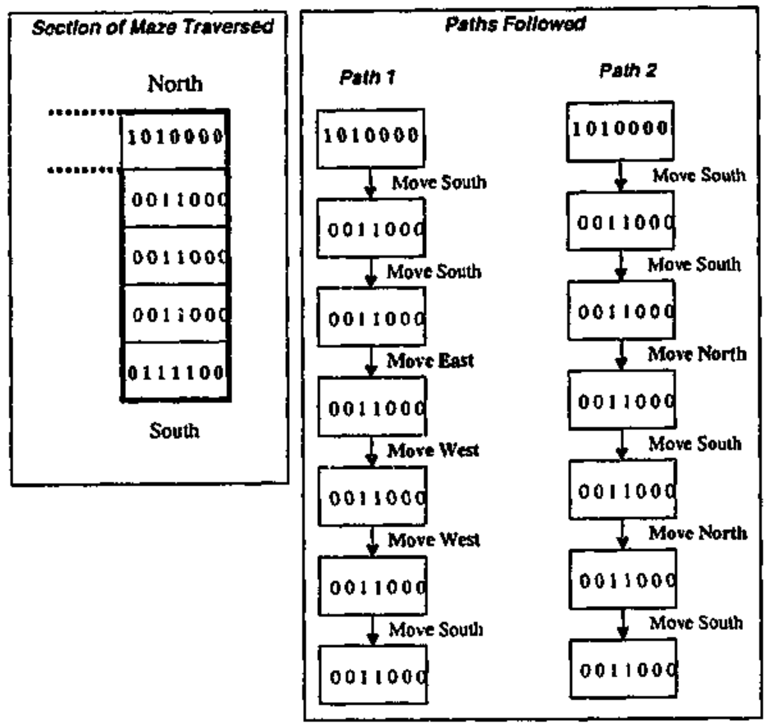


Figure 6.11: Two possible paths of the same length which do not lead to the same final state.

for different numbers of motor movements to reliably direct the agent to the end of that corridor.

TRACA's design for hidden-state therefore relies on a small set of paths being taken frequently during learning. A strategy for this in spatial navigation environments is presented in Section 7.6. This strategy allows learning in a relatively small number of trials and also avoids the potential exponential growth of temporal chains (a problem U-Tree is also susceptible to (McCallum 1995)).

6.5.3 Selecting Features for Chain Extension

When using localised sensors to represent features the extension of chains is relatively straight forward as a single unary group is matched at each timestep which summarises the features of the current world state. However, if using distributed sensors, many unary groups may be matched at each timestep, one for each individual feature in the environment, consequently there are a number of possible groups that can be selected from when doing chain extension. One possible method is to randomly select from the groups eligible for creating extensions (eligible groups are those which were matched and not support suppressed in the relevant previous timesteps). A second method is to select from the eligible groups the one which contains nodes with the highest predictive accuracy. Within TRACA predictive accuracy can be assessed using node ETP values.

The first method allows a variety of possibilities for different features while the second allows for the search to favour particular features. Take, for example, the maze shown in Figure 6.1 of Section 6.2. In this maze there are two identical heterogeneous corridors and there is a bit position in the agent's input string for each possible feature in the environment such as the presence of windows and doors. However, now add the possibility of a person being present at one or more of the positions in the maze. The presence of the person does not require any specific action by the agent (let us assume that people move out of the agent's way), however, there is an additional bit in the input string which contains a 1 when a person is present in the agent's current position. Given that the probability of a person being present in a location at any time is less than 1.0, then it may be better to select for temporal extension groups which represent more reliable features such as windows rather than the group indicating the presence of a person.

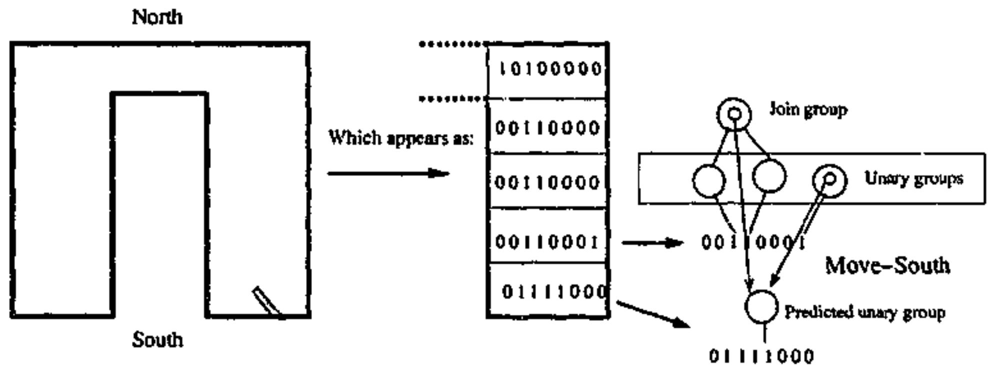


Figure 6.12: Hidden-state regions (corridors) with a homogeneous input pattern and the input strings experienced during one trial. The right-most bit indicates the presence of a person in the positions prior to the end of each corridor (with 0.25 probability). The node in the join group (representing the surrounding walls) predicts the door and will have an ETP of approximately 0.1. The node in the unary group representing the person also predicts the door and will have an ETP of 0.5. However, the precedence rate for the node in the join group is 1.0 while the precedence rate for the node representing the person is 0.25.

The use of node ETP values to distinguish more reliable features from less reliable ones works fine for heterogeneous temporal chains, however, it fails for homogeneous temporal chains. The problem in Figure 6.2 of Section 6.2 is similar to the one in Figure 6.1 except that each of the corridor locations has the same features and therefore the same input string. Taking this problem we now add a bit to indicate the presence of a person in a position and allow during trials a 0.25 probability of a person being at the position prior to the end position of each corridor location at any time. When constructing temporal chains to successfully navigate south in each corridor the group representing the presence of a person will appear a better predictor of the door (with an ETP of approximately 0.5) than the join of the groups representing the surrounding wall bits (with an ETP of approximately 0.1). In this case, it is better to use a third method for selecting links in temporal chains. This third method requires a new variable which estimates the rate at which a predicting node precedes its predicted group. In TRACA this variable is calculated by predicted groups sending a message to each of the nodes that predict them each time the predicted group is matched. Each time the predictor nodes receive this message they then update a *precedence rate* based on equation 6.1 (in Section 6.4.1) with r being 1.0 if they executed in the timestep prior to the group being matched, and zero otherwise. The resulting situation for the homogeneous maze is depicted in Figure 6.12. The precedence rate will indicate that a join group combining the two input bits based on surrounding walls precedes the door with probability 1.0 while the presence of a person would have an associated probability of 0.25. An experiment demonstrating the successful use of the precedence rate is presented in Section 7.5.

6.6 An Example of Representing Hidden-state

The following example uses a simple hidden-state problem to illustrate the creation of TRACA's temporal chains. It first describes structures created by TRACA's basic system of unary and join groups (which highlights inadequacies with the basic system for representing problems with hidden-state). Then it augments these basic structures with temporal chains to uncover hidden-state (this is done in Section 6.6).

6.6.1 The Example Problem

A letter prediction problem is used as the example problem. This problem requires the agent to correctly predict the letter at the end of two possible sequences of letters. The problem is designed so that memory of the initial letter presented in the sequence is necessary to do this successfully. The number of letters between the initial letter and the last remain the same in each sequence and determine the size of the gap across which the initial letter must be remembered (Mozer 1992; Ring 1994). The two sequences are shown in Figure 6.5 and each has four letters. Since two letters will be seen between the first and last letter of each sequence the gap is two. During each learning trial, letters are presented to the learning agent in each timestep one after another until the end of the sequence. The presentation of an entire sequence (to either C or D) constitutes a single trial. Trials are independent, hence the agent does not try to predict which letter or sequence will follow a completed sequence. The two sequences are presented with equal probability and in each case the letters A and B appear between the initial and last letter. If the initial letter is an X, the fourth letter will be C, if the initial letter is a Y the fourth letter will be D.

With each letter presented the learning agent can attempt to predict which letter will appear next. As done in the input generalisation experiments (see Section 5.2.4) the system indicates its prediction of one of A, B, C or D by its choice of one of the four possible effectors. Once an effector is selected (and regardless of which one is selected) the next letter in the sequence is presented. Each effector is labelled according to the letter prediction it corresponds to. This labelling is for the convenience of the following discussion and has no meaning to the agent. However, since rewards will be allocated based on the desired selection of effectors (according to their label) the agent is capable of learning the required behaviour. The required behaviour for our example is to correctly predict the final letter based on the preceding letters presented in each sequence. This means that for each sequence the agent must remember the first letter and after being presented with the letter B in that sequence it must select the effector whose label corresponds to the final letter in the sequence. If this is done correctly, the agent receives a positive reward of 100, otherwise the agent receives a reward of zero. Note, that TRACA's selection of an action to predict the final letter is quite separate from the predictions of TRACA's nodes. The first is a policy driven behaviour the second is a predictive model of observations based on prior actions and observations.

	t_1	t_2	t_3	t_4
Sequence 1	X	A	B	C
Sequence 2	Y	A	B	D

Table 6.5: The two sequences for a letter prediction problem with gap 2. In each trial the letters are presented in order from t_1 to t_4.

6.6.2 Initial Representation

As in all its tasks, TRACA receives information about the current state of its environment as fixed length input strings. Each position within the string corresponds to one of TRACA's detectors and contains either a 0 or 1. In our example, the letters are encoded for TRACA's detectors using localised sensors. That is, one bit position of the input string is reserved exclusively for each observation (letter). Only the current letter being presented is represented in the string. After each letter is presented, TRACA selects an effector which indicates its prediction of the next letter. The presentation of a letter and subsequent prediction of the next constitutes one *major cycle* (described in Section 4.2).

After a number of trials TRACA will have developed an initial representation of the sample problem (without temporal chains). The entire set of unary groups and nodes created by TRACA for this task after these trials is presented in Figure 6.13. In the figure rectangles are groups and squares are nodes. Each node's prediction is indicated using an arc from the node to the predicted group. The arc is labelled on top with the node's supported effector and underneath with the node's ETP. The number in each node's box is its utility value estimate using a discount rate of 0.1. Figure 6.13 indicates that a unary group has been created to represent each observation (letter) and within each group, nodes have been created for each possible action (unary group and node creation was explained in Chapter 4). An alternative representation of nodes and arcs for the bold path through the sequence shown in Figure 6.13 is presented in Figure 6.14.

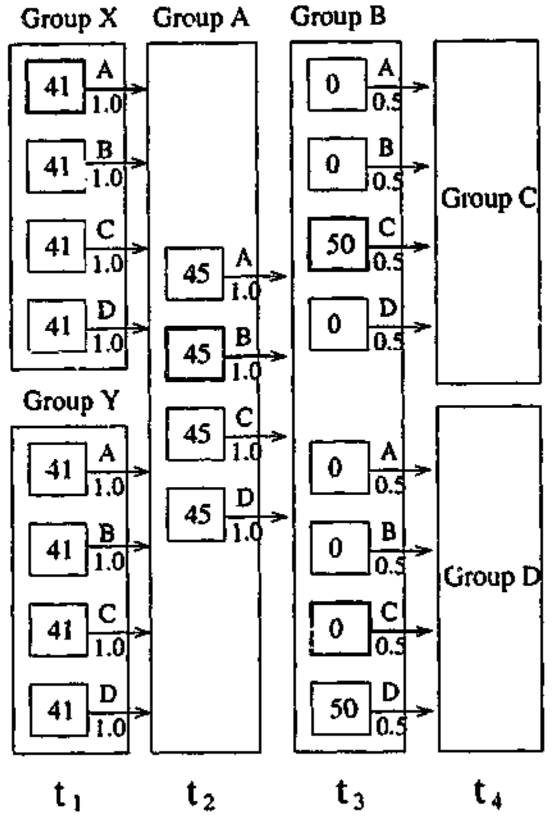


Figure 6.13: All the non-temporal nodes and groups developed. The group matched at t_4 in a trial could be either C or D depending on the sequence presented. Bold structures show one path which might be taken during a trial.

As TRACA has no initial knowledge of the states, transitions or reward function for the environment it is operating in, ETP's and value estimates are initially 0 and must be updated during trials in the environment. Consequently, in early trials when TRACA has no knowledge on which to base decisions, effector selections are made with uniform random probability. In the example, a group such as X has nodes for each effector because over trials when TRACA has received observation X it has selected all four effectors. The fact that the next observation will always be A can only be discovered through trial and error.

In our sample problem, letter A always follows both X and Y, and B always follows A, so each node in the groups X, Y and A has an ETP of 1.0 in Figure 6.13. The nodes in group B, however, have an ETP of 0.5 as letters C and D each appear after B with equal probability. The groups C and D have no nodes, as they are reached on completion of trials at which point no further predictions are possible.

6.6.3 Improving Predictions with Temporal Chains

For nodes to correctly predict the last letter in each sequence, temporal chains need to be added to the agent's basic model of unary groups and nodes. Figure 6.15 shows the commencement of a new temporal chain which is constructed in an attempt to reliably

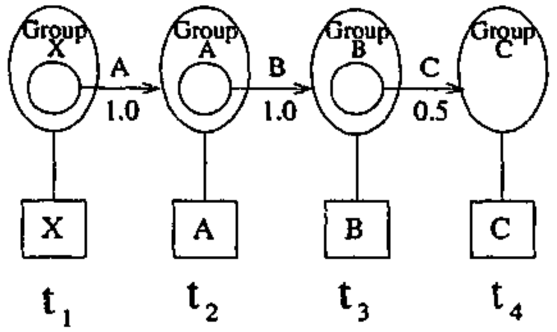


Figure 6.14: An alternative representation of the structures for one path in the gap problem.

predict the final letter C. The creation of this chain is triggered based on the fact that during one sequence (after unary nodes have had sufficient initial trials), the group representing C is predicted by an extant unary node with an ETP below 1.0, indicating the node has at some stage made a prediction incorrectly. In fact, at this stage all of the unary nodes in group B which predict C have made incorrect predictions as indicated by their ETPs of 0.5. However, in our example the chain was created based on the prediction of the node in group B which predicts C and has an action labelled C. The incorrect prediction is due to the transition probability from B to C being 0.5. At this point, without a-priori knowledge the agent has no way of knowing whether the problem it is facing is stochastic or whether it contains hidden-state. It assumes a hidden-state problem and attempts to discover the history (and if joins were possible, the combinations of currently available features) necessary to uncover the hidden-state.

The construction of the temporal representation begins with a temporal chain depicted as Chain T in Figure 6.15. The figure shows the chain at a point where chain T has already received several trials. Chain T was created with the presentation of the fourth letter in a trial (at t_4) when C was incorrectly predicted by our node. At its time of creation Chain T includes the nodes whose actions were selected for each of the two prior observations, the one received at t_2 and the one received at t_3. This is achieved by TRACA always keeping track of which nodes executed in the last two major cycles. This is done specifically for the creation of temporal chains when triggered.[12]

Chain T consists of two groups (links), depicted in Figure 6.15 as T2 and T3. The group T3 is the terminal group. If the chain is successful, the reliable predictions of nodes in the terminal group will eventually replace the less reliable predictions of the nodes in its subordinate unary group. Like their non-temporal counter-parts (from Chapter 4), temporal groups and nodes are created by existing groups, in our case the groups labelled C and D. Link groups in the chain, other than the terminal group (such as T3), have only one node for the action which predict the next group in the chain. Like nodes in other groups, these nodes

---

[12] The criteria for selecting nodes which TRACA keeps track of requires nodes to have completed initial trials and not being support suppressed.

record ETP and value estimates. The ETPs for nodes in links in temporal chains are updated based on the matching of the next link in the chain given that the current link's associated effector is selected. This reflects any non-deterministic transitions occuring during the execution of chains.

Temporal Rule Execution

Nodes in non-terminal groups update values whenever they execute which only occurs when all the nodes in the chain's temporal groups and terminal group have executed in sequence. There are therefore two concepts of matching and executing: *link matching and executing* and *chain matching and executing.* Each successive link group *executes* in a major cycle when its subordinate is matched, its node's effector selected and in which the chain's execution path has been followed up to the current timestep. Once all the link groups have been matched in turn, and the terminal group's subordinate is matched, the entire chain is matched. Now when the primary terminal node executes, the chain executes. Given these occurrences, temporal chains can be seen as having their own *temporal cycle* which encompasses all the major cycles (executions) of the consecutive links in the chain. For example, the node in T2 will commence matching at t_2, if the effector B is selected, T2 will execute and T3 will be matched in the next major cycle at t_3, at which point the entire chain is matched.

Updating Link Value Estimates

Once each link group executes, the link's node updates its utility value estimate based on the return from the next link and any immediate rewards received (if the next link is not matched in turn both the return and reward are zero). However, apart from nodes in the terminal group, the ETPs of nodes in links are not used in comparisons with equivalent subordinates.

6.6.4 Hierarchical Relationship

The terminal group T3 is built using group B as a subordinate. This relationship is shown in Figure 6.16 where T3 is depicted as superior to Group B. Each time T3 is matched, the node in T3 is compared to the equivalent subordinate node in the group for B (the node with the same prediction and action) to collect statistics for the test for noise (see Sections 4.2.9 and 6.4.3) which is used to determine if any of the chain terminal nodes provide an improvement over their equivalent subordinates.[13] Once the chain provides an improvement according to the test for noise, it can be marked as *final.* To prevent duplicate temporal chains, terminal groups which are matched but not final set a global flag which prevents the creation of any temporal chains in the next major cycle. But even before a chain is final, other nodes are created in the terminal group for each alternative prediction that occurs concurrently with the first node's prediction when the chain's temporal cycle is complete.

6.6.5 Temporal Value Estimates

New nodes in T3 initially update their utility value estimates and ETPs in conjunction with existing nodes in group B without sending support for effector selection. The aim of developing the chain is that once it is complete and provides a solution, the nodes in T3 send support in place of the nodes in B when T3 is matched (in a temporal cycle). Any rewards received as a result of correctly predicting C or D are now also passed back to nodes in T3 which will pass a discounted return to the node in T2 during subsequent trials. T2 may pass this back in turn to either nodes in previous links of the chain, or to nodes in groups which predict the first group in a final chain. Each link passes returns to the previous link when it is matched and the chain execution path has been followed to that point. Value estimates are updating using the rule described in Section 4.2.7.

The value that nodes' ETP and utility estimates converge to for our example are also shown in Figure 6.15. From these it can be seen that the chain so far does not yet have sufficient memory to provide an improvement in the prediction of C or D. The nodes in T3 predict both C and D with ETP's and value estimates similar to the original nodes for P3 in Group B.

At this point I will take a slight digression to re-state the effects of having only one node in each link of a temporal chain. Ideally each link would represent all action and state transitions from that link's state. Furthermore, each link should both *support* and *connect* suppress its subordinates since its estimates are likely to be more accurate. However, this is expensive because each link must create many nodes, often replicating with little change the nodes in its subordinate group. Also, before the chain nodes are of any use, they require a number of trials. This necessary experience is presumably obtained by taking a variety of exploratory actions. However, we would like to reduce the number of paths followed when creating chains, otherwise will we get many chains for each region of hidden-state and the lack of experience with any single path may result in many trials passing before useful chains are completed.

To avoid the necessity to gain all this experience and create all this structure, TRACA instead allows the nodes in a chain link's subordinate group to send support concurrently with its superior link's node. This allows superior link nodes to drive system behaviour based on the returns associated with following the chain's path, while the subordinate nodes indicate the worth of other possible paths. This is achieved by simply not allowing link groups to support suppress their subordinates (with the exception of the terminal group).

---

[13] As mentioned in Section 6.4.3, the comparison conditions are different for retaining chains representing homogeneous regions, their primary node's ETP must exceed a minimum threshold.

