Symbolic Performance & Learning In Continuous-valued Environments

by

Seth Olds Rogers

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1997

Doctoral Committee:

Associate Professor John Laird, Co-Chair
Assistant Research Scientist Paul Nielsen, Co-Chair
Professor Daniel Koditschek
Assistant Professor Thad Polk

ABSTRACT

Symbolic Performance & Learning
In Continuous-valued Environments

by
Seth Olds Rogers

Co-Chairs: John Laird and Paul Nielsen

Real-world and simulated real-world domains, such as flying and driving, commonly have the characteristics of continuous-valued (CV) environments. These environments are frequently complex and difficult to control, requiring a great deal of specific, detailed knowledge. Although past approaches to learning control policies employed various forms of numerical processing, symbolic agents can also perform and learn in CV environments. There are both functional and theoretical motivations for choosing symbolic processing. SPLICE (Symbolic Performance & Learning In Continuous-Valued Environments) is a symbolic agent for adaptive control implemented in the Soar architecture. SPLICE uses a three-level framework to first classify its sensory information into symbolic regions, then map the set of regions to a local model, then use the local model to determine an action that the agent predicts will achieve the goal in the current situation. The agent monitors the results of the action and learns by changing its action mapping and local models. Learning causes the models to gradually become more specific and more accurate. SPLICE is suited to complex dynamic environments because its incremental learning ensures the response time will not increase as the agent gains more experience, and it can be proven to achieve any goal for any environment that meets certain assumptions. The performance of the SPLICE agent is evaluated in simple domains using four experimental methodologies. SPLICE is shown to exhibit flexibility in learning a variety of domains. Lesion studies show that SPLICE's most effective capabilites are generalization and constructing local linear response functions. SPLICE's performance is compared to other adaptive control algorithms and found to behave comparably. We also instantiate SPLICE in a complex environment, simulated flight, and evaluate its performance. The final goal is not only to create an effective controller for complex continuous environments, but also to understand more clearly the ramifications of symbolic learning in continuous domains.

© Seth Olds Rogers 1997
All Rights Reserved

To my family, especially Peter,
who doesn't even like Barney anymore

ACKNOWLEDGEMENTS

The person most responsible (in a good sense) for this dissertation is my advisor John. I will always be grateful to him for giving me a chance and letting me have the freedom to explore what I found interesting. I think this led to a very different research direction than what he initially had in mind. I would also like to thank my other committee co-chair Paul. He was always available for consultations and always enthusiastic about my ideas. As for my other committee members, Dan was immensely helpful in linking this work to his and related fields. Although outside of his specific area, his interest was very encouraging. Finally, I would like to thank Thad for agreeing to jump in at the last moment and allow me to present this thesis to a complete committee, providing yet another viewpoint for evaluating and understanding my work. I am also very grateful to Colleen Seifert for her service on the committee, and I am sorry that circumstances prevented her from participating in the final moments.

I found the Soar community to be an unbelievably positive resource, both for advice and support. I particularly want to thank Craig Miller for developing SCA, a critical component of SPLICE, Scott Huffman, for first suggesting encoding numbers by breaking them apart, and Doug Pearson and Randy Jones for innumerable helpful advice and tutelage. It has been a pleasure working with the University of Michigan Soar group: Sayan, Ron, Clare, Karen, Peter, Mike, Frank, Joe, Kurt, Mike, Chris, and Bob. The friendly and supportive atmosphere at the lab made the daily research grind almost enjoyable at times. Finally, it has been my privilege to meet and get to know the rest of the Soar community at the Soar workshops and occasional conferences. I have known so many good people over the years who have gone out of their way to help me, I only regret I do not have the space or energy to list them all here.

On a personal note, I would like to thank my old friend Chris Edwards for many intellectually stimulating e-mail messages and interesting distractions from my single-minded assault on my research problem. Of course, I also appreciate my Mom and Dad's unshakeable faith and confidence in me, even when they had not the slightest idea what I was trying to do. Most importantly, thanks to my beautiful wife Yusong and son Peter who made all my effort worthwhile.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

LIST OF APPENDICES xi

CHAPTERS

1 Introduction 1
- 1.1 Agent-based Interaction 2
- 1.2 Continuous-valued Environments 3
- 1.3 Representation Styles 5
- 1.4 Overview of SPLICE 7

2 Background 9
- 2.1 Control: Performance in a Continuous-valued Environment 9
- 2.2 Symbolic Control 10
- 2.3 Adaptive Control: Adjusting to the Environment 11
- 2.4 Symbolic Adaptive Control 13
- 2.5 Summary 14

3 Performance 15
- 3.1 Map to Symbols 16
- 3.2 Retrieve Best Case 17
  - 3.2.1 Match to Case Database 18
  - 3.2.2 The Domain Model 19
- 3.3 Evaluate for Current Problem 20
- 3.4 Knowledge Requirements 23

4 Learning 25
- 4.1 Creating Cases 25
- 4.2 The False Sense of Accomplishment Case 27
- 4.3 The Goal-directed Case 29
  - 4.3.1 Expectation Failures 29
  - 4.3.2 Achievement Failures 30

4.4 Choosing the Action 30
4.5 Proof of Convergence 33
4.5.1 Environmental Assumptions 33
4.5.2 Time Behavior of SPLICE 33
4.5.3 Proof 33
4.5.4 Relaxing Assumptions 34
4.6 Knowledge Requirements 34
4.7 Properties of the Learning Component 35

5 Analysis of SPLICE Execution Properties 36
5.1 Artificial Domains 36
5.1.1 A Linear Domain 37
5.1.2 A Non-linear Domain 37
5.1.3 A Discontinuous Domain 38
5.2 Experimental Methodology 38
5.3 SPLICE Results 43
5.3.1 Random Methodologies 43
5.3.2 Repeated Methodologies 52
5.4 Lesion Studies 55
5.4.1 Generalization Capability 57
5.4.2 Serendipity Capability 58
5.4.3 Point Selection Heuristics 60
5.4.4 Interpolation Capability 63
5.5 Alternative Design Studies 63
5.5.1 Abstraction Order 63
5.5.2 Feature Selection 65

6 Performance Comparison with Other Algorithms 70
6.1 Algorithms for Comparison 70
6.1.1 PID Control 71
6.1.2 Spline Fitting 71
6.1.3 Receptive Field Weighted Regression 72
6.2 Results 73
6.2.1 The Linear Domain 74
6.2.2 The Non-linear Domain 77
6.2.3 The Discontinuous Domain 80
6.3 Discussion 83

7 Experiments in a Complex Domain 86
7.1 The Flight Simulator Domain 86
7.2 Controlling Speed 87
7.3 Controlling Speed and Altitude 89
7.4 Discussion 91

8 Discussion and Extensions 93
8.1 Contribution of Soar to Functionality 93
8.2 Multiple Levels of Control 94
8.3 Region Learning and Partitioning 95

8.4 Relaxing the Monotonicity Constraint 96
8.5 Simultaneous Goals and Multiple Effectors 96
8.6 Maintenance Goals 97
8.7 Generalizing Over Time 97
8.8 Acquiring and Revising the Domain Model 98