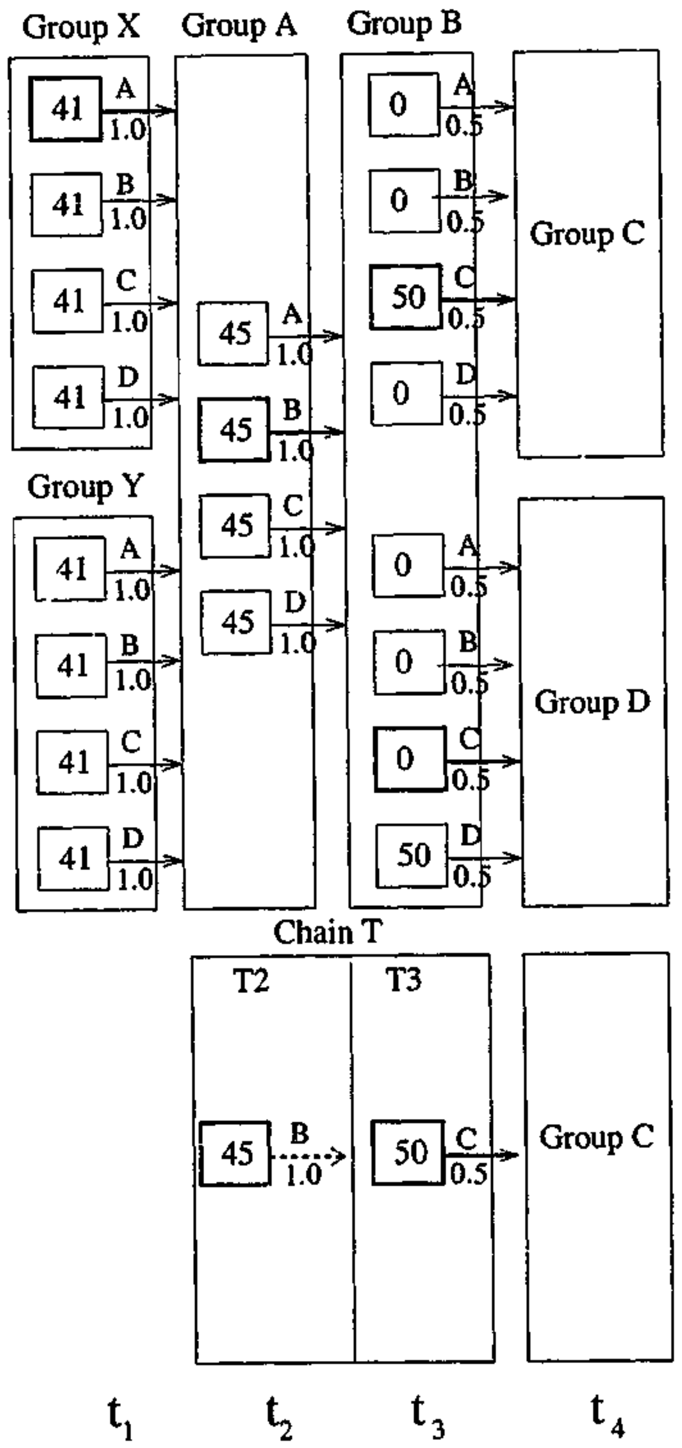


Figure 6.15: The commencement of the temporal chain T and the utility values converged to.

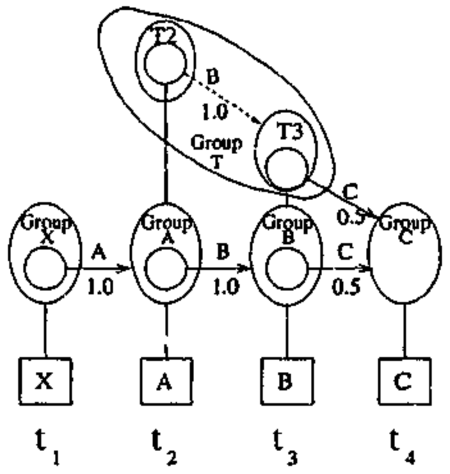


Figure 6.16: Alternative representation of one path along the temporal chain T. T3 is depicted as hierarchically higher, or superior, to group B.

6.6.6 Controlling Temporal Chain Extension

The problem of creating and extending temporal chains without any guarantee of improvement is unavoidable. As with joins (see Chapter 4), it is not possible to know whether a chain will improve on existing groups and nodes until it is generated and given a number of trials. After these trials, a decision can be made to either retain the new structure or remove it. However, temporal chains differ from non-temporal joins in that it may involve many combinations of existing groups (as it is extended) because it is not known before-hand how far back the chain should extend before considered it for removal. To allow for the sufficient extension of temporal chains, links are continually added to chains even if the chain does not currently provide an improvement over existing groups and nodes. This extension process could continue indefinitely, but in TRACA can be constrained using the discounted terminal value described in Section 6.4.3. This value is used to identify temporal chains that have been extended to the point where the rewards (or penalties) received as a result of following the chain are either too small or too distant to justify the continued construction of the chain.

6.6.7 Completing the Chain

Continuing our example, after more trials, the chain in Figure 6.15 is extended so that the new link in chain T is matched by the first letter in the sequence being presented. A group, either X or Y, depending on which was matched last given the current sequence, is chosen as a basis for extending the chain. Such a chain, extended to X is shown in Figure 6.17 and an alternative representation is given in Figure 6.18. If the chain had been extended to Y, it would have extended into an invalid search space. This mistake will be revealed when the temporal cycle completes and the terminal node's prediction is incorrect (not matched), causing the most recent extension to be discarded. In either case, the chain is eventually extended to X at which point the chain has sufficient memory to accurately predict C. The ETP and utility estimates now rise (due to the recency weighting of the update rules, see

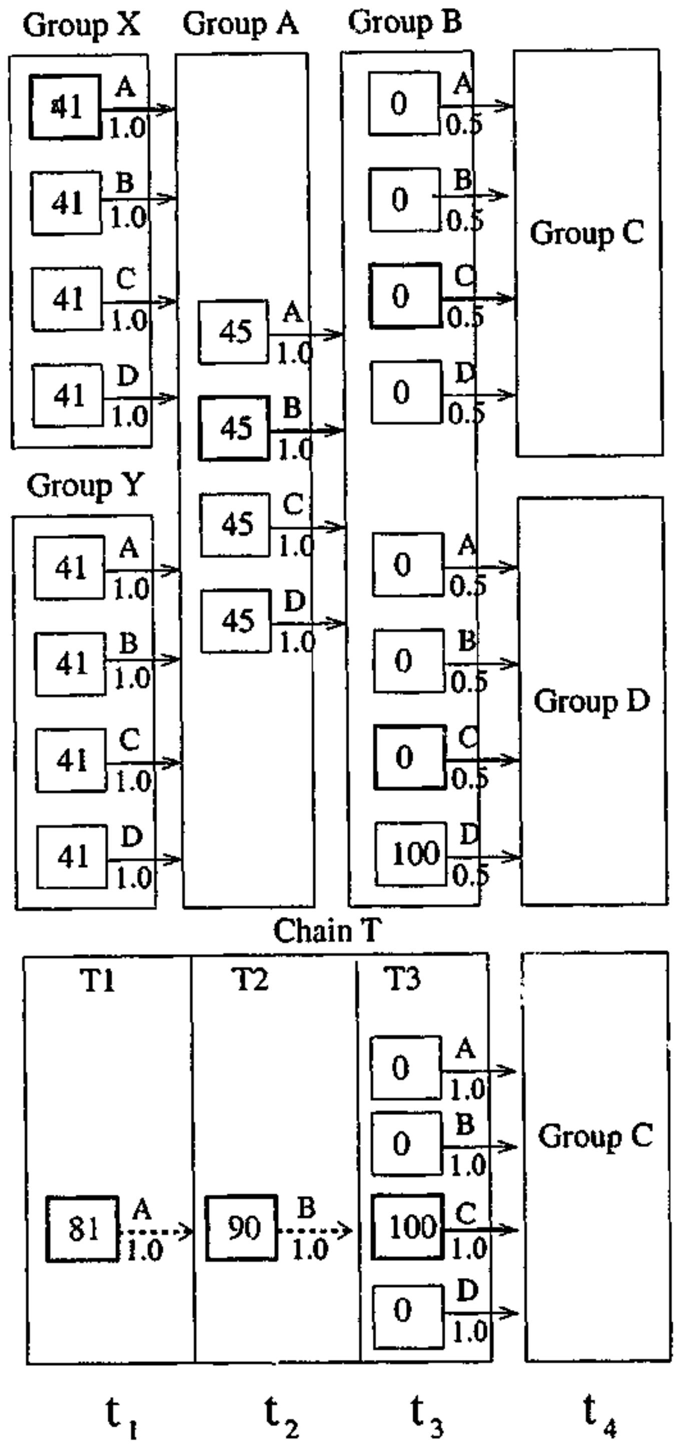


Figure 6.17: The Chain T after extension along with the utility values converged to.

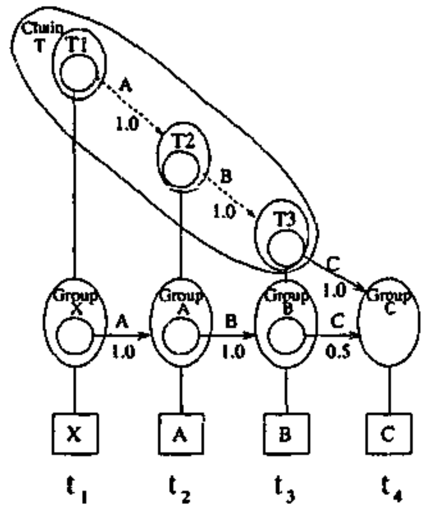


Figure 6.18: Alternative representation for one path along extended temporal chain.

Section 4.2.7) for the node in the terminal group T3 and the chain is set to final. Groups which are predicted by a final chain will remove subsequent chains which make predictions simultaneously with a completed chain.

The completion of the chain leads to the ETP and utility values depicted in Figure 6.17. The value for the nodes in group B which predict C fall to zero (as these nodes are support suppressed each time the chain is followed) and the value for nodes predicting D rise (as they are prevented from executing incorrectly). Although group A is not directly affected by the chain, the values of nodes in this group will approach 45. This occurs because after the sequence to D is experienced, they receive a reward of 100 (before discounting) from group D. However, when the sequence to C is experienced (and the chain's path followed) they receive a reward of zero, since their prediction is support suppressed. If another chain is formed to predict D, the value of the nodes shown in group A will drop to zero as they will always receive a reward of zero.

6.6.8 Representing Alternative Paths and Outcomes

The example solution extends the representation to accurately predict C. However, this example has omitted the creation of other chains that may take place concurrently with the one to predict C. For the agent to successfully perform the example task, it is also necessary to accurately predict the outcome D and to represent other paths since there are many possible execution paths. In this task, there is little that can be done to avoid creating multiple chains when using observation/action pairs (see Section 6.5.2 for a discussion of this), however, in tasks such as spatial navigation, it is possible to restrict the number of execution paths to a small set of frequently followed paths. This is demonstrated in experiments with TRACA presented in Section 7.6.

6.7 Steps for Creating and Using Temporal Chains

An outline of the steps for developing temporal chains is presented below. Descriptive comments are included using *italics*, values of state variables are indicated as `fixed width` and verbs are indicated as **bold**.

Steps for temporal chains:

The system starts with a set of effectors, a set of detectors, and a set of unary groups. There is one unary group for each detector and one detector for each bit in the input string. Unary groups initially contain no nodes. When created, temporal chains comprise of one or more `link` *groups one of which is a* `terminal group`*. The terminal group is at the end of the chain's execution path. The* `first link` *is at the start of the execution path and is the most recent extension to the chain.*

1. An input string is received by the system and detectors determine if they are `matched` based on the value in their bit-position.

2. Detectors notify their associated unary groups whether they are `matched` or `unmatched`. Groups pass the notification to their superiors in turn until all groups have received a notification. Links in temporal chains are matched if the chain's execution path has been followed to the link and the link's subordinate is matched. A temporal chain is `matched` once its path has been followed up to the terminal group.

   Final chains which are matched send a broadcast message which prevents the creation of chains in the next timestep.

   Note that there are two notions of matched in relation to temporal chains, there is the matching of the individual links, which must occur in sequence for the entire chain to be matched when the terminal link is matched (this is the temporal cycle).

3. Extant temporal chains update their ETP values if they were `matched` in the previous cycle based on whether the primary terminal node's prediction is `matched` in the current cycle.

   Chains which are not final and whose execution path was followed to this point, send a broadcast message preventing the creation of other chains in the current timestep.

4. If a chain was `matched` last major cycle and was extended at the start of its temporal cycle its prediction must be `matched` this major cycle or the most recent extension is removed. Homogeneous temporal chains do not remove extensions if their prediction was passed during the temporal cycle.

   This determines if a chain has been extended down an invalid search path.

5. Terminal groups belonging to `final` temporal chains which are `matched` this cycle and unsuspended send messages to their immediate subordinates which sets them to

`support suppressed` and `create suppressed`. This message is passed down the network by the immediate subordinates until all the subordinates are `support suppressed` and `create suppressed`.

6. Groups which are `matched` this cycle **create** terminal nodes to predict them in terminal groups whose temporal chains are final and were `matched` in the previous cycle. `create suppressed` terminal groups are excluded.

   Allow the chain's terminal group to have nodes which predict multiple groups. To ensure we only create predictors for a chain which will occur in a cycle following the chain being matched, new predictors in the terminal group of a non-final chain will only be created if the chain's execution path has been followed.

7. Groups which are `matched` this cycle send any nodes which predicted them a **return** based on the maximum reinforcement value of nodes in the matched group.

8. An `extend list` is created of all non-temporal groups which are eligible to be used in creating new links to extend temporal chains back in time. `Unsuspended` groups which were `matched` and were not `create suppressed` by superior groups in the previous cycle are added to this list.

9. Temporal chains which are not `final` and for which the most recently added link has not demonstrated an improvement over existing structures, **extend** themselves by creating a new link to a group selected randomly from the `extend list` in the previous step. Those which have demonstrated an improvement set themselves to final.

   For homogeneous chains, once the chain has had sufficient number of executions and the ETP of the primary terminal node has not improved on the ETP of its equivalent subordinate, the chain is eligible to be extended. Homogeneous chains are identified as such if their two initial links have the same subordinate group.

10. Nodes which **executed** in the previous cycle receive the system **reward**.

11. If the chain is `matched` in the current cycle it passes back a **`return`** to the previous group in the chain based on the maximum value of its nodes. The terminal value is also passed back along the chain to all links, each applying their discount before passing it back in turn.

12. Temporal nodes which `executed` in the previous cycle **update** reinforcement value estimates using immediate rewards and returns received. Returns are zero if the next link (group) in the chain is not `matched` this cycle.

13. A group which is `matched` this cycle and has one or more predicting nodes which fired in the previous cycle, whose group was not support suppressed and not suspended and those nodes have a dependent ETP less than the threshold value (have been incorrect in the past) **create** a new temporal chain.

Only one chain can be created each major cycle to prevent duplicates.

The created chain's terminal group is superior to the predicting node's group. The chain also has one link which is superior to a randomly selected group matched in the cycle prior to the previous (at t_n, create a temporal chain using the observations from t_{n-1} and t_{n-2}).

Groups which have one or more predicting nodes which fired in the previous cycle whose ETP exceeds a minimum threshold (i.e it is already accurately predicted) do not create temporal chains.

14. A `previous extend list` is created for extensions. It consists of all groups placed in the `extend list` this cycle.
15. Temporal chains which are too long to be useful are removed based on the chain length and the value of the terminal node's prediction.
16. Terminal and join groups which are `matched` this cycle send notification to their immediate subordinate groups setting them `support suppressed` this message is passed down the hierarchy until all the sending group's subordinates are suppressed.
17. Nodes in unsuspended join and terminal groups which were `matched` this cycle and not `support suppressed` **fire**, sending support to their associated effectors.
18. The system selects an effector based on the support sent by nodes this cycle. Nodes which supported the selected effector **execute**. Terminal groups in final temporal chains which are `matched` set a flag to prevent any other groups being created next cycle.
19. **Repeat**

6.8 Chapter Summary

Like TRACA's join structures, temporal chains have been designed to deal with large, stochastic environments which may change over time. TRACA's chains are incrementally extended down different paths in the search space which avoids the need for fixed size history windows. In general, the success of this technique depends on the ability to distinguish homogeneous regions of hidden-state from heterogeneous regions as the two need to be assessed differently. Homogeneous regions that extend over a number of states also require specialised treatment for correct extension. Finally, a range of measures are taken to reduce the search for useful chains and to prevent duplicate structures. These measures include using the reward function to direct the agent down a small number of paths and removing lengthy chains which lead to states with low utility.

Chapter 7

Experiments using Temporal Chains

In this chapter a number of experiments are conducted using temporal chains to represent hidden-state. The experiments test the ability of TRACA to deal with heterogeneous and homogeneous regions of hidden-state, hidden-state regions of varying lengths and the combination of hidden-state with noise. Most of the tasks used in the experiments in this chapter are commonly found in the literature and they are typically based on localised sensor schemes (a representation where each possible observation maps to a unique bit in the agent's input string). However, two tasks are included which use a distributed sensor scheme. The first is a maze task for which only the development of temporal chains can provide a solution. The second is a simulated truck driving task for which both temporal chains and non-temporal joins can provide the solution.

The next section describes the experimental methodology for the different hidden-state tasks using localised sensors.

7.1 Experimental Methodology

Developers of different algorithms for hidden-state have used a variety of tasks to demonstrate their systems. These tasks have been drawn from a wide range of domains, however, the spatial domain involving robot navigation is perhaps the most common. This domain provides sufficient variety to include many of the difficulties presented by hidden-state, as well as having the practical slant of allowing researchers to address current issues in robotics. Consequently, many of the following experiments are based on different spatial navigation tasks.

7.1.1 Dimensions of Hidden-state Problems

Different hidden-state problems in the literature include different dimensions of difficulty.[1] One dimension of difficulty is the length of memory required to be constructed in order to

---

[1] I have borrowed this term from (Ring 1994).

make correct predictions. A variety of domains have been used to demonstrate this ability, but they all have the common feature that an observation in a sequence needs to be remembered so as to correctly predict a later observation (Robertson and Riolo 1988; Mozer 1992; Hochreiter and Schmidhuber 1997). Since an arbitrary amount of time may pass between the initial observation and the last, these tasks are refered to as *gap* tasks (Mozer 1992). TRACA is applied to a number of gap tasks, cast in the form of simple navigation problems. TRACA's performance on the gap tasks is compared to various neural network algorithms for which gap tasks can be difficult. This difficulty appears to be primarily due to their shared internal representations which makes learning gaps and remembering information for long time periods difficult (see Section 2.10.4). The gap experiments presented in Section 7.2 demonstrate that TRACA can represent gaps of varying length, n and that variations to n in these tasks will affect TRACA's learning times but not TRACA's ability to find a solution. Furthermore, these experiments demonstrate TRACA's relative learning times for homogeneous regions (gaps) compared to heterogeneous regions (gaps) (see Section 6.2 for a discussion of these).

Another task that is regularly used for comparing hidden-state algorithms is a maze from McCallum (1993) called the M-maze (due to its shape). This maze is used for two experiments. The first (in Section 7.3) is similar to gap tasks, however, a wider choice of actions allows more varied sequences of experience. The second experiment (in Section 7.4) adds another dimension of complexity (noise) to the task by randomly changing the observation associated with states and the effects of actions. Following this noisy M-maze experiment is another gap task, but one in which distributed sensors are used. This gap task demonstrates TRACA's ability to deal with irrelevant attributes.

The gap and M-maze tasks consist entirely of corridors which have a constant discrete width. Many other mazes used in the literature are also based around corridors of discrete width but with varying corridor lengths and topology. One exception to this, which TRACA is compared to, is the work by Ring (1994) who developed mazes which resemble other types of real internal environments. In addition to the corridors commonly seen in other mazes, Ring (1994)'s mazes also have the equivalent to open spaces within rooms. These open rooms provide a range of hidden-state regions and more variety in the possible policies that may be learned (i.e from simple wall-following to direct traversal of hidden-state regions). For this reason two mazes used in Ring (1994)'s experiments are included in the trials of TRACA. The first is based on a 5x4 grid, the second is based on a 9x9 grid and is the largest of the mazes used in experiments by Ring (1994). These experiments are presented in Section 7.6.

7.1.2 Performance Metrics

The ability of a learning system to represent gaps is one metric commonly used in the literature to assess hidden-state algorithms. Others include the number of structures created, the computation involved in developing those structures and the number of trials required to

learn a solution. A further possible measure is the quality of the solution, in terms of its optimality. However, as discussed in Section 1.4.1, optimality is not necessarily a good measure, since the costs of obtaining it are often too high (Wiering and Schmidhuber 1998; Bowling and Veloso 1999). This leaves the other measures. Computation is important as demonstrated by results using belief state approaches. These approaches also demonstrate the importance of a small number of training examples which can be exorbitant, prohibiting real-world training (Littman 1994b).

In order for TRACA to meet the requirements in Chapter 1, it is necessary that it can learn using a relatively small number of trials. This is important for any agent operating in the real-world to minimise damage to the agent and its environment (McCallum 1995). The second requirement is that the number of structures created and the computation required is within tractable bounds. The experimental results that have already been presented, and those that follow, all demonstrate that neither TRACA's computational or space requirements are excessive. The final requirement is that the algorithm can solve a range of problems. While an algorithm may perform well on problems with hidden-state, unless it can also deal with input generalisation its usefulness as a multi-purpose learning agent is questionable.

7.1.3 Training and Testing Procedures

Different authors apply different procedures for training and testing their agents. However, agents are commonly provided with training episodes and testing episodes. During training episodes, exploratory actions may be taken and agents may develop internal structures and adjust value estimates. During testing episodes, all learning is turned off, no changes are made to internal structures and values and no exploratory actions are selected.

Each episode consists of timesteps and trials. A timestep consists of a single observation and action selection by the agent. Trials are a sequence of timesteps. In the following experiments, a trial may be terminated by either reaching the goal or by not reaching the goal within a specified time limit.

Episodes are terminated in several different ways. For the gap tasks, training and testing is completed when temporal chains are developed which manual inspection determines are sufficient to extend across hidden-state regions and make correct predictions. For the M-maze a fixed number of learning timesteps are provided in each training episode. Finally for Ring (1994)'s mazes, training and test episodes consist of a fixed number of trials. To learn Ring's mazes, TRACA uses a variant on Ring's action model which allows the agent to take full advantage of information provided by the reward function.

7.1.4 Parameter Settings

Where possible TRACA's parameters are set to those used by the original author of the experiments. However, when long gaps are present, TRACA's prefered exploration strategy is to select random actions using the roulette-wheel approach (see Section 4.2.13). This increases the probability of TRACA frequently traversing common paths. Other parameters are as follows, with variations noted as appropriate for individual experiments.[2] The number of cases for the test for noise was 20 and the Cox-Stuart test for trend used 10 cases. The number of initial trials required by a unary node in a unary group before the group was set to unsuspended was 20. The *improvement factor* (IF) applied to node ETP values, after their ETP rose above that of both equivalent subordinates, was 1.02 and the *temporal threshold*, (TT), was 0.98 (see Section 6.4.3). The best supporter method was used to determine the effective support value for each effector.

In most experiments a learning rate of 0.2 was used. This varied if the original experiments used a different learning rate and on noisy tasks a lower learning rate was used. Changing the learning rate for temporal structures has a similar effect as for non-temporal structures. A lower learning rate will reduce the effects of oscillations due to recency weighting and constant learning rates but may increase learning times as ETP values may take longer to reach their steady-state value. In all tasks, the standard learning rule was used, however, on the truck driving task the SRS rule was also trialled (this rule was described in Section 4.2.7).

Following is a description of the experiments, commencing with the gap tasks, then McCallum (1993)'s M-maze, Ring (1994)'s mazes and finally with a truck-driving task from McCallum (1995). The truck driving task uses distributed sensors, allowing both temporal chains and non-temporal joins.

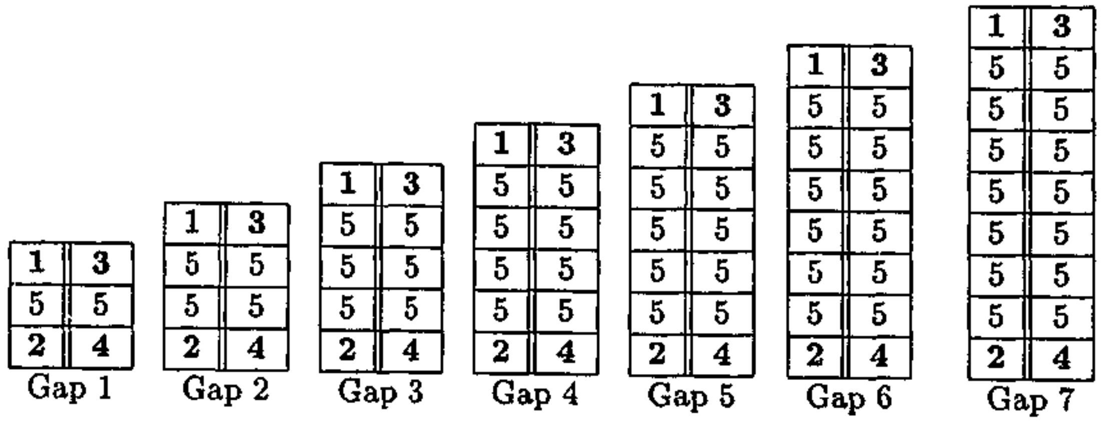
7.2 Gap Tasks

Mozer (1992) introduced gap tasks to test his *Reduced Description Network*. Success on these tasks required the system to remember an initial observation across a fixed sequence of observations in order to correctly predict the final observation in the sequence. Ring (1994) also tested his *Temporal Transition Hierarchies* on these tasks, and more recently Bakker (2002) has tested LSTM networks on T-maze problems similar to Mozer's gap tasks.

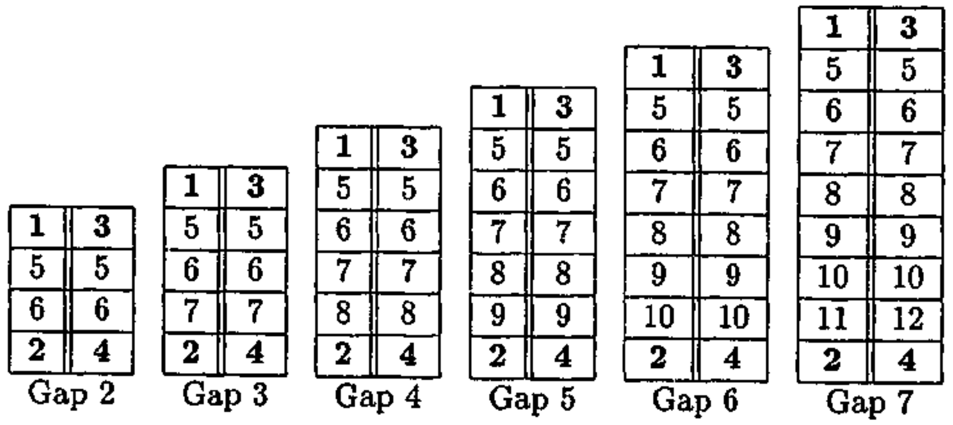
The gap tasks here serve two purposes. The first is to demonstrate that TRACA can remember observations for varying lengths of time. Gaps of up to 7 observations have been included in the experiments. These tests are sufficient to demonstrate that TRACA can represent gaps of arbitrary length, as long as a single path to the goal is followed with sufficient frequency.

---

[2] Many of these are unchanged from the input generalisation experiments in Section 5.2.



(a) The six homogeneous gap tasks and the Gap 1 task. In the homogeneous tasks the same observation is repeated across the shared region.



(b) The six heterogeneous gap tasks. Each observation is different within the shared region.

Figure 7.1: The mazes used in the gap experiments.

The second purpose of the gap tasks, is to compare the learning times for sequences of observations (corridors) that are homogeneous and heterogeneous. Homogeneous sequences are those with the same observation repeated across a number of states while heterogeneous sequences are sequences of different observations. In both cases, the sequences appear in multiple regions of the state space (see Section 6.2).[3] In the following experiments with TRACA the gap problems are represented as simple maze navigation tasks. The homogeneous tasks are depicted in Figure 7.1(a) and the heterogeneous tasks in Figure 7.1(b). Each gap experiment involves one maze with two possible corridors with a fixed length gap. In Figures 7.1(b) and 7.1(a) the numbers at the top of each task are the observations seen at the start of each sequence and the numbers at the bottom are the observations seen at the end of each sequence. The system must remember the observation at the start of a sequence in order to reliably predict the observation at the end.

---

[3] Mozer (1992)'s original experiments did not include homogeneous gaps.

7.2.1 Experimental Design

Each gap experiment consists of a single training episode with a number of trials. At the start of each trial the agent is placed at the top of one of the corridors (selected with equal probability). The agent can select from four possible actions, Move-Up, Move-Down, Move-Left, Move-Right. In positions other than those at the top and bottom of corridors, selecting Move-Up will place the agent in the position above its current position, selecting Move-Down will place it in the position below. In all positions selecting Move-Left and Move-Right will leave the agent's position unchanged, as will selecting Move-Up in a top position. The position at the bottom of each corridor is the goal. On reaching the goal the agent receives a positive reward of 100 and the trial terminates. No other rewards or penalties are provided. The experiment is completed once the agent has developed all the structures necessary to disambiguate each position in each corridor. For each task, two chains are required, one to represent each corridor. For example, the Gap 1 task in Figure 7.1(a), requires a chain to remember the initial observation 1 across the hidden-state region when observation 5 is presented in order to reliably predict observation 2. A similar chain is needed to predict observation 4 when the sequence started with observation 3. Construction of the required chains is confirmed using a manual inspection of the structures developed and retained in the system, guided by a visualisation tool which indicates when structures have been completed.

The probability of selecting actions to move into walls in the gap tasks depends on the exploration support given to actions with zero value. In these experiments this value is 0.5 (for details see Section 4.2.13).

During learning the agent selects actions using the *best supporter* method (see Section 4.2.13) with probability 0.9. With probability 0.1 an action is selected randomly using the roulette wheel approach. A learning rate of 0.2 is used.

7.2.2 Results

Table 7.1 shows the number of timesteps that elapsed during training trials on each of the gap tasks. For the task with a gap of 1 it took the agent 309 steps in the environment to learn the two temporal chains required. These steps will have included a number of exploratory moves into walls and along a variety of paths to the goal. Before chains can be created and support sent to effectors to drive the agent down the shortest path to the goal, a number of trials were needed for each of the nodes in the unary groups to develop ETPs and utility estimates. Learning times for homogeneous gaps increase linearly from 309 for the Gap 1 task up to 1339 timesteps for a gap of 7.

The heterogeneous gap tasks take up to 3 times more timesteps to learn than the homogeneous tasks with the same gap size. The heterogeneous learning times range from 535 timesteps for a gap of 2 up to 3902 for a gap of 7. These differences between learning times for homogeneous and heterogeneous mazes increase in proportion to gap lengths. The reason for this is that in homogeneous corridors, unary nodes in the hidden-state regions are executed more frequently as the observation matching their group is presented more often. This leads to the agent sending support and returns sooner (see Section 4.2.8) which leads it along paths to goals sooner and more frequently (this could be improved by reducing the number of trials required by unary nodes before setting their group to unsuspended).

	Steps to Learn	
Gap	Homogeneous	Heterogeneous
1	309	n/a
2	330	535
3	469	1046
4	636	1756
5	861	2166
6	1015	3116
7	1339	3902

Table 7.1: Comparative performance on the homogeneous and heterogeneous gap tasks.

Of interest in the gap tasks is the scenario of changing the reward function to make it equally desirable to go both ways in the hidden-state regions (this is related to the problems described by Whitehead and Ballard (1991) where learning may diverge from the optimal policy). Consider a variation on the gap tasks for homogeneous task 3 in Figure 7.1(a) where the only reinforcements provided are equal positive rewards for reaching positions 1 and 4. At the start of each trial the agent is placed in one of the positions 2 or 3 with uniform random probability. Once initial trials for nodes are complete and rewards have been received for each corridor, the agent oscillates between one direction and the next. Chains could overcome this problem, but the creation of chains is prevented by the agent's oscillating behaviour which rarely traverses paths to either goal. This is a problem which effects learning in the next experiment. A generic solution suitable for spatial navigation tasks is presented in Section 7.6.

7.3 McCallum's M-maze

The M-maze from McCallum (1993) is presented in Figure 7.2 which shows the states and the observations received in each state. There are 11 states and 7 observations, according to surrounding wall configurations. This maze was also used in experiments with belief state approaches by Littman, Cassandra, and Kaelbling (1995) (see Section 2.9 for a description of these approaches). For TRACA, as in Littman, Cassandra, and Kaelbling (1995), the goal state has an observation of its own. Following McCallum (1993), in each trial the agent is placed in a random position (other than the goal position) and must navigate to the goal.

The agent receives a reward of 1.0 when the goal position is reached. If the agent moves into a wall its position remains unchanged and it receives a penalty of -1.0. All other transitions have a reward of -0.1.

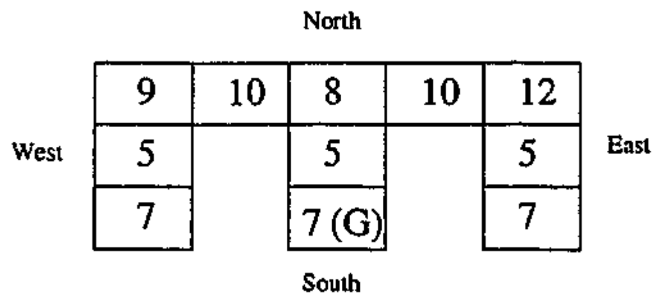


Figure 7.2: The M-maze showing the observations and goal state.

7.3.1 Experimental Design

The experimental design also follows McCallum (1993). The agents are allowed a series of training episodes in the maze. The number of timesteps allowed in each training episode is fixed at 500. Within a training episode when the agent reaches the goal it is replaced in a randomly selected position (other than the goal). After each training episode the agent is given a 100 test trials with random effector selections turned off. Each test trial terminates when the agent reaches the goal or a maximum of 500 timesteps pass without reaching the goal.

If over the 100 test trials the agent does not reach the goal on every test within 500 timesteps, the agent has failed the test and is given another training episode before being tested again. Once the agent reaches the goal on 100 consecutive test trials, the agent is considered to have successfully learned the task and training is completed.

TRACA was run on this problem with 1 in 10 actions selected with uniform random probability during learning, a learning rate of 0.6 and a discount rate of 0.7. 20 agents were run in the maze, each with a different random seed.

7.3.2 Results

All of the 20 TRACA agents passed the test within a maximum of 17 training episodes. Of these, the average steps to goal (over the 100 test trials) on completion of training was 4.3 (standard deviation 0.13) and the average number of training episodes was 8.3 (standard deviation 4.17). The average number of chains created during learning was 3.9 (standard deviation 1.5) using 4.4 temporal groups (standard deviation 2.4). McCallum (1993) reports that his Utile Distinction Memory (UDM) algorithm learned this task consistently within 5 episodes. Littman, Cassandra, and Kaelbling (1995) also report success on this task for a belief state POMDP learning algorithm within 75,000 training steps, however, they did not attempt to optimise learning times (TRACA's average was 4150.4, standard deviation 2084.4).

When comparing TRACA's result on the M-maze to McCallum (1993)'s reported result, TRACA required on average 3.3 more training episodes than UDM. In fact, the reason this maze takes so long to learn is because reinforcement based learning agents oscillate between states on the east and west branches of the maze. In TRACA's case, until appropriate temporal chains are developed, this occurs whenever the agent receives a return for moving south from a position labelled 5. One node in the group for observation 7 predicts a high return for moving north (to receive observation 5) while another node in the group representing observation 5 predicts a high return for moving south. If the agent is placed or moves itself to either the east or west branches of the maze, it will repeatedly oscillate between the two states associated with these observations. A similar oscillation occurs between the states with observations 10 and 9 and 10 and 12. This behaviour leads to long learning times as the agent rarely gets to experience other parts of the maze. Both McCallum (1993)'s UDM and TRACA relied upon the high learning rate of 0.6 to reduce oscillating behaviour on this task. A high learning rate quickly reduces the value of a repeated action which does not achieve immediate rewards allowing the agent to move away to another area of the maze.

7.4 Noisy Sensors and Effectors

McCallum (1995) also ran his UDM algorithm on some experiments in the M-maze with noisy sensors and effectors. In his experiments, with probability 0.1 the observation was selected randomly from all the possible observations in the maze. Also with probability 0.1 the agent's selected action was changed to an action randomly selected from the 4 possible actions. In this experiment we repeat McCallum (1995)'s experiment and compare TRACA's performance to UDM's.

7.4.1 Experimental Design

The parameters used by TRACA on this noisy version of the M-maze were identical to the deterministic version of the maze except the learning rate which was changed to 0.1 and the discount rate to 0.9. These parameters are different from the M-maze experiment in Section 7.3 since the noise allows the agent to escape from oscillating behaviour.

7.4.2 Results

In his trials, McCallum reports that the learning time for UDM increased to 15 training episodes. TRACA reliably learned the maze with an average of 9.2 training episodes (standard deviation 3.2). There was a large variation in the number of training episodes required to learn this task. The minimum was 3 the maximum 16. The average number of training steps required was 4600.4 (standard deviation 1594.6). The average number of timesteps to reach the goal on test trials was 6.1 (standard deviation 0.5) with noisy sensors and effectors. With this noise turned off the average time to goal for test trials was 4.6 (standard deviation 0.3). On average 8.2 (standard deviation 3.3) chains were created using 10.6 temporal groups (standard deviation 4.2). The maximum number of chains created in a single run was 17 and the minimum 2.

On this task TRACA required far less training than UDM and only 1.2 training episodes more than was needed to learn the M-maze without noise.

7.5 Irrelevant Noisy Features

In the real world, people move in and out of corridors. The presence of people in the corridor is both unpredictable and irrelevant to navigating the corridor (assuming they keep out of the way). This task tests the use of the precedence rate described in Section 6.5.3 to direct the extension of chains away from such irrelevant features.

7.5.1 Experimental Design

The homogeneous gap 5 task from Section 7.2 is used to test the use of the precedence rate variable to select links (paths) for chain extension. However, now distributed sensors are used so that the presence of a person in the same location as the agent (in any aliased corridor state) can be indicated by a value of 1 (and 0 otherwise) in a bit position in the agent's input string. The input string also has a unique bit for each of the corridor location observations from the original Gap 5 task. As in the original Gap 5 task, in the current task the agent has 4 possible actions, with actions which lead it into walls leaving its position unchanged. Again the agent is provided with a number of trials in a single training episode. On reaching either of the positions labelled 2 or 4 the agent receives a positive reward and is placed with equal probability in one the positions labelled 1 or 3. All other transitions have zero reward.

In this task distributed sensors are used. In addition to the bits in the input string indicating the agent's position in the maze (in which a number of locations are perceptually aliased) there is an additional input bit to indicate the presence of a person in the corridor. For each of the corridor locations labelled 5 in the gap 5 task of Figure 7.1(b) the probability of a person being present is 0.25.

The learning rate was 0.2 and the exploration rate 0.1 with the roulette wheel method used for action selection. Other parameters are unchanged from Section 7.2. Learning is stopped once chains have been constructed which remember the initial observation and accurately predict the two goal states when the shortest path is followed from the initial state (determined by code inspection guided by a visual tool). Therefore, one chain must extend

from the position labelled 1 to the position labelled 2, the other from the position labelled 3 to the one labelled 4. The presence of the required chains is determined using a tool to perform a manual inspection of TRACA' structures as they are completed. Five agents were trialled on this task, each with a different random seed.

7.5.2 Results

TRACA developed the required chains with an average of 871 (standard deviation 97.6) timesteps in the maze. This learning time is not substantially different from the same task without the person of 861 (see Section 7.2). For every chain created, no links were constructed on the group (bit position) indicating the presence of a person. A similar trial selecting the group for chain extension based on the dependent ETP of those groups failed to construct chains which excluded extensions based on the feature indicating the presence of a person.

The creation of only two chains for each run on this task provides a large efficiency in learning and representation. An approach using localised sensors would require chains for every possible combination of observations that could be traversed in the hidden-state region (assuming the best case scenario where the shortest path to each goal is always followed).

When using distributed sensors, the use of a precedence rate provides a powerful bias to extend chains down paths which most regularly lead to predictions. In this case, it allows TRACA to avoid irrelevant noisy features in such searches.