9 Conclusion 99
9.1 Motivation Summary 99
9.2 Approach Summary 99
9.3 Results Summary 100
9.4 Contributions 100

APPENDICES 103

BIBLIOGRAPHY 129

LIST OF TABLES

Table

3.1 The RETRIEVE-BEST-CASE algorithm 18

3.2 Sample case database for the driving example. 19

5.1 Early case database. 48

5.2 Relative error in response surface before and after training. 52

A.1 Final case database for the trace. 120

LIST OF FIGURES

Figure

1.1 The interaction between agent and environment. 2
1.2 State variables, desired variables, and deadlines in continuous-valued environments. 5
1.3 Overview of SPLICE. 7
2.1 The Air-Soar control hierarchy. 10
2.2 The three variable chains in Air-Soar. 11
2.3 Characterizing systems along three dimensions. 14
3.1 Performance arc of SPLICE. 16
3.2 Region subdivision example. 17
3.3 Qualitative model of a heat exchanger. 20
3.4 Sample translation and interpolation. 22
3.5 Graphical depiction of the control heuristics. 23
4.1 Overview of SPLICE with learning added. 26
4.2 Illustration of Most General Difference. 28
4.3 Change in response surface upon learning a new point. 28
4.4 Detecting nonmonotonic trends. 30
4.5 Point selection heuristics. (left) Points failing the delete filters. (right) Computing the residuals. 31
4.6 Progress toward achieving a goal. 32
5.1 True response surface of Domain 1 (linear). 37
5.2 True response surface of Domain 2 (non-linear). 38
5.3 True response surface of Domain 3 (discontinuous). 38
5.4 Playing golf by the **random-hard** rules. 41
5.5 Playing golf by the **random-soft** rules. 41
5.6 Playing golf by the **repeated-hard** rules. 42
5.7 Playing golf by the **repeated-soft** rules. 42
5.8 The response surface methodology. 43
5.9 Initial and desired speeds for trial 1. 44
5.10 Initial performance for the linear domain, **random-hard** methodology. . . 45
5.11 Intermediate performance for the linear domain, **random-hard** methodology. 45
5.12 Final performance for the linear domain, **random-hard** methodology. . . . 46
5.13 Early case database for the linear domain, **random-hard** methodology. . . 46
5.14 Intermediate case database for the linear domain, **random-hard** methodology. 47
5.15 Final case database for the linear domain, **random-hard** methodology. . . 47

5.16 Learning curve for the **random-hard** methodology. 49
5.17 Initial performance for the linear domain, **random-soft** methodology. . . . 49
5.18 Intermediate performance for the linear domain, **random-soft** methodology. 50
5.19 Final performance for the linear domain, **random-soft** methodology. 50
5.20 Learning curve for the **random-soft** methodology. 51
5.21 The sequence of speed goals for driving to work. 52
5.22 Initial performance for the linear domain, **repeated-hard** methodology. . . 53
5.23 Final performance for the linear domain, **repeated-hard** methodology. . . 54
5.24 Final case database for the linear domain, **repeated-hard** methodology. . . 54
5.25 Learning curve for the **repeated-hard** methodology. 55
5.26 Initial performance for the linear domain, **repeated-soft** methodology. . . 56
5.27 Final performance for the linear domain, **repeated-soft** methodology. . . . 56
5.28 Learning curve for the **repeated-soft** methodology. 57
5.29 Final case database for the **repeated-soft** methodology, generalization lesion. 58
5.30 Learning curve for the **random-hard** methodology, generalization lesion. . 59
5.31 Learning curve for the **repeated-soft** methodology, generalization lesion. . 59
5.32 Learning curve for the **random-hard** methodology, FSAC lesion. 60
5.33 Learning curve for the **repeated-soft** methodology, FSAC lesion. 61
5.34 Learning curve for the non-linear domain point selection heuristics. 62
5.35 Learning curve for the discontinuous domain point selection heuristics. . . . 62
5.36 Learning curve for the linear domain interpolation lesion. 64
5.37 Learning curve for the non-linear domain interpolation lesion. 64
5.38 Learning curve for the discontinuous domain interpolation lesion. 65
5.39 Learning curve for the linear domain abstraction alternatives. 66
5.40 Learning curve for the non-linear domain abstraction alternatives. 66
5.41 Learning curve for the discontinuous domain abstraction alternatives. . . . 67
5.42 Learning curve for the linear domain feature selection alternatives. 68
5.43 Learning curve for the non-linear domain feature selection alternatives. . . . 68
5.44 Learning curve for the discontinuous domain feature selection alternatives. . 69
6.1 Relative error in **random-hard**, the linear domain. 75
6.2 Control adjustments in **random-soft**, the linear domain. 76
6.3 Relative error in **repeated-hard**, the linear domain. 76
6.4 Control adjustments in **repeated-soft**, the linear domain. 77
6.5 Relative error in **random-hard**, the non-linear domain. 78
6.6 Control adjustments in **random-soft**, the non-linear domain. 79
6.7 Relative Error in **repeated-hard**, the non-linear domain. 79
6.8 Control adjustments in **repeated-soft**, the non-linear domain. 80
6.9 Relative error in **random-hard**, the discontinuous domain. 81
6.10 Control adjustments in **random-soft**, the discontinuous domain. 82
6.11 Relative error in **repeated-hard**, the discontinuous domain. 82
6.12 Control adjustments in **repeated-soft**, the discontinuous domain. 83
6.13 Attempts to improve performance for **random-soft**, the linear domain. . . 85
7.1 Screen view of the SGI flight simulator. 87
7.2 Control adjustments for the simulated flight domain. 88

8.1 SPLICE behavior for multiple levels of control. 95
A.1 Final case database for the trace. 119

LIST OF APPENDICES

APPENDIX

A SPLICE Traces 104
B Data-chunking Numbers in Soar 121
C Soar Production Templates 124
D Glossary 127

CHAPTER 1

Introduction

Barney is a dinosaur from our imagination ... [1]

The field of artificial intelligence has made great advances since its inception, but until recently progress has been limited to largely static, nominal environments. Researchers found symbol processing systems effective in solving problems in these environments (e.g., [39]), and went so far as to formulate the Physical Symbol System Hypothesis (PSSH) [37], which states that all intelligent behavior can originate from a symbol system. Although aspects of static, nominal environments still provide challenges today, there has been a shift in focus toward investigating dynamic, continuous environments, such as driving [23], flying [52], robot navigation [25], or juggling [53]. In the process, many of the new paradigms developed for continuous environments, such as neural networks, reinforcement learning, and Bayesian nets, lost touch with their symbolic relatives. These approaches sacrificed generality and deep knowledge because of efficiency issues in dynamic environments and the difficulty of representing knowledge in continuous environments. This led to a natural division in the artificial intelligence community between research in established, symbolic fields, such as natural language, planning, and qualitative physics, and research in new problems like control and uncertainty. Not only does this render the PSSH inapplicable to these new problems[2], but it discourages researchers in different camps from taking advantage of each other's results.

The purpose of this thesis is to propose and implement SPLICE, Symbolic Performance & Learning In Continuous-valued Environments. Since some parts of the complete SPLICE theory are not fully implemented, the relevant sections will specify the restrictions of the implementation. Fully symbolic processing is intended to allow tighter integration between established symbolic capabilities and capabilities in continuous environments. Moreover, it brings the PSSH back into relevance for continuous environments. Besides merely performing in continuous environments, a secondary research goal is to show that symbolic adaptive control is comparable to traditional numerical approaches along a number of evaluation dimensions. The underlying symbolic architecture supporting this approach is Soar [29], the automatic subgoaling [27] and chunking [28] rule-based system. Many researchers have used Soar with success for symbolic capabilities such as natural language [34] and planning [49]. Soar provides practical strength with its state-of-the-art matching technology [12] and theoretical strength as a candidate Unified Theory of Cognition [38]. The following sections

[1] Thanks to Barney the Dinosaur for chapter quotes.