7.6 Ring's Mazes

Ring (1994) uses a sequence of mazes to demonstrate both the ability of his Temporal Transition Hierarchies (TTH) to represent hidden-state and how his system adapts knowledge used on one task to solve other similar tasks. Ring's mazes are of progressive difficulty and each requires learning corridors of varying lengths. Their representation is similar to the M-maze, in which states are positions in the maze labelled according to the surrounding wall configuration. Attempts to move into walls leave the agent's position unchanged. In Ring's mazes the state labelling scheme leaves increasing numbers of states perceptually aliased (i.e more hidden-state) as the size of the maze increases. TRACA was applied to the smallest and largest of Ring (1994)'s maze tasks, a 5x4 maze and a 9x9 maze. These two mazes are treated separately in two following sub-sections and depicted in Figures 7.3 and 7.4 respectively showing the wall labelling for each position and the goal (marked as G).

As was argued in Chapter 1, for successful learning with a small number of trials some external assistance is required. In the experiments with TRACA in this section, assistance is provided by manipulating the reinforcement landscape. This approach is desirable because it is consistent with the reinforcement learning philosophy that a human programmer need not know how to solve the problem (a requirement if explicit teaching is used) nor need to explicitly specify a solution (for hand-coded solutions). The aim of the reinforcement scheme adopted here is to encode heuristics for moving in spatial environments. The scheme is intended to direct the agent's experience into useful areas of the problem domain and to encourage the repeated following of a smaller set of paths in each maze (this type of approach is also advocated in Koenig and Simmons (1996), see Section 2.14). This will hopefully also prevent oscillating behaviour associated with hidden-state regions such as experienced in the M-maze problem in Section 7.3.

In the following experiment, TRACA uses an *unoriented* action model. The *unoriented* action model has actions: Turn-Right-And-Move, Turn-Left-And-Move, Turn-Around-And-Move and Move-Forward. Ring (1994) uses an *oriented* action model in his experiments which allows the agent to select actions which move it north, south, east or west. As we will see, the unoriented action model allows the agent to take advantage of penalties for changing direction. The reinforcement scheme used with TRACA (and unlike the one used by Ring (1994)) involves a reward of 1000 for reaching the goal state (indicated as G in Figures 7.3 and 7.4). Wall avoidance is encouraged by providing a penalty of -7 for each attempt to move in a direction which is blocked by a wall. Turning left or right has a penalty of -3 and turning around (180 degrees) a penalty of -4. A similar reward scheme is used by Lin (1993) for his robot navigation task. The reward for the goal needs to be sufficiently high that the benefits of achieving it are not cancelled out by the penalties associated with turns required to reach it.

Penalising changes in direction requires that the agent has knowledge of at least one of the following: the last action, its current direction, or its orientation. In any of these cases the extra information increases the state space by a factor of 4 (compare Figure 7.4 with Figure 7.5). Ring (1994) included one set of experiments where inputs provided information about the agent's last action, which he refered to as using *proprioceptive sensors.* TRACA's experiments provide equivalent information based on the agent's current orientation (i.e currently facing north, south, east or west). This is consistent with (i.e an alternative to) the ability of Ring's agent to perceive wall orientations in relation to compass directions. Following Ring's experiments, localised sensors are used, therefore each possible wall configuration has 4 possible input strings, one for each orientation (see Figure 7.5).

7.6.1 Experimental Design

The experimental design follows that used in Ring (1994). Learning proceeds in a series of training episodes. Each training episode has 100 learning trials and each trial commences with the agent being placed in a random location with a random orientation (other than the goal location). A trial terminates when the agent reaches the goal or if it does not reach the goal within a maximum number of timesteps (1000). On completion of a trial the agent is placed in another randomly selected position with a randomly selected orientation. After each training episode (100 trials), the agent is provided one test episode. Test episodes also consist of 100 trials, but with learning turned off. If the agent fails to reach the goal on any trial in the test episode it fails the test and is provided another training episode. If after 5 training episodes the agent has not successfully completed a testing episode, it is deemed to have failed to learn the maze. 100 TRACA agents were trialled on the maze. Following Ring (1994), failures are excluded from the average results.

During training a learning rate of 0.2 is used and exploratory actions are selected with probability 0.1 using the roulette-wheel method.

7.6.2 Results on Ring's 5x4 Maze

On the 5x4 maze the average training time for TRACA was 3904.9 (standard deviation 1458.9). 57 agents required 1 training episode, 35 required 2 and 8 required 3. TRACA created 3.9 chains on average (standard deviation 1.14) using an average of 6.2 temporal groups (standard deviation 2.4). The average number of steps to the goal for the 100 TRACA agents over the 100 test episodes was 6.6 (standard deviation 0.68). Ring (1994) reported training times for TTH of 2984. TTH's average timesteps to goal during testing was 6.2 and an average 5.9 internal structures were created. On this task, TRACA required, on average, nearly 1000 training steps more, and the learned policies took slightly longer to reach the goal.

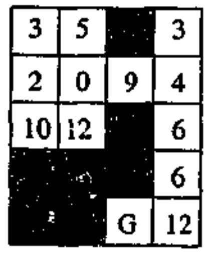
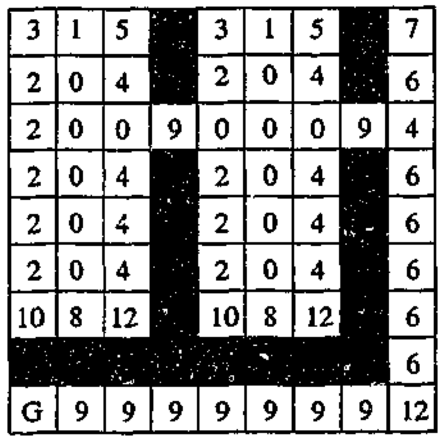


Figure 7.3: Ring's 5x4 maze.

In fact, this maze is not difficult to solve. TRACA can reliably and consistently find a policy for it without creating any temporal structures by learning a wall-following strategy. Trialling 100 TRACA agents under the same conditions as the experiment above but without temporal chains (temporal chain creation is turned off) TRACA achieves an average number of steps to goal of 6.7 (standard deviation 0.53) , with an average training time of 5142.8 (standard deviation 1739.6). 71 agents required 1 training episode, 27 required 2 and 2 required 4. This performance after training without temporal chains is very similar to that obtained using chains, however, the average training time was longer.

7.6.3 Results on Ring's 9x9 Maze

Ring (1994) reported average learning times of 38,153 training steps on his 9x9 maze (when learning from scratch). The average number of steps to the goal during successful testing was 24.8 with 1 of the 100 agents he trialled failing. On average his network introduced 14.5 internal structures.



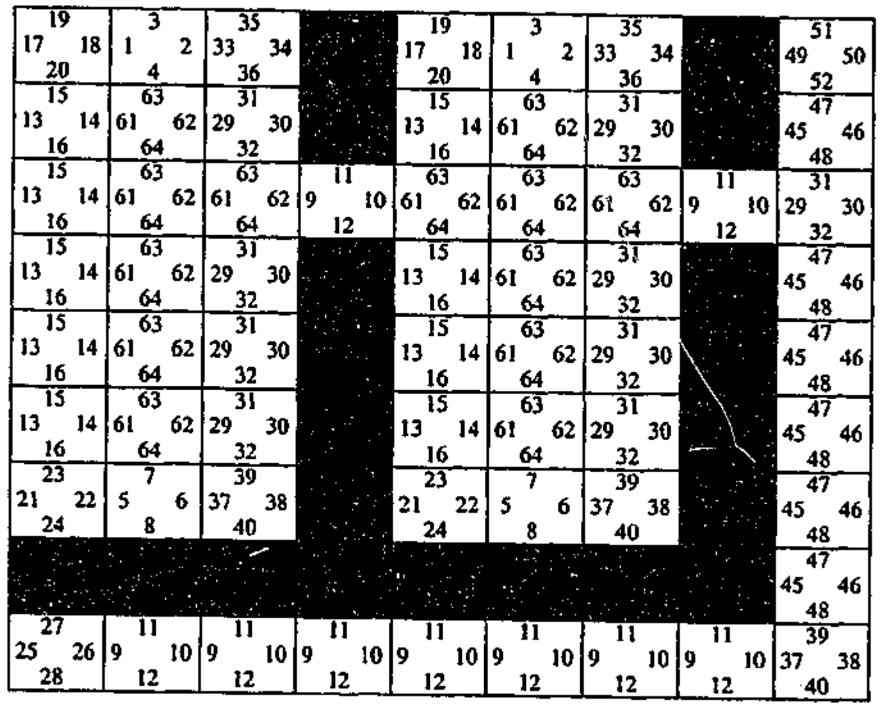
3	1	5	■	3	1	5	■	7
2	0	4	■	2	0	4	■	6
2	0	0	9	0	0	0	9	4
2	0	4	■	2	0	4	■	6
2	0	4	■	2	0	4	■	6
2	0	4	■	2	0	4	■	6
10	8	12	■	10	8	12	■	6
■	■	■	■	■	■	■	■	6
G	9	9	9	9	9	9	9	12

Figure 7.4: Ring's 9x9 maze.

In trials with TRACA, 3 out of 100 agents failed to learn the 9x9 maze and were excluded from the following results. However, the successful agents required an average of only 18222.4 training steps (standard deviation 10177.1) with nearly half the agents learning the maze within the initial 100 trials. During testing an average of 23.3 steps was was required to reach the goal (standard deviation 2.7). TRACA's learned network contained an average of 39.4 chains (standard deviation 15.8) with a maximum of 82 and a minimum of 17. On average 66.2 temporal groups were contained in chains (standard deviation 28.5). Because this is a deterministic task, all the chains created were due to the presence of hidden-state and the rewards/penalties associated with states was sufficiently high that all chains representing hidden-state were retained.

A graph of TRACA's training times using the unoriented action model is presented in Figure 7.6. This graph shows the cumulative number of successful agents after each training episode of 100 trials. After the first episode, 47 of the agents had successfully learned the maze. After the second episode, 71 are successful. The third episode sees 88 successful, the fourth 93 and the fifth 97. It is difficult to translate Ring's average training steps to an estimate of the average number of episodes his agents' required, but based on equivalent learning times using TRACA it is probably around 3.

A comparison of the structure created by TRACA and that created by TTH is presented in Figure 7.7. Ring (1994) provided only one description of the structure created by TTH on his maze problems and that was from what he described as a "favourable" run on a series of progressively more difficult mazes culminating in the 9x9 maze. Figure 7.7(a) provides an



19 17 18 20	3 1 2 4	35 33 34 36	■	19 17 18 20	3 1 2 4	35 33 34 36	■	51 49 50 52
15 13 14 16	63 61 62 64	31 29 30 32	■	15 13 14 16	63 61 62 64	31 29 30 32	■	47 45 46 48
15 13 14 16	63 61 62 64	63 61 62 64	11 9 10 12	63 61 62 64	63 61 62 64	63 61 62 64	11 9 10 12	31 29 30 32
15 13 14 16	63 61 62 64	31 29 30 32	■	15 13 14 16	63 61 62 64	31 29 30 32	■	47 45 46 48
15 13 14 16	63 61 62 64	31 29 30 32	■	15 13 14 16	63 61 62 64	31 29 30 32	■	47 45 46 48
15 13 14 16	63 61 62 64	31 29 30 32	■	15 13 14 16	63 61 62 64	31 29 30 32	■	47 45 46 48
23 21 22 24	7 5 6 8	39 37 38 40	■	23 21 22 24	7 5 6 8	39 37 38 40	■	47 45 46 48
■	■	■	■	■	■	■	■	47 45 46 48
27 25 26 28	11 9 10 12	11 9 10 12	11 9 10 12	11 9 10 12	11 9 10 12	11 9 10 12	11 9 10 12	39 37 38 40

Figure 7.5: Ring's 9x9 maze with orientation incorporated in inputs. The number at the top of each box is the input received when the agent is facing north, the number at the bottom the input received when facing south, and the numbers on the left and right are received when facing west and east respectively. Each number is represented as a single unique bit in TRACA's input string.

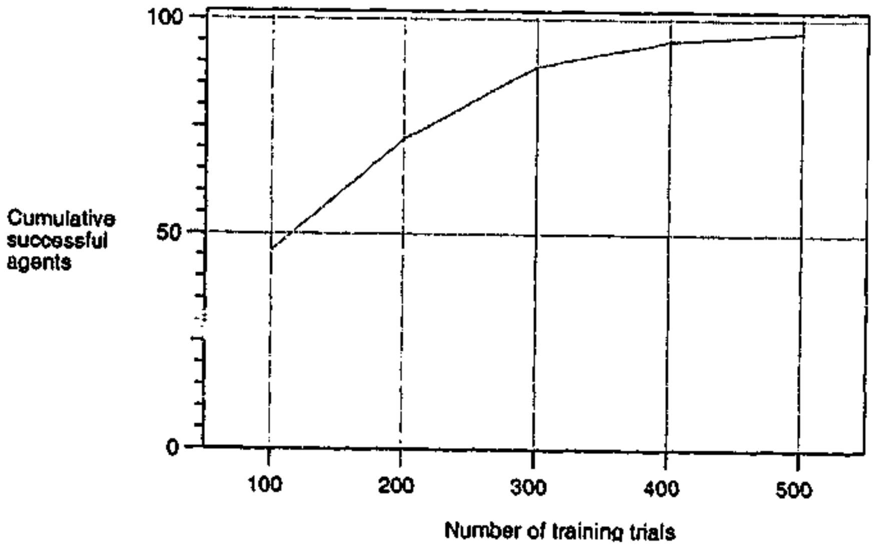
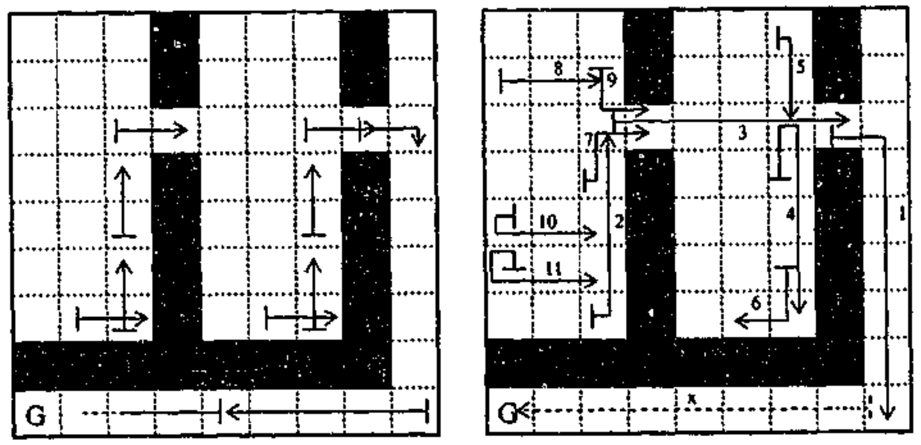


Figure 7.6: Number of training trials required by the 97 successful TRACA agents on Ring's 9x9 maze task.

indication of the structure developed by TTH to implement short-term memory based on his description. These structures were created while learning non-proprioceptive mazes (i.e the state space depicted in Figure 7.4). For comparison, an abstract of the set of structures created during TRACA's shortest run on the 9x9 problem are presented in Figure 7.7(b).

In both sub-figures of Figure 7.7, the butt of each arrow indicates the state a single memory sequence commences in and the head the state it completes in. In TTH's case, the memory sequences do not include actions (which is why Ring introduced proprioceptive sensors). In TRACA's case, changes in the direction of arrows indicate the actions associated with the sequence. The orientation of the butt indicates the agent's orientation when the sequence commences. However, since Ring's compared structures are from a non-proprioceptive run, orientation is irrelevant in Figure 7.7(a). In the case of TRACA, the head of the arrow indicates the action and prediction of the primary terminal node. However, the terminal group generally contains nodes with other actions and predictions. For clarity, Figure 7.7(b) excludes eight short chains which are less relevant to the solution. Most runs using TRACA constructed a chain to represent the region extending across the bottom of the maze to the goal, this is indicated using an arrow with a dashed line in Figure 7.7(b). The run represented in Figure 7.7(b) is atypical in that it did not quite complete this structure before successfully completing the 100 test trials, however, it still solves the maze.

The complete set of chains developed by TRACA is presented in Appendix B.1.



(a) Structure created in one run by TTH to solve the 9x9 maze problem.

(b) Structure created in one run by TRACA to solve the 9x9 maze problem.

Figure 7.7: Comparison of structures to solve 9x9 maze problem.

7.7 Discussion of Results on Maze Tasks

A summary of TRACA's training times and structures created by the successful agents on the different maze tasks; the M-maze, the noisy M-Maze, Ring's 5x4 maze and Ring's 9x9 maze, is provided in Table 7.2. For each of the tasks listed in the first column, the table shows the average number of training steps required, the average number of temporal chains and groups created and the maximum and minimum number of chains created. Mazes are listed in order of the average number of training steps required. The results in the table indicate that as the number of training steps required to learn a task increase, the number of temporal structures increases. This rise can be explained by the increasing difficulty of the tasks listed in Table 7.2. The M-maze and Ring's 5x4 maze required similar amounts of structure (chains and temporal groups) reflecting their similarity in difficulty, size and the number of hidden-states. However, Ring's 5x4 has a longer hidden-state region than the M-maze which is reflected in the greater number of temporal groups created. The noisy M-maze is more difficult, as more paths are possible due to the noise. This difficulty appears to be associated with a moderate increase in the number of structures, however, the creation of only 3 chains in one run suggests it is possible to learn this task with little more structure than required for the version without noise. Finally, there is Ring's 9x9 maze. This maze has significantly more hidden-state than the others which is reflected in the large increase in chains created for this task when compared to the others. The average number of chains created was 39.4, however, this has a large variance ranging from a minimum number of 17 chains to a maximum of 82. This suggests that TRACA often, and correctly, creates chains to represent hidden-state regions in this domain that are not necessary to the particular task at hand.

Task	Training steps	Chains created	Temporal groups created	Max chains	Min chains
Ring's 5x4 maze	3904.9	3.9	6.2	7.0	2.0
M-maze	4150.4	3.9	4.4	12.0	2.0
Noisy M-maze	5775.8	9.0	12.2	19.0	3.0
Ring's 9x9 maze	18222.4	39.4	66.2	82	17

Table 7.2: Training times and structure created for maze tasks.

7.8 Truck Driving

McCallum (1995) describes a truck driving domain in which a learning agent controls a vehicle travelling on a four lane highway. The agent must learn to avoid collisions with other slower vehicles. The agent has four actions, Shift-Gaze-Left, Shift-Gaze-Center, Shift-Gaze-Right, and Move-To-Gaze-Lane. All these actions except the Move-To-Gaze-Lane, are deictic actions (McCallum 1995). Time is discrete with each timestep corresponding to one half second. At each timestep vehicles progress forward according to their speed. The vehicle the learning agent controls moves at 16 metres per second and the other vehicles at 12 metres per second. The agent vehicle moves forward during lane changes.

When the agent is gazing at a lane its gaze slides along the lane until it reaches a vehicle otherwise it slides to the maximum distance (of approximately 32 metres). Slow vehicles are introduced at each time step with a probability of 0.5. Introduced vehicles are placed in each lane with equal probability. If the agent collides with a slower car, it scrapes past in the same lane and is provided with a penalty. A penalty is also provided for trying to move into the road shoulder. Using TRACA the penalties were -5. In each timestep that the agent does not collide with another vehicle, it receives a small positive reward of 0.1. Vehicles may be one of 6 colours while the road and road shoulder may appear as one of three colours.

The agent can also detect how far it is gazing. There is one broad division of gaze distance to detect if the agent's gaze is far (more than 12 metres), near (8-12 metres) or in front of the agent's nose (0-8 metres). Within each of these three divisions there is a finer grained division that indicates whether the gaze is in the far-half or near-half of the broad division. The agent can also detect if it is looking left, right or center. Finally, the agent can detect if the object it is looking at is a vehicle, the road or the road shoulder.

Since TRACA predicts percepts and not rewards it was necessary to add a *bump* percept using two bits positions to the input vector. When the agent collides with another vehicle or the road shoulder the first bit will contain 1, otherwise the second bit will contain 1. This input is essentially a reward percept. A similar percept has been used by Colombetti and Dorigo (1996). Excluding the bump percept, the size of the state space is 324, however, in practice only 123 of these states are possible.

7.8.1 Experimental Design

The agent is provided with 10,000 timesteps training in the maze environment. In TRACA the average support scheme was used with random actions selected with uniform probability. Like McCallum (1995), exploration reduced over time with decreases at set intervals. Random actions were selected as shown with the probabilities shown in the schedule in Table 7.3. Two sets of experiments were conducted using TRACA, one set using the SRS learning rule the other using the standard learning rule (see Section 4.2.7). In each set of experiments TRACA was run on the problem 6 times.

Many parameters were unchanged from previous experiments. The learning rate was 0.1 and the discount rate 0.9. The number of cases for the Cox-Stuart test for trend was 20 and for the test for noise, 10 (for both temporal and non-temporal nodes). Chains were removed if their *terminal value* rose higher than -3, for chain predictions with a negative value, or fell below 0.05, for chain predictions with a positive value. The *IF* for both temporal and non-temporal nodes was 1.01 and the *temporal threshold*, (TT), was 0.09. These parameters were selected based on several runs with various parameter settings, typically varying just the exploration schedule. However, for the experiments using the SRS learning rule, the *IF* was always applied to node ETP's rather than being applied only after a node's ETP value rose higher than its equivalent subordinates. To be consistent with prior experiments this modification was not applied to the experiments with the standard rule (and when tried it appeared to have no influence on performance). As for the gap tasks, a small constant was used as support for actions when their support was within a threshold range (see Section 4.2.13). For negative values this was -0.5, for positive it was 0.5.

Timesteps	Exploration Rate
0 - 2000	1.00
2001 - 4000	0.50
4001 - 6000	0.33
6001 - 8000	0.25
8001 - 9000	0.20
9001 - 10000	0.00

Table 7.3: Exploration rate for different timesteps during learning.

After training learning is turned off and TRACA is tested for 5000 timesteps. TRACA's result is then compared to the result obtained by McCallum (1995) for the U-Tree algorithm. A comparison is also made against flat Q-learning based on an enumeration of the state space excluding the colour attribute. This agent makes no use of memory. Two sets of control experiments are also run. One in which the agent selects random actions the other in which it is placed in a random lane where it remains for the duration of testing.

7.8.2 Results

The number of collisions for 6 runs of each of the different agents are presented in Table 7.4. The first column contains the results obtained by TRACA when using the SRS learning rule. The second column is the results TRACA obtained when using the standard learning rule (other parameters were unchanged). The third column is the results of using Flat Q-learning (however, in this case the colour attribute was excluded). The fourth column shows the results of an agent which selects entirely random actions and the fifth column the results of an agent which in each run is placed in a random column and selects no action (it never moves).

The agent selecting random actions has the highest collision rate with an average of 773 collisions. The next worst performance is by the agent which stays in the same position with an average of 621 collisions. Following this are the TRACA agents which use the standard learning rule which achieved an average of 314 collisions. The two best performers were flat Q-learning with an average of 222 collisions followed by the TRACA agent's using the SRS learning rule, with an average of 266 collisions. The best performing run for each of these

	TRACA's results		Other policies		
Run	SRS Rule	Standard Rule	Flat Q-learning	Random Action	No action
1	194	330	46	***751***	650
2	***60***	294	302	754	640
3	301	368	310	761	646
4	304	299	294	796	600
5	498	***277***	343	765	***596***
6	236	317	***39***	808	596
Average	266	314	222	773	621

Table 7.4: The comparative performance of TRACA on the truck driving task, in terms of number of collisions, to other possible policies. The lowest number of collisions obtained by each technique is highlighted in bold italics and the averages on the bottom row are rounded.

achieved 39 and 60 collisions respectively. McCallum (1995) cites the best result obtained by U-Tree on this task as 67 collisions. He also tested a hand-written policy which obtained 99 collisions. The ability of the SRS rule to achieve such a good result, relative to the standard rule, raises the question about which learning rule is most appropriate. Chapman and Kaelbling (1991) describe a rule which extends Q-learning to make additional distinctions and explain how this leads to better performance. This may also explain the difference between TRACA's SRS rule and the standard learning rule.

TRACA's representation for the best run using the SRS learning rule included 111 temporal, terminal and join groups. Only 10 of these groups were temporal groups. Forty of the join and chain groups were components in structures which depended on a colour feature. A selection of the joins and chains created which did not include a colour attribute are presented in Appendix B.2. The average number of groups created by TRACA using the SRS rule was 124.2 (standard deviation 20.9). At most 14 of these groups were temporal groups. McCallum (1995) describes the learned representation from his run as having 51 leaves and including no distinctions based on colour. Including only join groups (and not groups used in chains) TRACA's best performing run created 91 groups. An equivalent tabulated approach, such as used for Flat Q-learning, would require 123 states. However, this excludes any possible use of previous observations. In this task, the size of TRACA's representation is partially contributed to by the necessity to include the two additional bits for detecting collisions which doubles the state space size.

7.8.3 Discussion of Results

The fact that flat Q-learning achieved the best average as well as the best result for any single run suggests that the hidden-state in this task is not significant enough to prevent a good

policy being learned by a memoryless agent.[4] However, even for the two best sets of results there is considerable variation between the best performance and the average performance.

TRACA does create more structure than U-Tree. While this is partially due to the extra two input bits required by TRACA, it is primarily due to the fact that TRACA uses the colour attribute which is irrelevant given the presence of other attributes. While joins which include colour may not be necessary in a minimal solution, they can be useful in the absence of a more general rule. TRACA will create and retain any joins which provide predictive improvements until the prediction is always reliably predicted. This is exactly the behaviour desired if we wish to take maximum advantage of the agent's experience while constructing a better rule set. However, in this task it would most likely be beneficial to remove the rules based on colour once rules based on object types are created which can replace them. A possible method for removing groups containing such redundant rules is discussed in Section 8.4. On the other hand, in the real world redundant rules may be beneficial. For example, if the agent should occasionally not be able to discern the object type due to rain or other obstructions to visibility. Either way, as it stands, TRACA provides at best only a modest reduction in the internal state space than a tabulated approach would. However, the fact that TRACA's network is smaller than it otherwise could be, is due to the reuse of a number of join groups in different hierarchies. Evidence of this reuse is visible in the sample structures provided in Appendix B.2.

In terms of avoiding collisions, TRACA's best result resulted in only 60 collisions whereas U-Tree's single reported result was 67. TRACA's representation was larger than U-Tree's using 111 joins versus U-Tree's tree with 51 leaves (in which no distinctions were based on colour). Actually, given that TRACA creates internal structures based on the colour attribute, its opportunities for generalisation are few, since there are no other irrelevant attributes (and no completely irrelevant ones). TRACA does not create large amounts of temporal structure, which reflects the fact that good performance can be gained without temporal structure. When compared to the policies of random action selection and taking no action, TRACA's average number of collisions provides a substantial improvement. Using the standard learning rule, TRACA has approximately half the number of collisions than achieved by the best of these two policies, and using the SRS learning rule TRACA's result is even better.

As a final point, while flat Q-learning may perform better than TRACA on this task, in general it would be expected not to. Firstly, on this task the state space for Q-learning was reduced by eliminating the colour attribute. In addition, some tasks (such as Ring (1994)'s 9x9 maze) cannot be solved without memory, and others, such as the Monk tasks presented in Chapter 5, are represented very inefficiently using tabulated state representations.

---

[4] Sallans (2002) makes a similar observation on a version of the truck driving problem which also has faster cars.

7.9 Chapter Summary

This chapter has provided a range of experiments which demonstrate TRACA's ability to combine model-learning with policy learning for a variety of problems containing hidden-state. The experiments on the gap tasks demonstrate that TRACA has the ability to successfully remember previous observations for long periods. The noisy M-maze experiment demonstrates that TRACA can successfully learn tasks with noisy sensors and effectors while requiring less training than two other algorithms.

TRACA's ability to take advantage of its network structure and utilise message passing and additional variables allows powerful forms of bias, such as demonstrated when the precedence rate is used to select paths for extending temporal chains. Without this mechanism, agents in environments with many frequently occurring, but irrelevant, features face an enormous search problem, the scale of which will grow in proportion to the number of irrelevant features.

TRACA's results on Ring (1994)'s mazes demonstrate a number of important abilities. The 5x4 maze provides an example of TRACA finding a suitable deterministic policy, if such a policy exists, when temporal chains are not available (this could be useful if an agent is forced to act before having had enough experience to learn a better policy). When temporal chains are available, a better policy is discovered and used. The experiments using the 9x9 maze show that TRACA can construct temporal chains to represent a complex environment.

The final task of truck driving demonstrated that TRACA's temporal structures can be used in conjunction with non-temporal structures to successfully solve a non-trivial learning problem with both hidden-state and distributed sensors. However, there are open questions in relation to this task, including whether TRACA should retain redundant structures for situations where otherwise reliable rules fail and which learning rule is most appropriate for use with TRACA.

Chapter 8

Conclusion

Having presented all the components of TRACA and completed experimental comparisons with other systems this chapter now restates TRACA's relationship to some important related systems and summarises its strengths and weaknesses. This is followed by an evaluation of the thesis's contributions towards implementing situated learning agents. Finally, the envisioned future of TRACA is discussed along with the further research necessary to make progress towards that vision.

8.1 Comparison with Related Systems

While a wide range of systems addressing similar problems as TRACA were described in Chapter 2, two of those systems have a particularly close relationship to TRACA. These are Holland Style Learning Classifier Systems (LCSs) and Drescher's Schema mechanism. An alternative approach to the problems addressed by TRACA is to use utile distinctions. Examples of utile distinction algorithms include the two tree-based utile distinction algorithms: G and U-Tree. The following sections compare TRACA to the Utile distinction approach, the Schema mechanism and LCSs.

8.1.1 Utile Distinction Approaches

TRACA, the G-algorithm, and U-Tree all use statistical tests to introduce new internal states. However, both the G-algorithm and U-Tree base their tests on relative utilities received when bits are on and off. Instead of making utile distinctions, TRACA makes perceptual distinctions which allow it to create non task-specific representations. Evaluating structures based on perceptual distinctions offers the additional advantage of allowing internal model building to proceed in the absence of reinforcement feedback. Perceptual improvements also allow a common baseline for comparisons between structures. Knowing this baseline can assist a learning agent which is constructing a representation to be used for multiple tasks. Utile evaluations in this case can be complicated by the different utilities of structures in relation to different tasks.

By learning an internal model which is related to more than one task, it should be possible to reduce learning times (attempts to do this also include those of Thrun (1996) and Caruana (1997) which were mentioned in Chapter 2). The disadvantage, as argued by McCallum (1995), is that the representation may be much larger. An agent making perceptual distinctions will inevitably create rules unrelated to the task at hand. The example McCallum (1995) gives is the creation of rules to represent the fact that when looking at a red object if the agent looks away and then back at the object again it will (probably) see red again. In the truck driving domain McCallum (1995) is refering to, colour is irrelevant, given other attributes, and such rules are undesirable in the presence of rules which make use of the other attributes (in this case, the rules based on colour represent a local minima). However, for an agent performing multiple tasks, these rules may be important for other tasks required of the agent. Also, such rules might be essential for the discovery of higher order concepts required to complete some tasks. Drescher (1991)'s schema mechanism is one system which attempts to take advantage of a rich collection of such rules and relationships for knowledge about an environment, including the understanding of advanced concepts such as physical objects. Harnad (1990) makes similar arguments which may justify the presence of such rules.

8.1.2 Schema Mechanism

TRACA has several things in common with Drescher (1991)'s Schema Mechanism. Like the Schema mechanism TRACA creates structures to predict percepts, creates SRS rules and uses suppression mechanisms to facilitate learning. Both defer to more specific rules (or schemas) when actions are being selected and in both, join groups (or conjunctive results) can only be predicted once the predicted structure has been created as a predictor of something else (the chicken-or-egg problem). However, there are some substantial differences between the two approaches.

One difference is the motivation behind the two systems. The Schema mechanism is motivated to demonstrate Piaget's theories of childhood development. TRACA is motivated by the desire to create situated learning agents. In doing this TRACA builds on the theories of artificial intelligence researchers and cognitive scientists, such as those expressed in Holland, Holyoak, Nisbett, and Thagard (1986). However, it remains true that these theories have themselves been influenced by the work of Piaget. These motivational differences affect design with the result that the Schema mechanism required a Connection machine to run whereas all the experiments with TRACA were run on either a desk-top or lap-top computer.

Unlike the Schema mechanism, TRACA restricts its representation by avoiding explicit negation, only creating rules and predicting results that have 1's in their input string. However, this places some additional requirements on TRACA as described in Section 4.2.6. Another feature of TRACA's mechanism is that it assesses new structures based only on

comparisons with each new structure's immediate subordinates. This implicitly excludes any bit positions in the input string which are not included in the structure's subordinate hierarchies. On the other hand, Drescher's *marginal attribution* mechanism has simpler dynamics than TRACA's comparisons between nodes in groups. Drescher's search mechanism creates new schemas based on information on all bit positions. This means each new schema incurs higher computational overheads than TRACA's structures as each schema has an extended context and an extended result in which every item (bit position) is included.

Currently TRACA does not support an equivalent to Drescher's composite actions in which groups of rules are connected together in such a way as to create sub-routines or procedures (this issue is raised again in Section 8.4). However, like Drescher's chained schemas, TRACA's rule chaining can be utilised to find paths to goals using hypothetical look-ahead (see Holland (1990)).

Another major difference is related to the temporal chains created by TRACA. Drescher's approach to uncovering hidden-state is to create *synthetic items* to reify schema results. This method holds promise as a means of remembering a fact for an indefinite amount of time which is an important requirement for any system intended to complete complex tasks in the real world. Synthetic items also offer the potential for loops and the creation of concepts for understanding physical objects. Unlike temporal chains, where a fact is remembered for an arbitrary (but fixed) amount of time, indefinite memory of an event states that "an event X happened at some time in the indefinite past" (Holland 1990). This allows, for example, one to place their keys at a particular location and go about their daily business (which may be of different and varying duration each day) while remembering the location of their keys. Drescher's synthetic items support this type of activity. Essentially, repeated occurrences which are unexplained by immediately available inputs are explained by postulating some hidden variable in the environment which is represented by the synthetic item (a bit in an internal vector extending the vector of state variables). The setting and unsetting of the synthetic item's value is triggered by events in the environment, events which must be discovered (synthetic items were discussed along with other indefinite memory techniques in Section 2.11). The implementation of an indefinite memory mechanism in TRACA, based on synthetic items or an equivalent, is desirable and would complement the existing mechanism for temporal chains.

8.1.3 Learning Classifier Systems

TRACA has a number of similarities to Learning Classifier Systems (LCS) (Holland 1975). The use of a "don't care" symbol in an LCS rule condition allows general rules since that symbol will match any character in the corresponding position in the input string. In TRACA, joins explicitly include only relevant bit positions, allowing general rules which operate independently of the values in excluded positions.

In TRACA, rule chaining is achieved using explicit connections to subsequent rules while LCSs use indirect chaining. LCS strings produced by rule actions are added as messages to a global message list used to compare input strings to rule conditions. Using this list, actions posted by matched rules can match the conditions of other rules thereby allowing rules to be chained together. LCS rule chaining can also be used to implement short-term memory, unlike TRACA in which temporal chains are required (Robertson and Riolo 1988).[1]

Other differences in TRACA's implementation include basing the retention of new rules on comparisons with their subordinate components and the suppression of rules lower in the hierarchy to reduce search. TRACA also dynamically determines the number of rules required during learning. This offers an advantage over Michigan-style Classifier systems which typically require an a-priori determined number of rules. If the number estimated is too small, the system may be prevented from finding a solution, if it is too large, the rule base will contain redundant rules. An alternative is to use Pitt-style LCSs, however, this may not sufficiently constrain the size of the rule base (De Jong 1988). A major difference of LCSs to TRACA are LCSs' use of a Genetic Algorithm (GA). While GAs have a number of desirable properties, they often have difficulty creating rules, in which case a number of additional rule discovery operators are required. Furthermore, as the entire rule base is subject to genetic operators, useful rules (particularly in chains) may be inadvertently removed (Robertson and Riolo 1988; Booker, Goldberg, and Holland 1989; Wilson and Goldberg 1989). In place of a GA, TRACA uses random recombination of components with spatial and temporal selectivity. Suppression is used to remove subordinate structures from the search space in favour of their superior structures. Existing stable components remain unmodified and available for use in further structures as long they continue to demonstrate an improvement over their subordinates.

8.2 Evaluation of TRACA

While many of TRACA's strengths and weaknesses have been mentioned in the discussion above, a more explicit evaluation of TRACA is presented below. In most cases there are trade-offs. A weakness in one respect provides a strength or benefit in another and vice-versa. This section first lists the drawbacks incurred by TRACA's design along with the associated benefits. Following this list is another one which identifies TRACA's major strengths.

Potential Drawbacks:

- *Using perceptual distinctions rather than utile distinctions creates large representations.*

  A rich representation is required for groundedness if a learning system is to achieve the capability to perform complex tasks in the real world (Harnad 1995). Furthermore, it allows the agent the potential to learn about an environment independently of any future tasks (as done by Riolo (1991)). This can reduce the number of trials required in the environment to learn new tasks. Of course, not every relationship in the world can be modelled, and TRACA incorporates a number of techniques to reduce network size.

- *The use of SRS rules (situation-response-situation) has high combinatorics.*

  Such chaining of rules allows the use of look-ahead which can support multiple goals (as done in Section 5.7) and maximise the benefits of the agent's experience. The use of an efficient representation, such as default hierarchies, with other efficiency techniques in TRACA minimise the combinatorial problem.

- *Using binary input vectors prevents the fine granularity of representation required for continuous spaces.*

  Binary inputs can represent continuous values to arbitrary precision using coarse-coding techniques (Sutton and Barto 1998). However, this does require sufficient domain knowledge to make useful partitions for precision.[2]

- *Not making distinctions based on non-Markov rewards.*

  Because TRACA makes perceptual distinctions and not utile distinctions, it cannot make distinctions based on non-Markov rewards. For example, in a coffee-delivery problem described by Bacchus, Boutilier, and Grove (1996), an agent is rewarded for delivering coffee, but only if a request for coffee was placed at some time in the past. Typically TRACA deals with this problem by requiring a percept associated with the achievement of a reward state. For example, by placing the requirement that some perceptual feedback on the satisfaction (or dissatisfaction) with the coffee delivery is provided in addition to the reinforcement.[3] This effectively shifts the problem from the agent to the environment. A similar alternative is to use a reward sensor which allows the agent to perceive the receipt of rewards by including reward information in the agent's input vector. A reward sensor was used by Colombetti and Dorigo (1996) to train an agent on a hidden-state problem, in their case it provided information on the sign of the reward received in the previous time-step.

---

[1] Another difference is that the current implementation of TRACA uses Q-learning in place of the bucket-brigade, however, both are temporal-difference methods (Dorigo and Bersini 1994). Roberts (1993) uses Q-learning in classifier systems in place of the bucket-brigade.

[2] An experiment with TRACA using such inputs is presented in Mitchell (2002).

[3] This seems quite reasonable since humans receive this type of feedback on these tasks in the form of a smile or thank-you.

- *Detecting the usefulness of variables in isolation.*

  The requirement that the usefulness of variables can be detected in isolation is shared with both Chapman and Kaelbling (1991)'s G-algorithm and McCallum (1995)'s U-Tree. In TRACA's case, only unary or binary combinations of usefulness can be detected. Chapman and Kaelbling (1991) suggests that this problem be addressed by representing features appropriately (orthogonally) in the input vector.

Major Benefits:

- *Parallel rule execution.*

  Like LCSs, TRACA consists of a number of rules which may execute in parallel. Sets of rules can operate as sub-systems, where rules suggesting one course of action may be offset by other rules. This allows TRACA to avoid some undesirable features otherwise inherent in monolithic architectures (similar arguments are made by Mahadevan and Connell (1992)). For example, a rule that supports approaching some desirable object, may be overridden by a rule that supports the avoidance of a nearby predator. The cumulative support of rules for actions can reflect such conflicts and uncertainty.