[2] The Poverty Conjecture [15] is a theoretical refutation of the PSSH in continuous environments, claiming that spatial reasoning requires some quantitative component.

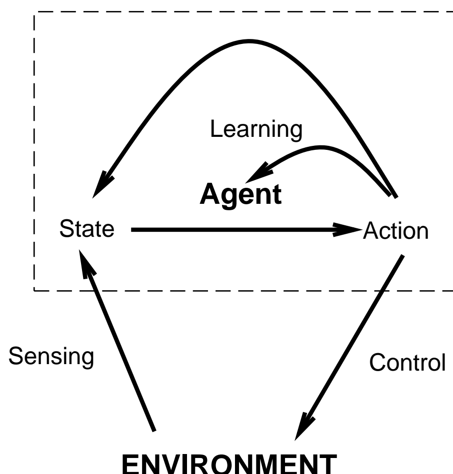

Figure 1.1: The interaction between agent and environment.

define more clearly the relationship between an agent and its environment, the nature of the environments covered in this thesis, and properties of the symbolic processing paradigm. The final section gives an overview of SPLICE in light of the design issues presented in this chapter, and summarizes SPLICE's results and contributions. This section also provides a road map to the remainder of this dissertation.

1.1 Agent-based Interaction

A common definition of artificial intelligence is "the study of rational action [51]." *Action* implies that there is some entity, called an *agent*, capable of affecting some external system, called an environment. *Rational* implies that this agent's actions are directed toward achieving its goals given its current knowledge. An agent interacts with its environment by receiving input with sensors and sending output with effectors, similar to how humans interact with the real world. Figure 1.1 depicts this interaction. The agent, enclosed in a dashed rectangle, senses information from the environment, uses its knowledge to map the environment state to an action, and then takes the action in the environment. The mapping may also change the agent's internal state, and it may change itself by modifying its knowledge.

Agents that learn from their experiences in dynamic environments have a significant constraint on the form of their learning. If the agent remembers every experience and must search them for every new situation, it will lose reactivity and be unable to respond to its changing environment after enough experiences. In fact, if the agent's response time is not $O(1)$ with respect to its number of experiences, the agent will eventually slow down to an arbitrarily large response time. A learning and retrieval system which is $O(1)$ is defined to be *incremental*. Since incremental architectures such as neural networks [50] and Soar do not slow down as they incorporate experiences, they are the only systems practical for agents with a long life expectancy in time-sensitive environments. In contrast, batch-style (non-$O(1)$) learning algorithms such as ID3 [46] perform best with either a upper bound on the number of experiences expected or unlimited computation time. Since SPLICE is intended for dynamic environments which require a fast response, the agent's learning component must be incremental.

A goal is a key component of the internal state of an agent. Most AI systems use some representation of a goal to guide behavior, with the notable exception of subsumption architectures [7], for example. Goals focus processing to achieve some desired situation. Explicit goal representations allow the agent to dynamically generate behavior intended to achieve the goals, giving the agent more flexibility than an agent with implicit goals in its behavior. The goal definitions vary widely, and there can be goals at many different levels. However, the highest-level goal of an agent is generally to complete some specific task, such as to survive or reproduce. Again, the definition of a task varies, but one usage is to change the observable state so that some or all sensed features match a particular pattern. The implementation adopts this definition of a task, with the further restriction that all goals may be achieved by a single action. This frees the agent from considering credit assignment, where a long chain of interactions fails to achieve a goal because an action somewhere in the middle was incorrect. Such a bottom-level agent can perform more complex tasks than a single-action task by receiving a sequence of goals explicitly describing the task. The source of this sequence can either be a human operator or a higher-level planning capability. For example, for juggling the sequence may be "throw with left hand, throw with right hand, catch with right hand, throw with left hand, catch with left hand, ...," and the low-level agent defines the specific arm motor commands for achieving this sequence.

Although any computer program with input and output can be considered an agent, there are several advantages to using agent terminology. First, it provides a clean separation between internal and external phenomena. Second, it utilizes the concept of a control cycle from the environment to the agent and back. Also, by carefully defining the interface, it is possible to measure the agent's autonomy in terms of the different types of input it needs and output it can generate. For example, an agent that constantly consults a human guidance "sensor" in addition to its environmental sensors is less autonomous than an agent that makes decisions based solely on its environmental sensors. Finally, the term "agent" does not specify the implementation technology, so in principle a computer or a human can be an agent for a particular environment.

1.2 Continuous-valued Environments

Continuous-valued environments, or CV environments, are a particularly challenging class of environments for AI agents. Although many environments have some continuous elements, this thesis is directed towards environments that fit a set of specific criteria commonly found together. The criteria attempt to be general enough to include many environments, but specific enough to allow a single basic performance and learning mechanism to apply to all of them.

1. Some or all environmental variables available to the agent are continuous-valued.

   According to standard practice in machine learning [14, 46], "continuous-valued" does not necessary imply real numbers. Continuous-valued variables can take on any element of a many-valued, totally-ordered set, such as the integers or natural numbers. Nominal variables take values from an unordered set, and there are many symbolic approaches for environments for these variables.

   The boundary between continuous and nominal variables is not always clear. For example, measurements such as height or width are commonly defined to be continuous, but features like color are commonly nominal. However, color is also a continuous

quantity, and the perception of hue as a continuous variable may allow an agent to generalize characteristics of red objects to maroon objects. Agents unable to handle continuous variables require continuous values to be discretized before reaching their sensors, like the quantization of the color wheel to specific colors. The choice between continuous and nominal representations is primarily a function of the ease of finding a suitable discretization of a variable and the ability of the agent to handle continuous values. For complex environments which contain some easily discretizable variables and some not, a hybrid approach that can handle both nominal and continuous values is necessary.

2. Some or all of the control variables, or effectors, available to the agent are continuous-valued.

   Continuous-valued effectors are very common in real-world environments, such as pressures on control pedals or chemical concentrations, but less common for AI research. Even research with continuous sensor readings frequently selects actions from a small set, such as bang-bang control in pole balancing [35, 16, 33]. AI systems with continuous actions include regression trees [6] and ALVINN [23].

3. The environment is dynamic.

   Although not all continuous-valued environments change over time independently of the agent, many realistic ones do, and these present particular problems to agents. One of the most significant problems is the need to consider time when computing a response. Since the environment is constantly changing, an agent attempting to accomplish a task in the environment must consider *when* the task should be achieved as well as *what* should be achieved. Another complication is that agents can only react to situations as they encounter them, so they cannot necessarily go back to a situation where they previously failed and try something else. Additionally, since the state variables are many-valued, it can be extremely difficult or impossible to achieve a specific value with any confidence, so the variable goal pattern should accept a number of values. These environmental properties lead to a more specific implementational definition of a task and a goal. A task is still a sequence of low-level goals, but the goal patterns are desired states, with each variable a range of acceptable values. Each goal also includes a deadline defining when the goal must be achieved. Figure 1.2 shows how variables change over time and the meaning of desired states and deadlines.

Many real-world CV environments include many state variables which interact over time nonlinearly, leading to extremely complex and difficult to control environments. The underlying patterns of change, taking the form of domain laws or equations in continuous environments, are frequently very difficult to acquire analytically. An additional complication is that these laws can change during performance or between analysis and performance, causing incorrect performance. A solution to this enormous complexity is to adapt performance based on experience in the environment. In this way, the agent learns from its own experience how to improve its performance. At the expense of initial poor performance, the agent developer saves the time and effort to explicitly model the environment, and the agent is robust to errors in the model and changes in the environment laws.

An important characteristic that is *not* assumed for this thesis is that the *domain laws* are continuous. There are many environments with continuous-valued variables that exhibit discontinuous behavior, such as automobiles when shifting gears and other physical systems

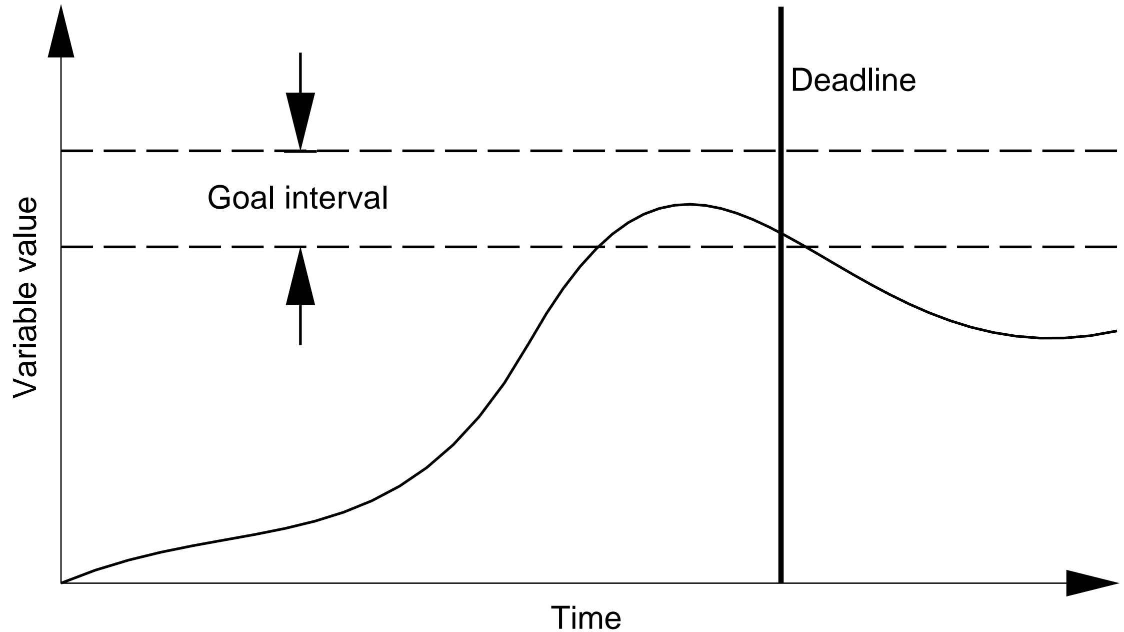

Figure 1.2: State variables, desired variables, and deadlines in continuous-valued environments.

when transitioning to different operating states. These domains are frequently difficult for numerical algorithms to model, but symbolic models naturally represent discontinuous phenomena. Another characteristic not explicitly exploited by this research is possible relationships among variables. For example, an image is an array of continuous-valued variables, but the important operations on images are pattern matching algorithms that search for significant patterns across the array. This work focuses on environments where variables measure different quantities and do not need to be combined to form patterns. Although numerical approaches such as neural networks have been applied to image processing [8], the fundamental independence of the input nodes implies that the network does not explicitly generalize patterns, but instead tends to memorize specific training examples.

1.3 Representation Styles

Most computational systems utilize some representation of the external environment. The system *encodes* its perceptions to the representation, and *decodes* the representation to action. The encoding and decoding functions depend on the nature of the environment and the nature of the representation. There are two major representational styles, numerical and symbolic. Numerical representations are quantities with arithmetical operations. Symbolic representations are combinatoric structures of atomic elements. Structures can be arbitrarily constructed and manipulated. This section discusses the numerical and symbolic representation styles, and motivates the use of symbolic representations for continuous-valued environments from a functional viewpoint.

Numerical representations naturally represent continuous environmental variables, but they can also be made to represent nominal values. The necessary encoding function is assigning some ordering to the set of possible nominal values, and the decoding function is a

rounding or thresholding operation back to the discrete ordering. Numerical representation of nominal values has some advantages over symbolic representation, because continuous quantities allow infinite shades of meaning and the potential to combine elements to create new atomic elements, instead of just additional structure. Numerical methods such as Neural Networks [50] commonly process nominal data.

Symbolic representations naturally represent nominal environmental variables, but they can also represent continuous variables. The necessary encoding function is to convert quantities to symbols using inequalities or some other qualitative measure. The decoding function re-quantizes symbols by reversing the encoding function. Like numerical representations of nominal values, symbolic representation of continuous values provides some potential functional advantage. The primary benefits are improved flexibility and generality, while potential drawbacks include computation speed and memory requirements.

Although the direct analogy between continuous environmental features and numeric representational elements allows rapid computation and efficient storage, it also reduces flexibility. The disadvantage to representing continuous values as numbers is that there is no capability for abstraction and high-level reasoning. In contrast, symbols can represent continuous values at different levels of abstraction, and symbols can combine in problem spaces to search for high-level solutions. For example, an airplane autopilot may perform well with all controls functional, but if the flaps tear off or an engine breaks down, the autopilot will not correctly control the plane unless these possibilities are implicitly part of its control program. A symbolic reasoner, however, may be able to reason about the new situation at a high level and take action to correct its behavior, even if the developers never planned for that particular situation.

Symbolic processing at varying levels of abstraction also potentially provides greater transparency to user monitoring. While numerical processing is limited to computing numbers with no opportunity for explanation, a symbolic system may be able to give a complete justification of its actions at multiple levels of detail.

In terms of computational speed, although numerical systems utilize mostly pre-compiled knowledge and process very quickly, their rate of computation is generally fixed without respect for the priority of the situation. In contrast, although a symbolic system has more overhead for symbol manipulation, it can be capable of reasoning about the time-criticality of the situation and make approximate, fast responses when necessary.

Space requirements for numerical systems vary widely depending on the characteristics of the environment. If the underlying representation used by numerical systems matches the underlying behavior of the domain well, the representation can be quite sparse. But if the match is not as good, space requirements can grow immensely and performance will suffer as well. For example, if a numerical system attempts to fit a continuous model to a highly discontinuous environment, it will require many high-order terms to fit the discontinuities, and it will never be perfectly accurate. Symbolic systems can avoid this problem by not requiring that the domain fit a specific numeric model.

Finally, a possibly key advantage for some hybrid environments is that a symbolic system naturally handles nominal variables, while numerical systems must encode and decode discrete unordered sets to an arbitrary order for processing. A mix of continuous variables and nominal variables is extremely common in complex environments, such as the current gear for an automobile or the type of object for a juggler.

Although not all require all of the capabilities of symbolic processing, and not all symbolic systems posses all of those capabilities, these observations show that a symbolic approach to control is a valid alternative to traditional analytic computation from a purely

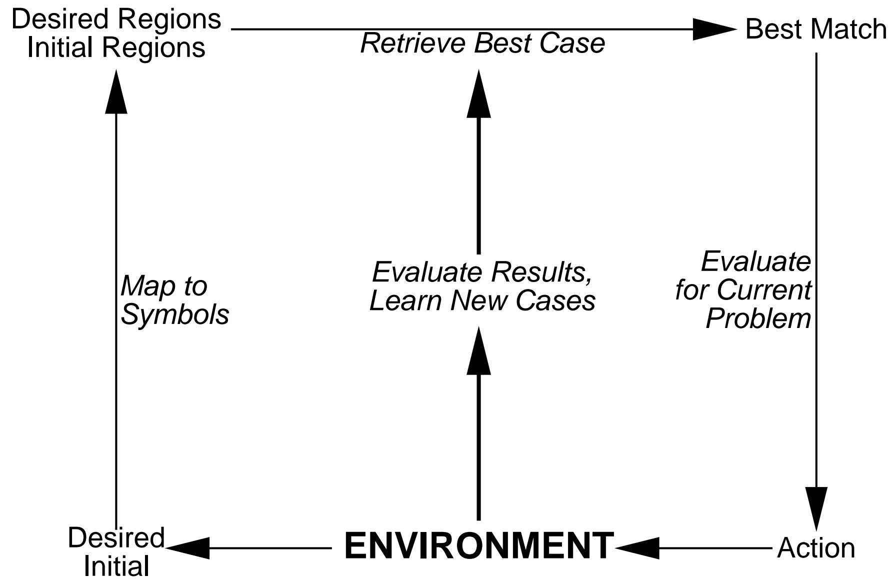

Figure 1.3: Overview of SPLICE.

performance point of view. As mentioned above, additional theoretical support comes from the potential for a tight integration with other symbolic capabilities, and an additional testing domain for the Physical Symbol System Hypothesis.