- *SRS rules minimise required experience.*

  Reducing the number of trials in the environment is important (McCallum 1995). SRS rules in TRACA are built based on perceptual distinctions, not utile distinctions. Drescher (1991) argues that learning using utile distinctions is "infeasibly slow". Perceptual SRS rules, on the other hand, allow the development of islands of rules during learning. These islands can be connected together as learning progresses to provide paths to goals. This may require fewer trials than a process of rule development which gradually extends the fringe of a single continental rule set as useful rules (based on utilities) are developed (Drescher 1991). There are counter arguments, such as one provided by McCallum (1995) which describes an environment where it is necessary to distinguish states based on utilities (this again is an issue of non-Markov rewards). However, his discussion raises two issues. First, he states that his scenario may reasonably arise in practice, but he provides no empirical evidence on the frequency of this occuring. Secondly, his claim states that utile based distinctions are necessary for the optimal solution, however, it may be possible that satisfactory non-optimal solutions are attainable in many cases.

- *Capable of representing long temporal dependences.*

  The experimental results in Chapter 7 demonstrate that TRACA can cope well when long memories are required. U-Tree can also represent history for an arbitrary length of time, but discovering appropriate history requires a fringe for every branch of the tree.[4] The search for appropriate history features is restricted by the fringe depth. In the absence of a discernible feature within the fringe on a branch of the tree, the branch will not be extended, preventing U-Tree extending its search beyond the existing fringe. Increasing the fringe depth dramatically increases the size of U-Tree's tree which could reasonably be expected to affect U-Tree's performance and scalability. In comparison, TRACA attempts to conduct a series of depth first searches that extend down long paths looking for relevant history features. It aims to avoid exploring down all possible branches of the search tree at once (see Chapter 6), prefering instead to search a smaller set of paths to greater depth. In this process it is possible that TRACA's search extends down inappropriate paths, however, the use of additional biases like the precedence rate (as done in Section 7.5) can be used to guide TRACA's search more effectively. Calculating a precedence rate efficiently requires direct chaining as provided by SRS rules.

  In summary, TRACA's search is not restricted by a fixed size fringe, but varies according to the utility value associated with the state it is predicting. This combined with the potentially reduced number of paths being searched concurrently allows TRACA's search to extend down paths further, including more history as appropriate while using less resources.

- *A Multi-purpose system.*

  TRACA constructs its network entirely during learning. It can do this in environments that are noisy, require input generalisation and contain hidden-state (as demonstrated by the experimental results in Chapters 5 and 7). This, combined with TRACA's robustness to parameter changes (as demonstrated in Section 5.4), allows TRACA to be easily appied to a number of different tasks, minimising the demands on the system developer. Being multi-purpose is desirable for a learning agent which supports multiple tasks.

8.3 Contributions

In this thesis I have presented a new learning system for implementing situated learning agents. This system integrates a number of existing and new techniques into a novel architecture which is capable of input generalisation, representing problems with hidden-state and hypothetical look-ahead. The following summarises the contributions of this thesis:

---

[4] However, on the complex task on which McCallum (1995)'s U-Tree was demonstrated, good performance can be achieved without using any history features (Sallans 2002).

1. *Input Generalisation*
    - TRACA learns on-line and incrementally while minimising problem specific parameter tuning. This is achieved by TRACA's construction of its network representation entirely during learning. Parameter tuning can be used to improve predictive performance and reduce network size (see Section 5.4).
    - TRACA is scalable. This is achieved through a unique combination of existing techniques (such as default hierarchies) and novel techniques which allow it to exclude many inputs from individual rules. The novel techniques include the aggregation of nodes into groups and the use of suppression to represent logical NOT (see Sections 4.2.5 and 5.2).
    - TRACA can learn in the presence of both noise and irrelevant attributes (see Sections 5.2, 5.6.3 and 7.8).
    - TRACA learns quickly, requiring relatively few training examples when compared to some well-known neural network approaches. For input generalisation this is achieved primarily by having relatively independent vertices in the network (compared to shared distributed representations such as used in Back-propagation neural networks, see Section 5.2).
    - TRACA avoids problems such as catastrophic forgetting and interference (see Section 5.6). Again, this is achieved by using relatively independent structures (compared to shared distributed representations such as used in Back-propagation neural networks).

2. *Handling Hidden-State*
    - TRACA addresses several outstanding problems for constructing Markov-k memory to represent problems with hidden-state. These problems include having fixed size history windows (or "fringes" such as required by McCallum (1995)'s U-Tree) and possible exponential growth. TRACA's techniques for dealing with these problems include:
        - A depth-limited iterative depth-first search technique (see Section 6.3);
        - A mechanism to represent both homogeneous and heterogeneous regions of hidden-state (see Sections 6.2 and 7.2); and
        - Use of the standard reinforcement-learning method of discounting utilities to restrict memory (see Section 6.6.5).
    - TRACA can efficiently represent hidden-state in noisy environments and environments which contain irrelevant attributes (see Sections 7.4 and 7.5).
    - TRACA can represent hidden-state with relatively few trials. This is achieved through several mechanisms:
        - TRACA's use of independent vertices in the network (see Section 7.2);
        - Appropriate biases provided via reward functions (see Section 7.6); and

        - Additional statistics collected which can substantially reduce the search space for solutions and the size of the final solution (see Section 7.5).

3. *Hypothetical look-ahead*
    - TRACA's rules and input generalisation mechanism allow hypothetical look-ahead planning with distributed sensor schemes to efficiently represent and switch between multiple tasks (see Sections 2.15 and 5.7).

8.4 Future Work

This thesis has evaluated TRACA and compared it to other systems on a number of input generalisation and hidden-state problems. However, this evaluation was not exhaustive. Furthermore there are a number of possible extensions to TRACA that could both improve its performance and extend its capabilities.

The following sub-sections deal with issues related to the current version of TRACA as presented in this thesis then discuss some possible future extensions to TRACA.

8.4.1 Further Experiments with the Existing System

Because of the complexity of TRACA and the different environments it is intended for, the research in this thesis has been largely an empirical study. As part of this study a number of experiments were conducted in which many of TRACA's parameter settings were varied. The experimental results demonstrated that TRACA is generally robust to parameter changes, however, appropriate changes to parameters can lead to improvements in predictive performance and the amount of structure created. But the experiments examining the effects of parameter changes were not exhaustive and further investigation is needed. An outstanding problem is that some parameter settings require knowledge of the environment (e.g the reward landscape) the agent is operating in. While it is likely that performance will always be able to be improved by customising parameters, some characterisation of environments, along the lines of that conducted by Littman (1993), may assist here. A related question, which applies to reinforcement learning agents in general and not just TRACA, is: which learning rule is best? The results in this thesis were obtained primarily using Q-learning, however, a number of learning rules are possible (see Section 2.6). The comparative results on the truck driving task in Section 7.8 suggest that TRACA's SRS rule may achieve better performance on that task than the standard learning rule (see Section 4.2.7). A possible explanation for this (offered by Chapman and Kaelbling (1991)) is that the SRS rule makes better distinctions. Further analysis also needs to be conducted into the effects of (and the necessity for) sending both the maximum and minimum value of groups in support of actions (see Section 4.2.13).

Finally, TRACA needs further evaluation in relation to both its limits in the face of noise and on factors which may affect its scalability. For example, on problems which contain hidden-state and in which there is effector noise, TRACA creates multiple models. However, using this approach increasing amounts of noise lead in turn to both longer learning times and increasingly larger models. An indefinite memory mechanism as proposed in the next section may offer an alternative in these cases.

8.4.2 Possible Extensions to the Existing System

TRACA is intended as a sub-cognitive architecture for situated agents, as such it needs to support a number of higher-level activities. To achieve this a number of extensions are necessary to the existing mechanisms presented in this thesis. The following emphasises some features of TRACA that seem necessary for situated agents and offers suggestions on how they may be extended or supplemented.

One of the most important issues for learning, first raised in Chapter 1, is the importance of bias. When doing input generalisation, TRACA uses a range of techniques to: (i) bias its search for useful structures towards simple structures while avoiding complex equivalents; and (ii) select appropriate structures as the basis for creating new structures (see Chapter 4). For hidden-state the thesis has emphasised the need to direct an agent so that it frequently follows a small set of paths. The benefit of this was demonstrated on Ring's 9x9 maze in Section 7.6. A further important bias for representing hidden-state was the use of a precedence rate to eliminate irrelevant attributes from the search space when constructing temporal chains (see Section 7.5). However, the truck driving task (see Section 7.8) demonstrated that it may be desirable to have even more forms of bias. In particular, if a general rule reliably predicts a group then more specific rules, which are perhaps redundant, can be removed. One simple mechanism to implement this type of pruning in TRACA is to identify predicting nodes which have both an ETP and a precedence rate with an asymptote of 1.0. The simultaneous presence of such values in a node imply that the node represents the necessary and sufficient conditions for the matching of its predicted group. In these cases, the predicted group could send a message to other predicting nodes flagging them so that their containing groups do not retain themselves based on the performance of those nodes (since the nodes are unnecessary). This in turn may result in the removal of some of the groups containing flagged nodes. A similar mechanism could also be used for flagging nodes in unary groups (which cannot be removed) to prevent them from sending support to effectors.

TRACA uses parallel rule structures for both input generalisation and representing hidden-state. TRACA's parallel rules for input generalisation implement default hierarchies and the independence of these hierarchical rules avoids the problems of interference and catastrophic forgetting associated with Back-propagation neural networks (see Sections 5.2 and 5.6). Similarly, temporal chains may execute in parallel and their independence allows TRACA to represent long temporal dependencies (see Section 7.2). The use of independent rules may result in TRACA creating more structure than some alternatives approaches, but this independence is necessary to minimise the amount of experience required for learning (the requirement of *efficient learning* from Chapter 1). However, the greatest potential of independent rules in achieving this requirement lies in their ability to support multiple tasks. This requires some form of hypothetical look-ahead search which, in the absence of a truly parallel hardware implementation, is expensive. However, there are other possible means for improving the efficiency of hypothetical look-ahead. One possibility is to adopt a strategy such as that used in assumptive planning by Nourbakhsh, Powers, and Birchfield (1995) (as discussed in Section 3.5.4), where hypothetical look-ahead is only propagated along most likely paths with replanning performed as necessary.

While this thesis has addressed some of the issues associated with combining hypothetical look-ahead with distributed sensor schemes (see Section 5.7) a lot more remains to be done. Three potential areas of investigation include: (i) searches involving temporal chains; (ii) identifying the usefulness of structures across multiple tasks; and (iii) the use of sub-routines. So far TRACA has not been applied to tasks in which hypothetical search extends down temporal chains. In regard to the second issue, it would certainly be desirable to take into account the role of groups and nodes across many tasks rather than in relation to a single task. One possible approach to this is to distinguish task independent rewards from task dependent rewards (Drescher (1991) has a measure along these lines called *delegated value*). For example, the receipt of a bump penalty for colliding with an obstacle is independent of any task and could be included as such in the calculation of utilities during hypothetical look-ahead. Finally, there is the potential of implementing sub-routines. It seems quite likely that the full potential of combining independent structures with hypothetical search will only be realised when sub-routines can be implemented. Again, Drescher (1991) has a mechanism along these lines called *composite actions*, but many other researchers have also tackled this issue, including Lin (1993) and Dietterich (2000). It is possible that in problem domains with sub-goals, sub-routines could be used to determine limits for the depth of look-ahead addressing the problem raised in Section 4.2.14.

Like many concept learning systems, TRACA is limited to learning lower-order concepts. For example, it has no means of representing an abstract concept such as *physical object* or of manipulating abstract symbols. TRACA needs to either support or incorporate additional mechanisms, such as Drescher (1991)'s invention of synthetic items, if an understanding of higher-order concepts is to be acquired.

One final issue, not yet addressed by either TRACA or Drescher's mechanism, is the ability to deal not just with distributed sensors, but also with distributed effectors. This ability would allow the system to learn how and when to execute independent effectors concurrently.

Appendix A

Skeleton Pseudo-code for the Basic System

The following pseudo-code provides a skeleton implementation for the major-cycle of TRACA's basic system which was described in Chapter 4.

The implementation uses a number of globally accessible collections, the names of which reflect the objects they contain. It also uses a collection class that provides the method `forEach(String methodName)`, which at (pseudo) runtime dynamically invokes `methodName` on each of the elements in the collection. The collection is ordered and method `elementAt(int position)` can be used to locate an element in the collection using the position as an index.