1.4 Overview of SPLICE

The objective of this thesis is to create a symbolic agent capable of performing in a continuous-valued environment, and learning to improve its performance from its own experience. Chapter 2 presents related work in continuous-valued environments.

Chapter 3 introduces the performance component. The SPLICE agent maintains an explicit internal goal consisting of a numeric desired state and a deadline for the achievement of that state, relative to the instant SPLICE sends a control adjustment to the environment. After the agent senses the environment, it classifies the current state and the desired state into a set of symbolic regions. Each variable receives a hierarchy of regions of varying size or generality. The agent uses these regions to search for the most specific relevant case stored in a database of generalized past experiences. The cases contain local linear models based on experienced points. If there is no relevant case, the agent uses a domain-specific qualitative model and a qualitative reasoner to synthesize an initial qualitative case. When the agent selects a case, it uses the case's linear model to interpolate a control given the current and desired state. The agent sends the interpolated control adjustment to the environment. The outer edges of Figure 1.3 define the performance arc.

Chapter 4 adds an incremental learning component to SPLICE. The learning component evaluates the sensed state at the deadline and compares it with the desired state. Based on this difference, SPLICE creates two new cases for future use. One case corrects the linear model local to the actual result, and the other case improves the linear model local to the desired result. Given some environmental assumptions, the local model for the desired result will provably converge to be accurate enough so that the effect of the interpolated control is arbitrarily close to the desired result. The interior of Figure 1.3 depicts the learning

component.

Chapter 5 covers several evaluation methods and test domains, SPLICE's execution performance, and the contribution of individual SPLICE mechanisms through lesion studies. This analysis concludes that the learning rate and final accuracy depends greatly on the experimental methodology used, as well as the characteristics of the test domain. The lesion studies show that SPLICE's key capabilities are generalization and interpolation.

Chapter 6 compares SPLICE against popular numerical control algorithms. SPLICE is found to perform comparably with the algorithms. SPLICE's knowledge representation is more flexible than some algorithms, allowing it to learn in a wider range of environments. SPLICE's learning rate is never notably slower than other algorithms, and is better than all other algorithms for some experiments.

The final evaluation chapter introduces a new realistic and complex domain, simulated flight, and evaluates SPLICE performance in this domain. Chapter 7 also includes a general discussion on applying general learning theories to complex, real-world domains. SPLICE is capable of learning to achieve simple goals in the flight simulator, but its action model proved insufficient for complex goals requiring several control adjustments over time.

Chapter 8 discusses properties of SPLICE and investigates future enhancements. Finally, Chapter 9 derives some overall conclusions about the research and presents the thesis contributions.

The appendices contain additional information. Appendix A contains Soar traces of a single SPLICE problem-solving cycle for more specific insight on its mechanisms. Appendices B and C contain specific Soar techniques necessary for implementation. A glossary of terms is in Appendix D.

Although this thesis makes advances in extending Soar's set of capabilities and combining qualitative and quantitative reasoning, the major unique contribution is the development of an entirely symbolic (except for the necessary translation from continuous sensors and to continuous actions) learning and acting agent for continuous-valued environments.

CHAPTER 2

Background

Everyone is special! Everyone in his or her own way!

Researchers have studied the problem of action in continuous-valued environments from a variety of viewpoints. Mechanical engineers need to control their devices; AI researchers study the relationship between sensing, thought, and action; human interface designers attempt to simplify operation of complex systems; and psychologists investigate how the mind processes information and interacts with the world. The most direct antecedents to this work are algorithms from control theory and AI designed to operate in continuous dynamic environments. The following sections introduce and summarize work in numerical static control and adaptive control, and symbolic variants of these.

2.1 Control: Performance in a Continuous-valued Environment

The most straightforward approach to control in the domains of interest in this thesis is the concern of traditional control theory [13]. In control theory, researchers first model the environment laws using equations, then study the model for stability and other properties. When the model is well understood, researchers derive control policies that guide the system to desired or stable states. An additional step is proving that the given control policy will achieve the desired situation, assuming that the environmental model is correct.

A simple form of control models the environment laws as a function f. This function maps a state x and a control u to a subsequent state $y = f(x, u)$. If the system needs to attain a desired state y^*, the control policy or response surface g is just the inverse of f, $g(x, y^*) = f^{-1}(x, y^*) = u$. The difficulty with this solution is that the inverse does not always exist for all values of (x, y^*), or it may be difficult to compute.

An impressive example of a traditional control system is the Büeggler [47]. This is a robotic actuated paddle combined with video cameras for sensing. It can simultaneously bounce two table tennis balls in the air at a specified height. The control laws are robust to small disturbances and provably correct.

Chapter 6 compares a simple algorithm from traditional control theory, PID control, to the algorithm developed in this research and other control algorithms.

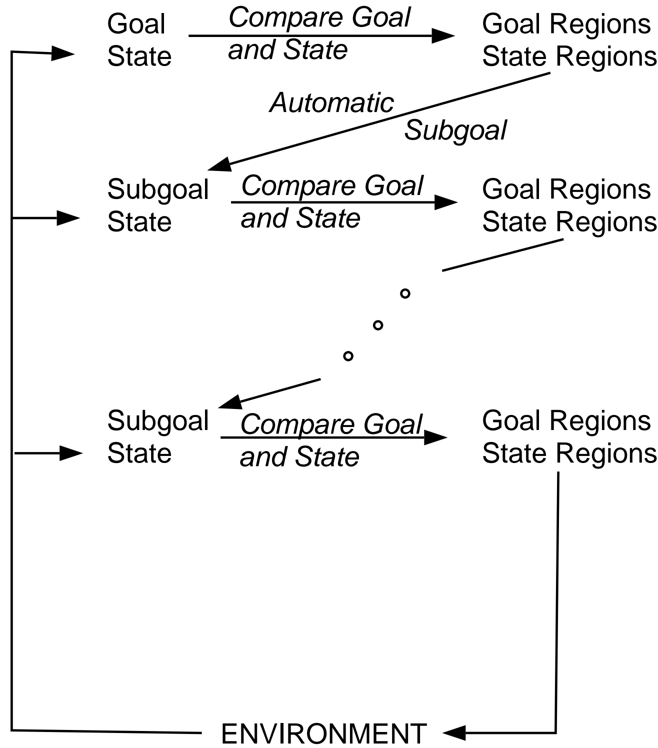

Figure 2.1: The Air-Soar control hierarchy.

2.2 Symbolic Control

The Air-Soar system [43] is an example of a symbolic system for control. Air-Soar acts as a pilot in a simulated airplane. The environment fits the definition of continuous because its sensed variables and effectors are continuous-valued, and the domain is dynamic. Air-Soar is symbolic because it is implemented in Soar, like SPLICE in this thesis. Instead of explicitly modeling the environment and inverting it, Air-Soar uses a great deal of domain knowledge to decompose the flying problem. The first step in processing sensory information is to convert continuous values to symbols. Air-Soar compares input to desired values, and categorizes the difference into rough symbolic descriptions, such as *too-high; large* or *achieved*. If not achieved, the agent automatically generates a subgoal. Once achieved, the subgoals were designed to guarantee to achieve the higher-level goal after some time delay. If a subgoal is not currently achieved, Air-Soar recursively selects a subsubgoal for this subgoal, resulting in a hierarchical series of subgoals, bottoming out in specific effector adjustments for the plane. In terms of the dynamic nature of the flight environment, Air-Soar implicitly attempts to achieve its goals as soon as possible. A diagram of the subgoal mechanism is in Figure 2.1.

For example, for a plane to achieve a particular altitude, domain knowledge decomposes this goal to achieving a particular climb rate. If the plane is below the desired height, the climb rate is positive, and negative if above. The magnitude of the climb rate is proportional to the magnitude of the altitude difference. Next, the agent decomposes this subgoal to climb acceleration, using the same differential reasoning as before. Finally, the agent derives the desired pitch of the airplane. The agent effectors are able to directly achieve this pitch by a high-level simulator command to keep adjusting the elevator control surfaces until the

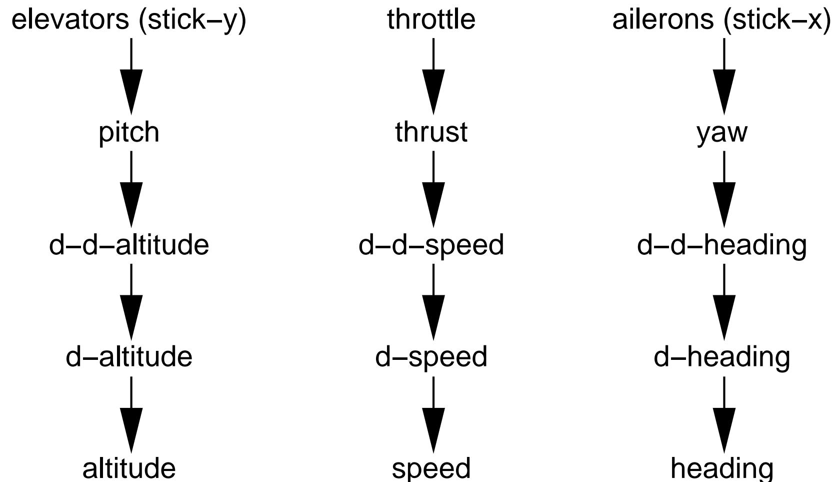

Figure 2.2: The three variable chains in Air-Soar.

plane has the desired pitch. This sequence of subgoals is a chain of cause and effect, starting at the level directly accessible to Air-Soar and ending at the highest level goals needed in Air-Soar. There are a total of three of these variable chains, as shown in Figure 2.2.

Although Air-Soar successfully flies the plane, it is not robust to changes or errors in its assumptions. It assumes that its domain knowledge is correct and complete for a number of different knowledge types. The first knowledge type is the mapping between values and symbolic qualitative descriptions. The mapping is very important for correct execution, but it cannot be altered if it turns out to be incorrect. The second type is the goal decomposition hierarchy. Although the hierarchy itself is probably accurate, the specific subgoals generated are predefined, but may not be appropriate for the environment. The final knowledge type is what subgoal level is directly achievable by a single action in the environment. Not only is this detailed and complex knowledge difficult to develop correctly, it may fail if the environment changes in unanticipated ways. For example, this system was adequate for a simulated Cessna airplane, but it was unable to control a fighter jet. The ability to adapt to changes in environment, and to adapt to an environment without specific control knowledge, requires a system capable of learning to change its own performance from experience with the environment.

2.3 Adaptive Control: Adjusting to the Environment

For environments too complex to model, or for environments where a model may become invalid due to changing domain laws, traditional control techniques fail. For these difficult problems, one solution is to gather data from the environment and adapt the controller to the observations. There are a number of approaches to implement this, but the most straightforward method returns to viewing the environment laws as a function $f(x, u) = y$. Instead of using a predefined function, *function approximation* methods either approximate f from data points (x, u, y) and invert it for the control law, or they directly approximate $g = f^{-1}$. Statistics provides a number of models for function approximation, such as linear and polynomial functions and spline curves [9]. Generally, the approximation algorithm attempts to specify parameters for the model in order to minimize the error between each observed data point and the model m at that point. For example, a linear model in one

dimension is $m_{linear}(x) = \alpha \cdot x + \beta$. A popular approach to finding these parameters is called the LMS (least mean squares) error [58]. It is based on minimizing the RMS (root mean square) error measurement for the set of n observed points (X, U, Y),

$$RMS(X,U,Y) = \sqrt{\sum_i \frac{(y_i - m(x_i,u_i))^2}{n}}.$$

Function approximations are mathematically proven to find the best parameters for a given model to fit the data, but they may not perform well if the model is not appropriate, or the data is incorrect.

One solution to the problem of finding appropriate models is designing extremely flexible models with many parameters. One example of a model with many parameters is a spline fit, which will be discussed in more detail and experimentally compared in Chapter 6. Another popular example is neural networks ([51], Chapter 19). Based on some aspects of human neurons, these networks combine input using a fixed number of weights to compute an output value. Since the network computes a complex function which is not in general invertible, the network directly computes the inverse model, $m_{neural}(x, y^*) = u$. The network uses two input nodes for x and y^* and one output node for u. In training mode, the neural net compares the computed output to the actual control, and adjusts the weights to reduce the error. After many iterations of the training data, the neural net is capable of approximating the correct control on new problems. Although artificial neural networks are popular for a wide range of machine learning tasks, their continuous nature makes them particularly appropriate for control in a continuous environment. One successful example of using neural networks to control a vehicle is ALVINN [23]. However, this work and others show that it still takes considerable effort to successfully train neural networks in very complex environments.

The above function approximation algorithms are considered parametric methods because a fixed number of parameters for the model are adapted to fit the data. Since the model itself is fixed, the possible domains for a particular model is limited. A second class of function approximation algorithms are non-parametric algorithms [54], where the data defining the model are not fixed in advanced. These algorithms are considered more flexible because they can have a higher degree of self-organization. Hart described an early example of a non-parametric method with the Nearest-Neighbor algorithm [18]. This algorithm simply recorded data as experienced. When the algorithm was required to issue a control, it found the k nearest points to the query (x, y^*) and performed a weighted average to find the correct control. One advantage of this algorithm is that it is guaranteed to respond with the correct control if it has experienced it in the past. A disadvantage is that the algorithm is expensive in space because it has to store every point, and expensive in time for control because it has to search every point for the nearest neighbors.

Kibler and Aha independently derived a similar algorithm called Instance-Based Learning [1] for symbolic classification and introduce some heuristics to improve efficiency. Their most efficient version only adds instances to memory if the classification is incorrect, so the algorithm only learns when it makes a mistake. Schaal and Atkeson extend this concept to continuous domains with Receptive Field Weighted Regression (RFWR) [55]. They collect data in receptive fields, and each receptive field contains a linear model of its data. RFWR only modifies a receptive field if the field's model is inconsistent with a data point. This algorithm is covered in more detail in Chapter 6. Schaal and Atkeson use yet another non-parametric method to estimate the value function for reinforcement learning [24], where

they train a robot to perform devil-sticking [53].