The naming convention for global variables has words separated by hyphens, and for local variables has words concatenated with an uppercase letter for each word after the first.

```
/*****************************************************************************
// Globally accessible collections used to hold all unary groups, all join
// groups and sets of all groups depending on the values of their attributes:
    global Collection collection-of-matched-groups = new Collection();
    global Collection collection-of-unary-groups = new Collection();
    global Collection collection-of-join-groups = new Collection();
    global Collection collection-of-fired-nodes = new Collection();

// The globally accessible agent-environment interface class:
    EnvironmentInterface environment-interface = new EnvironmentInterface();

/*****************************************************************************
class Controller {

   float maxValue=0; // Assume only positive returns are possible.

   // Create a new controller, create unary groups for each detector
   // and start the major-cycle loop
    method main() {
       Controller controller = new Controller()
       controller.createUnaryGroups();
```

```
    controller.runMajorCycle();
 }

// Loop through the system's steps
 method runMajorCycle() {
    // In theory, the agent can continue learning for ever.
     while (true) {

        // use input string to match detectors and groups
         environment.getInputString();

        // Matched groups create nodes in groups matched last cycle
        // to predict them setting their action to the last system action
         collection-of-matched-groups.forEach("createPredictorNodes");

        // Fired nodes whose associated effector was selected by the system
        // execute (prepare to update ETPs and utility estimates
        // in the next cycle).
         collection-of-fired-nodes.forEach("checkExecute");

        // Join groups which are matched and unsuspended
        // support suppress their subordinates.
         collection-of-join-groups.forEach("checkSupportSuppress");

        // Matched groups pay predictor nodes using the maximum
        // value of their nodes.
        // This will update the system's maximum return for groups
        // matched by the current state. Updated by matched groups
        // and used by nodes which executed to update Qvalue estimates.
        // Nodes in groups which were support suppressed last cycle
        // do not update utility estimates.
         collection-of-matched-groups.forEach("payReturnForPredictors");

         collection-of-fired-nodes.forEach("updateValueEstimate");

        // Set all fired nodes to not-fired and remove them from
        // the list of fired groups.
         collection-of-fired-nodes.forEach("setFiredToFalse");
         collection-of-fired-nodes.removeAll();

        // Each group will get each of its nodes to check its ETP
        // values against its subordinates using the test for noise
        // if it passes the Cox-Stuart test for trend. If a node improves
        // on its subordinates it flag itself as improved.  If the group
        // is suspended, it will set it to unsuspended at the same time.
         collection-of-unary-groups.forEach("checkStatistics");
         collection-of-join-groups.forEach("checkStatistics");

         // Groups which are unsuspended and contain no nodes flagged
         // as improved remove themselves.
         collection-of-join-groups.forEach("checkForRemove");

         // Groups createSuppressed in the previous cycle
         // reset createSuppressed to false, remembering if
         // they were createSuppressed for the following step
         collection-of-unary-groups.forEach("resetCreateSuppressed");
         collection-of-join-groups.forEach("resetCreateSuppressed");
        // Superior groups which are matched this cycle create suppress
        // their subordinates.
         collection-of-matched-groups.
             forEach("createSuppressSubordinates");

        // Each matched group creates a join using as subordinates the
        // groups of two randomly selected predictor nodes. The join
        // group will contain a node to predict the creating group.
        // Groups that were create suppressed in the previous cycle
        // are excluded from being used as subordinates.
        // Groups that were create suppressed
        // in this cycle are excluded from creating new joins.
         collection-of-matched-groups.forEach("createJoinGroups");

        // Now clear the predictor-node list in each group predicted last
        // cycle.
         collection-of-matched-groups.forEach("clearPredictors");

       // Matched nodes (nodes in matched groups) set themselves to fired
       // and add themselves to the collection-of-fired-groups
       // Nodes which are not in groups which are support suppressed send
       // support to effectors
         collection-of-unary-groups.forEach("fireNodes");
         collection-of-join-groups.forEach("fireNodes");

        // The system selects an effector based on the support sent
        // by nodes
         environment.selectEffector();

        // Matched groups set themselves to unmatched
        // and are removed from the collection-of-matched-groups
         collection-of-matched-groups.forEach("resetMatched");
         collection-of-matched-groups.removeAll();

        // Groups which were support suppressed this cycle reset
        // suppressedSuppressed to false, remembering if they
        // were support suppressed for updating ETP's in the next cycle.
         collection-of-unary-groups.forEach("resetSupportSuppressed");
         collection-of-join-groups.forEach("resetSupportSuppressed");

        // Clear matched detectors and the support stored for each effector.
         environment.reset();

     } // end loop
}

// Create a unary group (at this stage without any contained nodes)
// for each detector.
 method createUnaryGroups() {
    // Get the number of detectors from the environment-interface
     int detectors = environment-interface.
                           collection-of-detectors.getCount();
     for (int index=0; index < detectors; index++) {
         UnaryGroup group = new UnaryGroup();
         group.controller = this;
         unaryGroupCollection.addElement(group);
       // add the unary group to the environment
         environment-interface.add(group, index);
```

```
        }
    }
}


/*******************************************************************************
/* This implements the environment and provides the agent's interface
/* to that environment.

class EnvironmentInterface {

   // Detectors for each bit in the fixed length input strings:
    Collection collectionOfDetectors = new Collection();
   // A effector for each possible action the system can take:
    collection collectionOfEffectors = new Collection() ;
   // The system's most recent reward:
    float systemReward;
   // The last action selected by the system
    int selectedEffectorPosition;

   // Set up environment and environment interface
    method EnvironmentInterface() {
        // set up environment from environment file
        // including effectors and the state space.
    }

   // Add a group for each detector (called by Controller on start up)
    method add(UnaryGroup group, int position) {
        collectionOfDetectors.elementAt(position).setUnaryGroup(group);
    }

   // Read the input string for the current state and using the values
   // in its bit positions match the appropriate detectors.  Detectors
   // send a message to their associated group indicating if they are
   // matched or not.  Groups then pass on messages to their superiors
   // based on whether or not all their subordinates are matched.

    method readInputString() {
        collectionOfDetectors.forEach("matchGroups");
     }

   // Select an effector based on the support received by nodes and the
   // the system selection strategy.
    method selectEffector() {
    // the selectedEffectorPosition stores the agent's selected action.
        selectedEffectorPosition = 0;
    }

   // Add the support sent by nodes to the associated effector
    method addSupport(float value, int effector) {
        collectionOfEffectors.elementAt(effector).addSupport(value);
    }

   // Re-initialise each effector to have zero support.
   // if the learning is trial based, this all gives the environment the
   // opportunity to move the agent back to its start position
   // for the next trial.
    method reset() {
        reward = 0;
        effectors.forEach("resetSupport");
    }
}

/*****************************************************************************
/* This class implements the system's detectors. There will be one detector
/* for each bit position in the fixed length input strings.

class Detector {

    UnaryGroup group;
    boolean matched = false;

   // Each detector has a unary group (initially this contains no nodes).
    method setUnaryGroup(UnaryGroup group) {
        this.group = group;
        group.detector = this;
    }

   // Detectors are matched when their corresponding bit position in the
   // current input string contains a one. When a detector is matched, so is
   // its corresponding unary group. The unary group will pass the matched
   // message up to any superiors it has, who will propagate further matched
   // messages up the network hierarchies indicating whether or not both
   // their immediate subordinate groups are matched.
    method match(boolean aBoolean) {
        matched = aBoolean;
        group.match(aBoolean);
    }
}

/*****************************************************************************
/*  Groups may be unary or joins, but both have some state and behaviour in
/*  common.

abstract class Group {

    boolean suspended = true; // all groups are initially suspended
    boolean previousSupportSuppressed = false;
    boolean supportSuppressed = false;
    boolean previousCreateSuppressed = false;
    boolean createSuppressed = false;
    boolean matched = false;

    Controller controller = null;

    Collection immediateSuperiors = new Collection();
    Collection containedNodes = new Collection();
    Collection predictorNodes = new Collection();

    float etpThreshold = 0.99;

    abstract match(boolean aBoolean);
    abstract setSupportSuppressed(boolean aBoolean);
    abstract createSuppressSubordinates();
    abstract setCreateSuppressed(boolean aBoolean);
```

```
   // Send a message to each node which checks if the node's ETP value
   // has stopped trending up, if so and the node provides an improvement
   // over its equivalent subordinates (for nodes in joins) then set the
   // node to improved (or unset it if it was improved and no longer is).
   // The first time a node is improved its containing group is set to
   // unsuspended.  Unary groups are set to unsuspended once one node
   // reaches a minimum number of trials.
    method checkStatistics() {
         containedNodes.forEach("checkStatistics");
    }

   // Remember whether this group was supported suppressed so that
   // its node's can correct update ETP values in the next cycle.
    method resetSupportSuppressed() {
        previousSupportSuppressed = supportSuppressed;
        supportSuppressed = false;
    }

   // Remember whether this group was create suppressed so that
   // it is not used to create duplicates next cycle.
    method resetCreateSuppressed() {
        previousCreateSuppressed = createSuppressed;
        createSuppressed = false;
    }

   // If the group is matched it sends a message to each of its nodes
   // telling them to fire.
    method fireNodes() {
        if (matched) {
            containedNodes.forEach("fire");
        }
    }

   // Reset matched variable for the next cycle
    method resetMatched() {
        matched = false;
    }

   // Store predicting nodes so that they can be used to create new joins
   // groups. The contents of this list is also used to prevent
   // duplicate copies of predicting nodes being created.
    method addPredictor(Node node) {
        predictorNodes.addElement(node);
    }

   // Empty the list of predictors so that joins can be created next cycle
   // from the new set of predictors.
    method clearPredictors() {
        predictorNodes.removeAll();
    }

   // Create nodes in groups which were matched in the cycle prior to
   // the current cycle to predict this group next time the prior groups
   // are matched and the same action is selected.  First check that a
   // predicting group does not already exist.
    method createPredictorNodes() {
        if (matched && !supportSuppressed) {
            Iterator iterator = collection-of-fired-nodes.getIterator();
        while (iterator.hasMoreElements()) {
            Node predictor = iterator.nextElement();
            if (!predictor.group.previousSupportSuppressed
              && predictor.effector ==
                     environment-interface.selectedEffectorPosition)
              {
                 Node node = new Node();
                 node.effector =
                     environment-interface.selectedEffectorPosition;
                 node.prediction = this;
                 predictor.group.addContainedNode(node);
              }
        }
    }
 }

// Add a new contained node to this group
 method addContainedNode(Node node) {
     containedNodes.addElement(node);
     node.group = this;
 }

// Send the maximum value of the group to nodes which predicted this
// group in the previous cycle.
 method payReturnForPredictors() {
     if (supportSuppressed)
         return;
     float maxValue = getMaxValue();
     if (maxValue > controller.maxValue)
         controller.maxValue = maxValue;
 }

// Retrieve the maximum value of the nodes in the group
// (the following assumes only positive rewards are possible)
 method float getMaxValue() {
     float max = 0;
     if (containedNodes.size() > 0) {
         max = containedNodes.elementAt(0).value;
         Iterator iterator = containedNodes.getIterator();
         while (iterator.hasMoreElements()) {
             Node node = iterator.nextElement();
             if (node.value > max)
                 max = node.value;
         }
     }
     return max;
 }

// Create a new join group based on the groups of two selected
// predicting nodes.
 method createJoinGroups() {

     if (supportSuppressed)
        return;

    // Possibly return if already accurately predicted
    // (this can be done more efficiently).
```

```
        Iterator iterator = predictorNodes.getIterator();
        while (iterator.hasMoreElements()) {
             Node node = iterator.nextElement();
            if (!node.group.suspended
                && node.value >= etpThreshold)
                return;
        }

        // create list of subordinate groups for new join
        Collection subordinateGroups = new Collection();
        Iterator iterator = predictorNodes.getIterator();
        while (iterator.hasMoreElements()) {
            Node node = iterator.nextElement();
            if (!node.group.createSuppressed
                && !node.group.suspended)
                subordinateGroups.addElement(node);
        }
        if (subordinateGroups.size() > 1) {
            Node node1 = selectRandomNode(subordinateGroups);
            Node node2 = selectRandomNode(subordinateGroups);
            node1.group.createSuppressed = true; // to prevent duplicates
            node2.group.createSuppressed = true; // to prevent duplicates
            Group group = new JoinGroup();
            group.controller = controller;
            controller.joinGroupCollection.addElement(group);

            Node node = new Node();
            node.effector =
                        environment-interface.selectedEffectorPosition;
            node.prediction = this;

            group.addContainedNode(node);
        }
    }

   // Select a group from the collection and remove it.
    method Group selectRandomGroup(Collection subordinateGroups) {
        // randomly select a group, remove it from the set and return it.
   }

}

/******************************************************************************
/* Join groups are those with 2 subordinate groups

class JoinGroup extends Group {

   // A collection to contain the group's immediate subordinates.
   // Note: there is ony only ever 2 immediate subordinates
    Collection subordinates = new Collection();

   // The number of times the group has recieved a matched message this
   // cycle.  It will receive two message each cycle - one from each
   // immediate subordinate
    int matchCount = 0;

   // Support suppress immediate subordinates, this is passed down the
   // hierarchy to the unary groups.
```

```
    method checkSupportSuppress() {
        if (matched && !suspended)
            subordinates.forEach("setSupportSuppressed", true);
    }

   // Set this group's supportSuppressed variable to the parameter value
   // and pass the message and value on to your immediate subordinates.
    method setSupportSuppressed(boolean aBoolean) {
        supportSuppressed = true;
        subordinates.forEach("setSupportSuppressed", true);
    }

   // Create suppress immediate subordinates, this is passed down the
   // hierarchy to the unary groups.
    method createSuppressSubordinates() {
        if (matched)
            subordinates.forEach("setCreateSuppressed", true);
    }

   // Set this group's createSuppressed variable to the parameter value
   // and pass the message and value on to your immediate subordinates.
    method setCreateSuppressed(boolean aBoolean) {
        supportSuppressed = true;
        subordinates.forEach("setCreateSuppressed", true);
    }

   // Receive a matched message from the immediate subordinates.  Once two
   // messages have been received determine if this group is matched or not.
    method match() {
        matchCount++;
        if (matchCount == 2) {
           if (subordinates.elementAt(0).matched
               && (subordinates.elementAt(1).matched) {
              matched = true;
              collection-of-matched-groups.addElement(this);
           }
           superiors.forEach("match");
           matchCount = 0;
        }
    }

   // If the group is unsuspended and contains no improved nodes
   // (i.e nodes which have finished trending and demonstrated an improvement
   // over their equivalent subordinates - See Sections 4.2.9 and 4.2.11)
   // remove themselves.
    method checkForRemove() {
        // compare values of nodes in this group with values of nodes in both
        // subordinate groups and remove this group, its nodes and superiors
        // if the node's provide no improvements
    }

   // Reset matched variable for the next cycle and matchedCount
    method resetMatched() {
        super.resetMatched();
        matchCount = 0;
    }
}
```

```
/******************************************************************************
/* Unary groups are connected directly to a detector

class UnaryGroup extends Group {

    Detector detector = null;

    method setSupportSuppressed(boolean aBoolean) {
        supportSuppressed = true;
    }

    method createSuppressSubordinates() {
        // do nothing for UnaryGroups.
    }

    method setCreateSuppressed(boolean aBoolean) {
        createSuppressed = true;
    }

    method match(boolean aBoolean) {
        controller.matchedGroupCollection.addElement(this);
        superiors.forEach("match");
    }
}

/******************************************************************************
class Node {

   // The group this node predicts
    Group prediction;
   // The node's supported effector
    int effector;
   // The node's estimated transition probability
    float independentEtp=0;
   // The node's utility value estimate
    float Qvalue=0;
   // The learning rate for both ETP and Qvalue updates
    float learningRate = 0.2;
   // A flag as to whether or not this node improves on its subordinates or
   // not (used if it is in a join group only).
    boolean isImproved=false;

    Group containingGroup;
    boolean fired = false;
   // The number of times this node has executed.
    int executionCount;
   // This is the number of executions a node must have before
   // sending returns to predictors.
    final int executionThreshold = 20; // or some sensible value

   // Nodes fire and send support if their containing group is not suspended
   // and not support suppressed.
   // Omitted from the following is the code that allows only the two nodes
   //  with the highest and lowest utility values in the group to send support.
    method fire() {
        fired = true;
        collection-of-fired-groups.addElement(this);
        if (!containingGroup.suspended
       && !containingGroup.supportSuppressed) {
        environment-interface.addSupport(value, effector);
 }

// Reset fired variable
 method setFiredToFalse() {
     fired = false;
 }

// If the node fired last cycle and its action was selected, the node
// executes, updating its ETP with 1 if its predicted group is matched
// and 0 if it is not (see Equation 4.1 in Section 4.2.7).
 method checkExecute() {
     float updateValue = 0;
     if (environment-interface.selectedEffectorPosition == effector) {
         // Add to list of predictor nodes for the creation of new joins
         prediction.addPredictor(this);
         if (prediction.matched)
              updateValue = 1;
         independentEtp = independentEtp +
             (learningRate * (updateValue - independentEtp));

         activationCount++;
         // also update ETP using equation in Section 4.2.7
     }
 }

// Calculate new utility value based on update value.
// This demonstrates the standard learning rule.
// See Equation 4.4, Section 4.2.7.
 method updateUtilityValue() {
     float discountRate = 0.1;

     float reward = environment-interface.systemReward;

     if (fired
        && !containingGroup.previousSupportSuppressed
        && (environment-interface.selectedEffectorPosition == effector)) {

     // calculate new utility value based on update value and reward

           Qvalue = Qvalue +
               learningRate * ((reward +
                 (discountRate * controller.maxValue)) - Qvalue);
 }

// Check if this node's value has stopped trending.  If so, then compare
// its ETP values to its subordinates (these are collected only once a
// node's ETP exceeds its subordinates, so this can only be done if
// sufficient ETP's have been collected for the test for noise.
// If it provides an improvement set the isImproved flag. If the node's
// containing group is suspended and the node is improved, set the
// group to unsuspended. If the node has stopped trending
// and no other node has demonstrated an improvement, remove the containing
// group.
 method checkStatistics() {
    // Here also include the check to see if this node has
    // improved on its subordinates before setting the group to
```

```
       // unsuspended.

        if (activationCount > activationThreshold)
            group.suspended = false;
    }
}

/******************************************************************************
/* There is one effector for each possible action.
class Effector {

    private float support = 0.0f;

   // Calculate support using one of the methods described in Section 4.2.13.
    method addSupport(float value) {
        support = support + value;
    }

    method resetSupport() {
        support = 0.0f;
    }
}

/******************************************************************************
```

Appendix B

Sample Structures Created By TRACA

B.1 Structure Created During One Run on the 9x9 Maze Problem.

The following is the set of chains created during one run on the 9x9 maze task in Section 7.6. The format for each link is *link_group_number (subordinate_group, effector)* links are listed in order from the first group to the terminal group and each link is separated by a colon. The chain prediction is indicated by the arrow (– >).

```
222 (34,2): 221 (32,0) - > 64
962 (38,1): 449 (31,0): 226 (31,0): 225 (31,0) - > 63
230 (63,0): 231 (31,0) - > 35
2254 (614,2): 1961 (48, 0): 1351 (48,0): 911 (48,0): 299 (48,0): 300 (48,0) - > 40
312 (32,0): 313 (64,0) - > 32
401 (32,0): 402 (40,2) - > 5
451 (16,1): 452 (62,0): - > 30
475 (48,0): 476 (40,2) - > 9
528 (13,3): 529 (62,0) - > 30
540 (62,0): 541 (10,0) - > 30
607 (30,2): 608 (40,2) - > 5
613 (10,0): 614 (30,2) - > 48
630 (62,0): 631 (30,2) - > 40
885 (30,2): 886 (64,1) - > 10
891 (64,1): 892 (10,0) - > 62
2173 (892,0): 895 (62,0): 471 (62,0): 472 (62,0) - > 10
1059 (285,3): 441 (32,0): 202 (32,0): 201 (32,0) - > 40
1391 (62,0): 991 (30, 0): 992 (30, 2) - > 40
2321 (30,2): 1371 (40,0): 1372 (40,2) - > 5
1492 (15, 2): 1493 (62, 0) - > 607
```

B.2 Sample Structures Created for Truck Driving Problem.

The following are extracts from TRACA's best run on the truck problem in Section 7.8. The extract demonstrates how join groups, such as those numbered 856 and 883, are reused by a number of superior structures in multiple hierarchies.

```
Rule group number: 12581
    AND (first input for 12581 not unary):
Rule group number: 8729
    AND (first input for 8729 not unary):
Rule group number: 856
    Smooth
    AND WithinAtFar
    finished first input for 8729
    AND SeeTruck
    finished first input for 12581
    AND LookingLeft
    finished second input for 12581

Rule group number: 16892
        GazingNear
        AND (second input for 16892 not unary):
Rule group number: 8729
        AND (first input for 8729 not unary):
Rule group number: 856
        Smooth
        AND WithinAtFar
        finished first input for 8729
        AND SeeTruck
        finished second input for 16892

Rule group number: 17228
        LookingLeft
        AND (second input for 17228 not unary):
Rule group number: 5388
        WithinAtFar
        AND GazingNear
        finished second input for 17228

Rule temporal group number: 20994
         AT T0: LookingLeft
                T0 Effector: ShiftGazeRight
        AND AT T1 (second input for 20994 not unary):
Rule group number: 8681
        LookingRight
        AND (second input for 8681 not unary):
Rule group number: 883
        GazingFar
        AND WithinAtFar
        finished second input for 20994

Rule group number: 24345
        AND (first input for 24345 not unary):
Rule group number: 8681
        LookingRight
        AND (second input for 8681 not unary):
Rule group number: 883
        GazingFar
        AND WithinAtFar
        finished second input for 8681
        finished first input for 24345
        AND Bumped
        finished second input for 24345

Rule group number: 34187
        AND (first input for 34187 not unary):
Rule group number: 15325
        LookingLeft
        AND (second input for 15325 not unary):
Rule group number: 8174
        Smooth
        AND WithinAtNear
        finished second input for 15325
        finished first input for 34187
        AND (second input for 34187 not unary):
Rule group number: 14310
        LookingLeft
        AND GazingNear
        finished second input for 34187

Rule group number: 41949
        AND (first input for 41949 not unary):
Rule group number: 2055
        AND (first input for 2055 not unary):
Rule group number: 856
        Smooth
        AND WithinAtFar
        finished first input for 2055
        AND (second input for 2055 not unary):
Rule group number: 883
        GazingFar
        AND WithinAtFar
        finished second input for 2055
        finished first input for 41949
        AND (second input for 41949 not unary):
Rule group number: 23111
        SeeTruck
        AND LookingLeft
        finished second input for 41949
```

Bibliography

Bacchus, F., C. Boutilier, and A. Grove (1996). Rewarding behaviours. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence*, pp. 1160–1167. AAAI Press.

Bacchus, F., C. Boutilier, and A. Grove (1997). Structured solution methods for non-markovian decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pp. 112–117. AAAI Press.

Baird, L. and A. Moore (1999). Gradient descent for general reinforcement learning. In M. Kearns, S. Solla, and D. Cohn (Eds.), *Advances in Neural Information Processing Systems 11*, pp. 968–974. MIT Press.

Bakker, B. (2002). Reinforcement learning with long short-term memory. In T. Dietterich, S. Becker, and Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems 14*, Cambridge, MA, pp. 1475–1482. MIT Press.

Baxter, J. (2000). A model of inductive bias learning. *Journal of Artificial Intelligence Research 12*, 149–198.

Becker, M., E. Kefalea, E. Mael, M. Pagel, J. Triesch, J. Vorbrueggen, R. Wuertz, S. Zadel, and C. Von der Malsburg (1999). GripSee: A gesture controlled robot for object perception and manipulation. *Autonomous Robots 6*, 203–221.

Bellman, R. (1957). *Dynamic Programming.* Princeton University Press.

Booker, L., D. Goldberg, and J. Holland (1989, September). Classifier systems and genetic algortihms. *Artificial Intelligence 40*(1-3), 235–282.

Boutilier, C. and D. Poole (1996). Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, USA, pp. 1168–1175. AAAI Press.

Bowling, M. and M. Veloso (1999). Bounding the suboptimality of reusing subproblems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Volume 2, pp. 1340–1345. Morgan-Kaufmann.

Box, G., W. Hunter, and J. Stuart (1978). *Statistics for experimenters, an introduction to design, data analysis and model building.* Wiley, New York.

Boyan, J. and A. Moore (1995). Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. Touretzky, and T. Leen (Eds.), *Advances in Neural Information Processing Systems 7*, pp. 369–376. MIT Press.

Boyen, X. and D. Koller (1998, July). Tractable inference for complex stochastic processes. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, Madison, Wisconsin, pp. 33–42.

Boyen, X. and D. Koller (1999). Exploiting the architecture of dynamic systems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pp. 313–320. AAAI Press.

Brooks, R. (1991). Intelligence without representation. *Artificial Intelligence 47*, 139–159.

Caruana, R. (1997). Multitask learning. *Machine Learning 28*, 41–75.

Cassandra, A., L. Kaelbling, and J. Kurien (1996). Acting under uncertainty: Discrete Bayesian models for mobile robot navigation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems.*

Cassandra, A., M. Littman, and N. Zhang (1997). Incremental pruning: A simple, fast exact method for partially observable Markov decision processes. In D. Geiger and P. Shenoy (Eds.), *Proceedings of the Thirteenth Conference on Uncertainty In Artificial Intelligence*, Providence, RI, USA, pp. 54–61. Morgan-Kauffman.

Chapman, D. (1991). *Vision, Instruction and Action.* MIT Press.

Chapman, D. and L. Kaelbling (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Conference on Artificial Intelligence*, Volume 2, pp. 726–731. Morgan Kaufmann.

Choi, S., D. Yeung, and N. Zhang (2001). Hidden-mode Markov decision processes. In R. Sun and C. Giles (Eds.), *Sequence Learning: Paradigms, Algorithms and Applications*, pp. 264–287. Springer-Verlag.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 183–188. AAAI Press: Morgan-Kaufmann.

Cliff, D. and S. Ross (1995). Adding temporary memory to ZCS. *Adaptive Behaviour 3*(2), 101–150.

Cohen, P. (1995). *Empirical Methods for Artificial Intelligence.* MIT Press.

Colombetti, M. and M. Dorigo (1994). Training agents to perform sequential behaviour. *Adaptive Behaviour 2*(3), 247–275.

Colombetti, M. and M. Dorigo (1996, June). Behaviour analysis and training – a methodology for behaviour engineering. *IEEE Transactions on Systems, Man and Cybernetics – Part B: Cybernetics 26*(3), 365–380.

Daniel, W. (1990). *Applied Non-parameteric Statistics* (2nd ed.). Boston: PWS-Kent.

Davis, L., S. Wilson, and D. Orvosh (1993). Temporary memory for examples can speed learning in a simple adaptive system. In J. Meyer, H. Roitblat, and S. Wilson (Eds.), *From Animals to Animats 2*, USA, pp. 313–320. Second International Conference on Simulation of Adaptive Behaviour: MIT Press.

Dawkins, R. (1976). Hierarchical organisation: a candidate principle for ethology. In P. Bateson and R. Hinde (Eds.), *Growing Points In Ethology*, Chapter 1, pp. 7–54. Cambridge University Press.

De Jong, K. (1988). Learning with genetic algorithms: An overview. *Machine Learning 3*, 121–138.

Dempster, A., N. Laird, and D. Rubin (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society 39*(1), 1–38.

Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Artificial Intelligence Research 13*, 227–303.

Dorigo, M. (1995). ALECSYS and the Autonomouse: Learning to control a real robot by distributed classifier systems. *Machine Learning 19*, 209–240.

Dorigo, M. and H. Bersini (1994). A comparison of Q-learning and classifier systems. In D. Cliff, P. Husbands, J. Meyer, and S. Wilson (Eds.), *From Animals to Animats 3*, pp. 248–255. Third International Conference on Simulation of Adaptive Behaviour: MIT Press.

Dorigo, M. and M. Colombetti (1994). Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence 71*, 321–370.

Drescher, G. (1991). *Made-Up Minds: A constructivist approach to Artificial Intelligence.* MIT Press.

Elman, J. (1990). Finding structure in time. *Cognitive Science 14*, 179–211.

Fahlman, S. (1988a). An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie Mellon University, USA.

Fahlman, S. (1988b). Faster-learning variations on back-propagation: An empirical study. In D. Touretzky, G. Hinton, and T. Sejnowski (Eds.), *Proceedings of the 1988 Connectionist Models Summer School*, pp. 38–51. Morgan-Kaufmann.

Fahlman, S. (1991). The recurrent cascade-correlation architecture. In D. Touretzky (Ed.), *Advances in Neural Information Processing Systems 3*, pp. 190–196. Morgan Kaufmann, Los Altos CA.

Fahlman, S. and C. Lebiere (1990). The cascade correlation learning architecture. In *Advances in Neural Information Processing Systems 2*, pp. 524–532. Morgan Kaufmann.

Foner, L. and P. Maes (1994). Paying attention to what's important: Using focus of attention to improve unsupervised learning. In D. Cliff, P. Husbands, J. Meyer, and

S. Wilson (Eds.), *From Animals to Animats 3*, pp. 256–265. Third International Conference on Simulation of Adaptive Behaviour: MIT Press.

French, R. (1999). Catastrophic forgetting in connectionist networks: Causes, consequences and solutions. *Trends in Cognitive Sciences 3*(4), 128–135.

Glickman, M. and K. Sycara (2001). Evolutionary search, stochastic policies with memory and reinforcement learning with hidden-state. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pp. 194–201.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley.

Hansen, E. (1998). Solving POMDPs by searching in policy space. In *Fourteenth International Conference on Uncertainly in Artificial Intelligence*, Madison, Wisconsin, pp. 211–219. UAI-98.

Harnad, S. (1990). The symbol grounding problem. *Physica D 42*, 335–346.

Harnad, S. (1995). Grounding symbolic capacity in robotic capacity. In L. Steels and R. Brooks (Eds.), *The Artificial Life Route to Artificial Intelligence. Building Embodied, Situated Agents*, Chapter 2, pp. 277–286. Lawrence Erlbaum Associates.

Henery, R. J. (1994). Classification. In D. Michie, D. Spiegelhalter, and C. Taylor (Eds.), *Machine Learning, Neural and Statistical Classification*, Chapter 2, pp. 6–16. Ellis-Horwood.

Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. *Neural Computation 9*(8), 1735–1780.

Hoey, J., R. St-Aubin, A. Hu, and C. Boutilier (2000, June). Optimal and approximate stochastic planning using decision diagrams. Technical Report TR-00-05, University of British Columbia.

Holland, J. (1975). *Adaption in natural and artificial systems.* University of Michigan Press.

Holland, J. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. Michalski, J. Carbonell, and T. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Volume II, pp. 593–623. Kaufmann.

Holland, J. (1990). Concerning the emergence of tag-mediated lookahead in classifier systems. *Physica D 42*, 188–201.

Holland, J., K. Holyoak, R. Nisbett, and P. Thagard (1986). *Induction. Processes of inference, learning and discovery.* The MIT Press.

Holte, R. (1993). Very simple classification rules perform very well on most commonly used datasets. *Machine Learning 11*, 63–91.

Howard, R. (1971). *Dynamic Probabilistic Systems*, Volume 1, Markov Models. New York: John Wiley and Sons.

Jaakkola, T., S. Singh, and M. Jordan (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen (Eds.), *Advances in Neural Information Processing Systems 7*, pp. 345–352. MIT-Press.

Kaelbling, L. (1993). *Learning in Embedded Systems.* MIT Press.

Kaelbling, L., L. Littman, and A. Moore (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research 4*, 237–285.

Karlsson, J. (1997). *Learning to Solve Multiple Goals.* Ph. D. thesis, Department of Computer Science, University of Rochester, New York, USA.

Koenig, S. and R. Simmons (1996). The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning 22*, 227–250.

Koller, D. and R. Parr (2000). Policy iteration for factored MDPs. In *Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence*, pp. 326–334. UAI-2000.

Korf, R. (1987). Planning as search: a quantitative approach. *Artificial Intelligence 33*, 65–88.

Laird, P. and R. Saul (1994). Automated feature extraction for supervised learning. In *IEEE World Congress on Computational Intelligence*, Volume II of *Proceedings of the First IEEE Conference on Evolutionary Computation*, USA, pp. 674–679. IEEE.

Langley, P. (1996). Order effects in incremental learning. In P. Reimann and H. Spada (Eds.), *Learning in Humans and Machines: Towards an Interdisciplinary Learning Science*, pp. 154–167. Elsevier.

Lin, L. (1993). *Reinforcement learning for robots using neural networks.* Ph. D. thesis, School of Computer Science, Carnegie Mellon University, Pittburgh USA.

Lin, L. and T. Mitchell (1992). Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, Carnegie Mellon University, USA.

Lin, L. and T. Mitchell (1993). Reinforcement learning with hidden states. In J. Meyer, H. Roitblat, and S. Wilson (Eds.), *From Animals to Animats 2*, USA, pp. 271–280. Second International Conference on Simulation of Adaptive Behaviour: MIT Press.

Lin, L.-J. (1992). Self improving reactive agents based on reinforcement learning. *Machine Learning 8*, 293–321.

Littman, M. (1993). An optimization-based categorization of reinforcement learning environments. In J. Meyer, H. Roitblat, and S. Wilson (Eds.), *From Animals to Animats 2*, USA, pp. 262–270. Second International Conference on Simulation of Adaptive Behaviour: MIT Press.

Littman, M. (1994a). Memoryless policies: Theoretical limitations and practical results. In D. Cliff, P. Husbands, J. Meyer, and S. Wilson (Eds.), *From Animals to Animats 3*, pp. 238–245. Third International Conference on Simulation of Adaptive Behaviour: MIT Press.

Littman, M. (1994b, December). The witness algorithm: Solving partially observable Markov decision programs. Technical Report CS-94-40, Brown university, Department of Computer Science, Providence, RI, USA.

Littman, M., A. Cassandra, and L. Kaelbling (1995). Learning policies for partially observable environments: Scaling up. In A. Preiditis and S. Russel (Eds.), *Machine Learning: Proceedings of the Twelfth International Conference*, pp. 362–370. Morgan Kauffman.

Loch, J. and S. Singh (1998). Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pp. 323–331.

Lovejoy, W. (1991). A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research 28*, 47–66.

Lusena, C., L. Tong, S. Sittinger, C. Wells, and J. Goldsmith (1999). My brain is full: When more memory helps. In *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence*, pp. 374–381. UAI-99.

Maclin, R. and J. Shavlik (1996). Creating advice-taking reinforcement learners. *Machine Learning 22*(1-3), 251–281.

Maes, P. (1991). A bottom-up mechanism for behaviour selection in an artificial creature. In J. Meyer and S. Wilson (Eds.), *From Animals to Animats*. pp. 238–246. First International Conference on Simulation of Adaptive Behavior: MIT Press.

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning 22*, 159–195.

Mahadevan, S. and J. Connell (1992). Automatic programming of behaviour-based robots using reinforcement learning. *Artificial Intelligence 55*(2-3), 311–365.

Markovitch, S. and P. Scott (1993). Information filtering: Selection mechanisms in learning systems. *Machine Learning 10*, 113–151.

Matarić, M. (1994). Reward functions for accelerated learning. In *International Conference on Machine Learning*, pp. 181–189.

Matarić, M. (1997). Behaviour based control: Examples from navigation, learning and group behaviour. *Journal of Experimental and Theoretical Artificial Intelligence 9*(2-3), 323–336. Special issue on Software Architectures for Physical Agents.

McCallum, A. (1995). *Reinforcement Learning With Selective Perception and Hidden State*. Ph. D. thesis, Department of Computer Science, University of Rochester, NY.

McCallum, A. (1997, November). Efficient exploration in reinforcement learning with hidden state. In *Working notes of the AAAI Fall Symposium on Model-directed Autonomous Systems*.

McCallum, A, R. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Machine Learning Conference*, Amherst, pp. 190–196.

McCloskey, M. and N. Cohen (1989). Catastrophic interference in connectionist networks: The sequential learning problem. *The Psychology of Learning and Motivation 24*, 109–164.

Meuleau, N., K.-E. Kim, L. Kaelbling, and A. Cassandra (1999). Solving POMDPs by searching the space of finite policies. In *Proceedings of the Fifteenth Annual Conference on Uncertainty in AI*. UAI-99.

Millān, J. (1994). Learning efficient reactive behavioural sequences from basic reflexes in a goal directed autonomous robot. In D. Cliff, P. Husbands, J. Meyer, and S. Wilson (Eds.), *From Animals to Animats 3*, pp. 266–274. Third International Conference on Simulation of Adaptive Behaviour: MIT Press.

Mitchell, M. (2002). An evaluation of TRACA's generalisation performance. Technical Report 2002/127, School of Computer Science and Software Engineering, Monash University. Available at http://www.csse.monash.edu.au/publications.

Mitchell, T. (1990). The need for biases in learning generalizations. In J. Shavlik and T. Dietterich (Eds.), *Readings in Machine Learning*, pp. 184–191. Morgan-Kaufmann.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill.

Monahan, G. (1982). A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science 28*(1), 1–16.

Moore, A., L. Baird, and L. Kaelbling (1999). Multi-value-functions: Efficient automatic action hierarchies for multiple goal MDPs. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Volume 2, pp. 1316–1321. Morgan-Kaufmann.

Moore, W. and C. Atkeson (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning 13*, 103–130.

Mozer, M. (1992). Induction of multiscale structure. In J. Moody, S. Hanson, and R. Lippmann (Eds.), *Advances in Neural Information Processing Systems 4*, San Mateo, California, pp. 275–282. Morgan Kaufmann Publishers.

Mozer, M. and J. Bachrach (1991). SLUG: a connectionist architecture for inferring the structure of finite-state environments. *Machine Learning 7*, 139–160.

National Institute of Standards and Technology (2001). *Dataplot Reference Manual, NIST Handbook Number 148*. Gaithersbury, MD: National Institute of Standards and Technology. http://www.itl.nist.gov/div898/software/dataplot/document.html.

Neal, R. (1990, November). Learning stochastic feedforward networks. Technical Report CRG-TR-90-7, Department of Computer Science, University of Toronto. Available at: www.cs.toronto.edu/~radford/.

Newell, A. (1990). *Unified Theories of Cognition.* Harvard University Press.

Nikovski, D. and I. Nourbakhsh (1999, November). Learning discrete Bayesian models for autonomous agent navigation. In *Proceedings of IEEE Conference on Computational Intelligence in Robotics and Automation*, pp. 137–143. CIRA '99.

Nilsson, N. (1995). Eye on the prize. *AI Magazine 16*(2), 9–17.

Nourbakhsh, I., R. Powers, and S. Birchfield (1995). DERVISH an office navigating robot. *AI Magazine 16*(2), 53–60.

Papavassiliou, V. and S. Russell (1999). Convergence of reinforcement learning with general function approximators. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Stockholm, pp. 748–757.

Parr, R. and S. Russell (1995). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Montreal, Canada, pp. 1088–1095.

Peng, J. and R. Williams (1993). Efficient learning and planning within the dyna framework. In J. Meyer, H. Roitblat, and S. Wilson (Eds.), *From Animals to Animats 2*, USA, pp. 281–290. Second International Conference on Simulation of Adaptive Behaviour: MIT Press.

Peshkin, L., N. Meuleau, and L. P. Kaelbling (1999). Learning policies with external memory. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 307–314. Morgan Kaufmann.

Pineau, J. and S. Thrun (2002). An integrated approach to hierarchy and abstraction. Technical Report CMU-RI-02-21, Carnegie-Mellon University.

Rabiner, L. (1989). A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE 77*(2), 257–286.

Randløv, J. and P. Alstrøm (1998). Learning to drive a bicycle using reinforcement learning and robot shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pp. 463–471. Morgan-Kaufmann.

Ring, M. (1994). *Continual learning in reinforcement environments.* Ph. D. thesis, The University of Texas at Austin.

Riolo, R. (1989). The emergence of coupled sequences of classifiers. In J. Grefenstette (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, California, pp. 256–264. Morgan Kaufmann.

Riolo, R. (1991). Lookahead planning and latent learning in a classifier system. In J. Meyer and S. Wilson (Eds.), *From Animals to Animats*, pp. 316–326. First International Conference on Simulation of Adaptive Behaviour: MIT Press.

Rivest, R. and R. Schapire (1993). Inference of finite automata using homing sequences. *Information and Computation 103*, 299–347.

Rivest, R. and R. Schapire (1994, May). Diversity based inference of finite automata. *Journal of the Association for Computing Machinery 41*(3), 555–589.

Roberts, G. (1993). Dynamic planning for classifier systems. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms*, California, pp. 231–237. Morgan-Kaufmann.

Robertson, G. and R. Riolo (1988). A tale of two classifier systems. *Machine Learning 3*, 139–160.

Rodríguez, A., R. Parr, and D. Koller (1999). Reinforcement learning using approximate belief states. In M. Kearns, S. Solla, and D. Cohn (Eds.), *Advances in Neural Information Processing Systems 11*, pp. 1036–1042. MIT Press.

Rosenblatt, K. and D. Payton (1989). A fine-grained alternative to the subsumption architecture for mobile robot control. In *Proceedings of the IEEE/INNS International Joint Conference on Neural Networks*, pp. 317–324.

Roy, N. (2000, May). Finding approximate POMDP solutions through belief compression. Carnegie-Mellon University. Thesis Proposal.

Sallans, B. (2002). *Reinforcement Learning for Factored Markov Decision Processes.* Ph. D. thesis, Graduate Department of Computer Science, University of Toronto.

Schmidhuber, J., J. Zhao, and M. Wiering (1997). Shifting inductive bias with success-story algorithm, adaptive levin search and incremental self-improvement. *Machine Learning 28*, 105–130.

Shatkay, H. and L. Kaelbling (1997). Learning topological maps with weak odometric information. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 920–929. IJCAI-97.

Shu, L. and J. Schaeffer (1991). HCS: Adding hierarchies to Classifier Systems. In R. Belew and L. Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms*, California, pp. 339–345. Morgan Kaufmann.

Simmons, R. and S. Koenig (1995, August). Probabilistic robot navigation in partially observable environments. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Montreal, Canada, pp. 1080–1087.

Simon, H. (1981). *The Sciences of The Artificial* (2nd ed.). MIT Press.

Singh, S., T. Jaakkola, and M. Jordan (1994). Learning without state-estimation in partially observable Markovian decision processes. In W. Cohen and H. Hirsh (Eds.), *Machine Learning: Proceedings of the Eleventh International Conference*, pp. 284–292. Morgan-Kauffman.

Singh, S., T. Jaakkola, M. Littman, and C. Szepesvári (2000). Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning 38*(3), 287–308.

Smallwood, R. and E. Sondik (1973). The optimal control of partially observable Markov processes over a finite horizon. *Operations Research 21*, 1071–1088.

Sondik, E. (1973, March-April). The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research 26*(2), 283–304.

Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning 3*, 9–44.

Sutton, R. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning: Proceedings of the Seventh International Conference*, pp. 216–224. Morgan Kauffman.

Sutton, R. (1991a). Dyna, an integrated architecture for learning, planning and reacting. In *Working Notes of the AAAI Spring Symposium*, pp. 151–155.

Sutton, R. (1991b). Reinforcement learning architectures for animats. In J. Meyer and S. Wilson (Eds.), *From Animals to Animats*, pp. 288–296. First International Conference on Simulation of Adaptive Behaviour: MIT Press.

Sutton, R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pp. 1038–1044. MIT Press.

Sutton, R. and A. Barto (1998). *Reinforcement Learning: An introduction.* MIT Press.

Tan, M. (1991). Learning a cost sensitive internal representation for reinforcement learning. In L. Birnbaum and G. Collins (Eds.), *Proceedings of the Eighth International WorkShop on Machine Learning*, pp. 358-362. Morgan-Kaufmann.

Tenenberg, J., J. Karlsson, and S. Whitehead (1993). Learning via task decomposition. In J. Meyer, H. Roitblat, and S. Wilson (Eds.), *From Animals to Animats 2*, USA, pp. 337-343. Second International Conference on Simulation of Adaptive Behaviour: MIT Press.

Thrun, S. (1994). A lifelong learning perspective for mobile robot control. In *Proceedings of the IEEE/RSJ/GI Conference on Intelligent Robots and Systems.*

Thrun, S. (1996). *Explanation-based Neural Network Learning: A Lifelong Learning Approach.* Kluwer.

Thrun, S., J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. D. Jong, S. Dzeroski, S. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R. Michalski, T. Mitchell, P. Pachowicz, B. Roger, H. Vafaie, W. V. de Velde, W. Wenzel, J. Wnek, and J. Zhang (1991, December). The MONK's problems. A performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, Carnegie Mellon University. http://www-2.cs.cmu.edu/~thrun/papers/thrun.MONK.html.

Thrun, S., K. Möller, and A. Linden (1991). Planning with an adaptive world model. In D. Touretzky and R. Lippmann (Eds.), *Advances in Neural Information Processing Systems 3*, pp. 450–456. NIPS.

Thrun, S. B. (1992). The role of exploration in learning control. In D. A. White and D. A. Sofge (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, Florence, Kentucky. Van Nostrand Reinhold.

Tsitsiklis, J. and B. Van Roy (2002). On average versus discounted reward temporal-difference learning. *Machine Learning 49*(2), 179–191.

Tyrell, T. (1993). The use of hierarchies for action selection. In J. Meyer, H. Roitblat, and S. Wilson (Eds.), *From Animals to Animats 2*, USA, pp. 138–147. Second International Conference on Simulation of Adaptive Behaviour: MIT Press.

Watkins, C. (1989). *Learning From Delayed Rewards.* Ph. D. thesis, Cambridge University.

Watkins, C. and P. Dayan (1992). Technical note, Q-learning. *Machine Learning 8*, 279–292.

Werger, B. and M. Matarić (1996). Robotic "food" chains: Externalization of state and program for minimal-agent foraging. In P. Maes, M. Matarić, J. Meyer, J. Pollack, and W. S.W (Eds.), *From Animals to Animats 4*, pp. 553–561. Fourth International Conference on Adaptive Behaviour: MIT Press.

Whitehead, S. (1989). Thesis proposal: Scaling reinforcement learning systems. Technical Report 304, Computer Science, University of Rochester.

Whitehead, S. and D. Ballard (1991). Learning to perceive and act by trial and error. *Machine Learning 7*, 45–83.

Whitehead, S. and L. Lin (1995). Reinforcement learning in non-Markov environments. *Artificial Intelligence 73*(1-2), 271–306.

Wiering, M. and J. Schmidhuber (1997). HQ-learning. *Adaptive Behaviour 6*(2), 219–246.

Wiering, M. and J. Schmidhuber (1998). Efficient model-based exploration. In R. Pfeiffer, J. Blumberg, S. Meyer, and W. Wilson (Eds.), *From Animals to Animats 5*, pp. 177-182. Fifth International Conference on Simulation of Adaptive Behaviour: MIT Press.

Williams, R. and J. Peng (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation 4*, 490–501.

Williams, R. and D. Zipser (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation 2*, 270–280.

Wilson, S. (1991). The animat path to AI. In J. Meyer and S. Wilson (Eds.), *From Animals to Animats*, pp. 15–21. First International Conference on Simulation of Adaptive Behaviour: MIT Press.

Wilson, S. and D. Goldberg (1989). A critical review of classifier systems. In J. Grefenstette (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, California, pp. 245–255. Morgan Kaufmann.

Yates, D. and A. Fairley (1993). An investigation into possible causes of, and solutions to, rule strength distortion due to the bucket-brigade algorithm. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 247–253. Morgan-Kaufman.

Zubek, V. and T. Dietterich (2000). A POMDP approximation algorithm that anticipates the need to observe. In R. Mizoguchi and J. Slaney (Eds.), *Sixth Pacific Rim International Conference on Artificial Intelligence*, Melbourne, Australia, pp. 521–532. PRICAI 2000: Springer.