2.4 Symbolic Adaptive Control

Finally, there are symbolic and hybrid systems which perform adaptive control. The most direct analogues to numerical adaptive control are regression trees [6], which are standard decision trees [46] with linear models in the leaves, and variable inequality tests in the internal nodes. For control, the decision tree classifies the current and desired states (x, y^*) to find the correct linear model, and then computes the control from the model. Sammut et al. describe a specific application which uses decision trees for control [52]. This system collects a large amount of human data from flying a plane in a flight simulator, and uses the data to generate a decision tree. Including the decision tree as the control law for the plane actually allows the plane to fly slightly better than the human pilots, because the tree induction includes mechanisms for simplifying the tree by removing errors. However, the autopilot is unable to further learn from its own experience, and it is unable to deviate from the course of the pilots. The data set was also extensively processed to manually simplify the training task. The researchers are working on techniques for generalizing the autopilot to other tasks. A particularly interesting aspect of this work is that the flight simulation domain is the same as the one used in the experiments in complex environments in Chapter 7 of this thesis.

DeJong develops a hybrid system for adaptive control in complex environments [10]. The system first attempts to predict what variables will be useful in solving a problem using a symbolic qualitative physics model of the environment. Since the qualitative model is ambiguous and can generate many possible solutions, the system by default chooses the simplest explanation of the qualitative relationships. The system uses these relationships to send commands to the environment, and records the responses. It builds a piecewise linear model of the environment until its qualitative model is violated. It then returns to its next-simplest qualitative explanation and continues with the linear model. It continues until it detects no more violations of its current explanation. This system was implemented in an automotive domain. It is the most similar to the current research in terms of approach, but the only symbolic processing is in the qualitative reasoner, which acts as a front end to the numerical piecewise linear model. Additionally, although not documented, recording all experienced points for a piecewise linear model can lead to inefficient storage and execution costs. The system takes advantage of the symbolic capability of high-level reasoning about the tasks and environment. This allows it to greatly reduce dimensionality of learning and perform at a basic level of competence quickly. However, using numerical processing for performance and learning after computing the qualitative model prevents the system from fully taking advantage of the symbolic capabilities discussed in Chapter 1.

A related problem to continuous control is scientific discovery of natural laws [31]. These systems gather data from experiments and attempt discover regularities. BACON [30] is a well-known example. BACON replicates the discoveries of early chemists by systematically varying conditions in experiments. There are two major differences between scientific discovery and adaptive control. The first is that the objective in discovery is just to derive laws describing a domain, whereas controllers attempt to use these laws to allow an agent to accomplish a task. The second is that scientific discovery normally implicitly assumes that the relationship between variables will be simple, so it biases its search toward simple laws first. Adaptive control uses flexible representations for domain laws because it expects

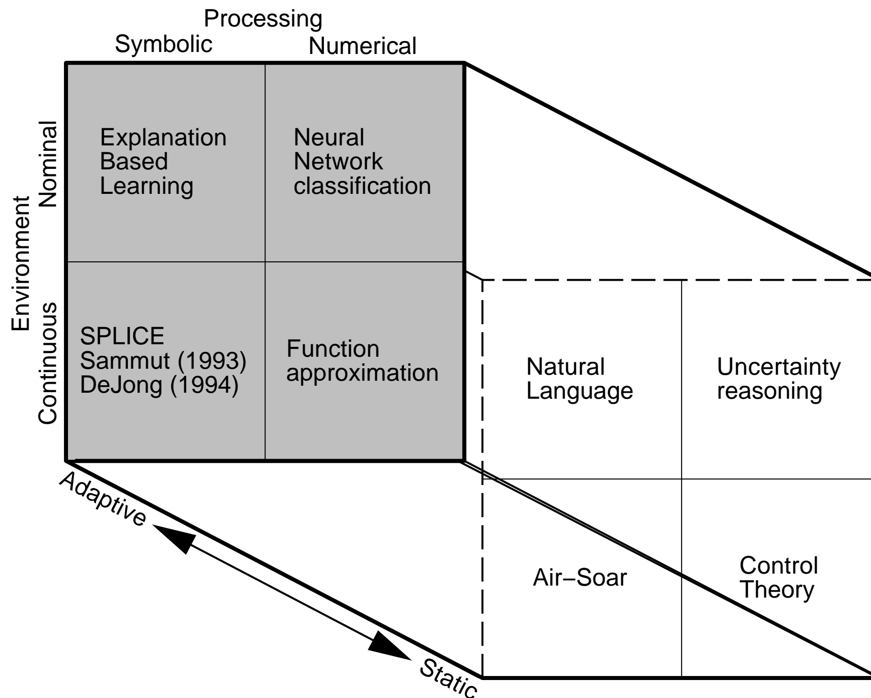

Figure 2.3: Characterizing systems along three dimensions.

complex relationships.

2.5 Summary

There are many dimensions for characterizing systems that interact with an environment. The two explored here are style of processing (symbolic vs. numerical) and adaptivity. For completeness, an additional dimension is the types of features in the environment (nominal vs. continuous-valued). The two possibilities for each of these three dimensions result in eight different general approaches. Figure 2.3 presents some examples for each approach and their relationships.

Since continuous domains are common in the real world, there has been much interest in creating systems that perform in these environments. There have been many approaches, but none are capable of adapting themselves to very complex environments and integrating tightly with symbolic capabilities. This thesis work fills this gap by creating an entirely symbolic system capable of learning and performance in continuous-valued environments.

CHAPTER 3

Performance

Hurry, hurry, drive the fire truck ...

The first requirement of an agent existing in an environment is to interact with the environment. This is the task of SPLICE's Performance Element (PE). The PE computes and executes an action based on the initial state which attempts to satisfy the current goals. For dynamic environments, a goal consists of a list of desired values for some variables in the state and a deadline for the achievement of that state, as described in Chapter 1. So the task of the PE is to adjust a control so that the state of the environment at the deadline is sufficiently close to the desired state. This state at the deadline becomes the initial state for the next iteration of the PE. Goals can remain constant (a setpoint) or vary over time (a goal trajectory) [13].

Some definitions are necessary to completely describe the Performance Element.

Definition 1 *The basic unit of information is the* State*, a set of values for variables,* $\mathcal{S} = (x_{variable_1}, \ldots, x_{variable_N})$*, for* $i = 1 \ldots N$*.*

There are several kinds and combinations of a state.

Definition 2 *The* Initial State I *is the state of the environment at the beginning of the PE cycle. The* Final State F *is the state at the deadline. The* Desired State D *defines the goal.*

The Desired State need not be fully specified, so the implicit goal of the PE is to get as close as possible to the specified values and ignore the unspecified values. Variables in D are *Desired Variables.*

Definition 3 *The pair* (I, D) *is a* Problem P*, which is solved by a control* U*. The action is a set of control adjustments,* $U \in \mathcal{S}$*.*

Variables allowed as actions are *Control Variables*, which the PE can set to any value in the allowable range in one cycle. Note that this definition may be nonintuitive, depending on the input interface to the agent. For example, an agent controlling a radio may have a tuning knob and be able to tune to any station in one cycle, or the agent may have "up" and "down" pushbuttons and only be able to change one frequency unit in one cycle. The tuning knob position is many-valued and ordered, so it is a valid control for SPLICE, but the pushbuttons are discrete, so plain SPLICE would not accept this as a control.

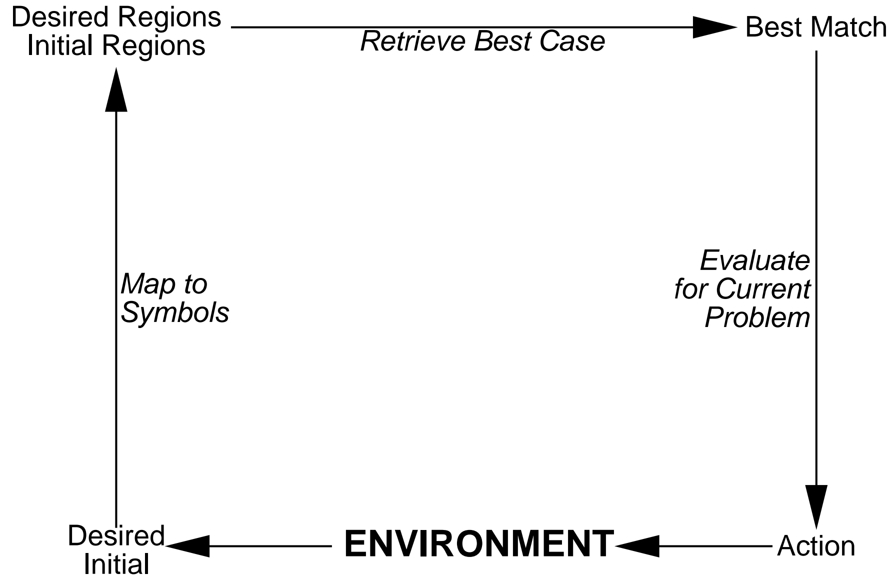

Figure 3.1: Performance arc of SPLICE.

For example, suppose a SPLICE agent is operating in a simplified driving domain. The state consists of the car speed and the car pedal position. Let the Initial State I be $(x_{speed}, x_{pedal}) = (0, 0)$. The Desired State D is $y_{speed} = 65$, making *speed* a Desired Variable. Controlling multiple variables, such as $(y_{speed}, dy_{speed}) = (65, 5)$, is beyond the reach of the current SPLICE implementation. Chapter 8 covers extending SPLICE to allow this. So the Problem P is $(I, D) = (0, 65)$, to achieve the desired speed of 65 from the initial speed of 0. The Control U is $y_{pedal} = 10$, making *pedal* a Control Variable. After the deadline, the Final State F is $(50, 10)$. This means that the PE sensed that the car was initially stationary, computed a pedal adjustment of 10 attempting to achieve a speed of 65 by the deadline, and actually attained a speed of 50, 15 less than its goal.

Figure 3.1 depicts the components of the Performance Element for SPLICE. As described in the following sections, the agent first maps the continuous state and goals to symbols, then uses the symbols to select a case out of the database, then computes a continuous-valued control from the selected case. The control is finally executed in the environment.

3.1 Map to Symbols

The first step in generating a continuous-valued action for SPLICE is to convert the current state and goals to symbolic form. A straightforward approach would be to assign a set of "threshold values" to each variable, and classify current state variable values and current desired values according to the intervals in which they fall (e.g., [14]). The problem with this approach is that the threshold values themselves are very hard to extract from the environment through interaction. The most successful results [14] come from batch analysis of data, which is not tractable for incremental, integrated learning and performance systems. An additional problem is a lack of structure in the threshold set. An incremental algorithm must expect to only provide general answers at first, and gradually become more specific and accurate. The threshold set needs to be augmented by a structure defining different levels of generality.

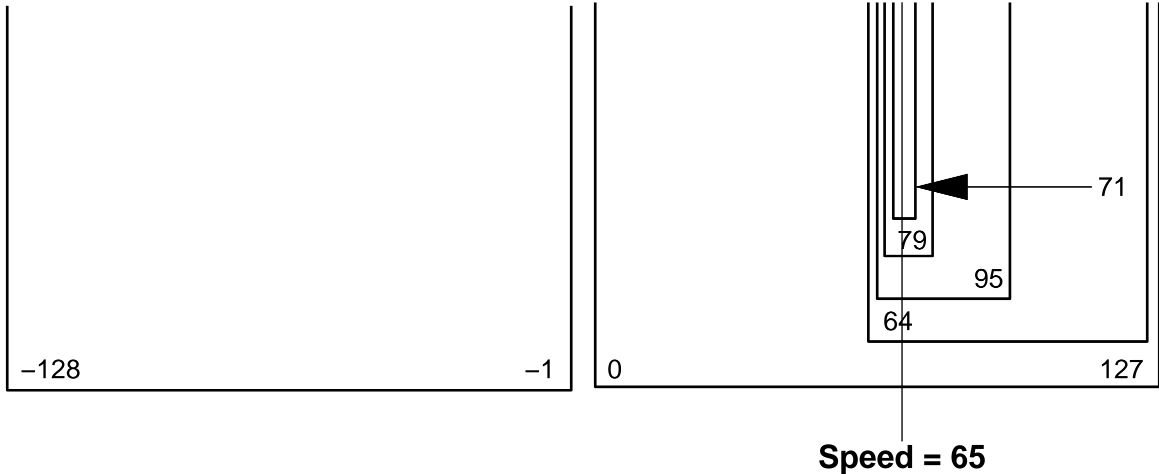

Figure 3.2: Region subdivision example.

The symbolic conversion data structure SPLICE uses is tailored to its incremental, successive-approximation nature. The space of possible values for a variable is partitioned into a hierarchy of ranges or regions. The top of the hierarchy is the most general partition, positive and negative. Each variable i has an allowable range of $(min_i, max_i) = (2^m, 2^n)$. The range must be a power of 2 because following levels are binary subdivisions of their parent levels. The second level divides positive and negative into four regions. The recursive binary subdivision continues to a user-defined lower limit.

Definition 4 *Let x_i be the value of state variable i. The recursive binary subdivision $BinSub(Min, Max) = \{(Min, \lfloor Min + Max/2 \rfloor), (\lceil Min + Max/2 \rceil, Max)\}$. The regions to which x_i belongs are denoted $x_{i,j}$, where $x_{i,1} \in \{positive, negative\}$, and $x_{i,j} \in BinSub(x_{i,j-1})$, up to some maximal j called lsr_i, the least significant range of i.*

This implies that the width of the smallest bucket is

$$\frac{max_i - min_i + 1}{2^{lsr_i}}$$

units.

To continue the driving example, let $min_{speed} = -128$, $max_{speed} = 127$, and $lsr_{speed} = 5$. The width of the smallest bucket is $256/32 = 8$. Since the desired speed (65) is positive, the most general region is 0–127. Since 65 is greater than 64, the next region is 64–127. Since 65 is less than 96, the next region is 64–95. Since 65 is less than 80, the next region is 64–79. Finally, since 65 is less than 72, the most specific region is 64–71. Figure 3.2 illustrates the region assignment process.

3.2 Retrieve Best Case

SPLICE stores the cases in the database as generalized problems.

Definition 5 *The generalization Gen of a state S is a vector of generalizations $g = (g_1, \ldots, g_N)$, $g_i \in \mathbb{N}$, $1 \leq g_i \leq lsr_i$ on individual variables, so that $Gen_S(g)_i = \{x_{i,j} | 1 \leq j \leq g_i\}$. $Gen_{S'}(g)$* matches *S if and only if $Gen_{S'}(g)_i \subseteq Gen_S((lsr_1, \ldots, lsr_N))_i, 1 \leq i \leq N$.*

```
function RETRIEVE-BEST-CASE(C, I, D) returns a case
    inputs: C, the case database
            I, the initial state
            D, the desired state
    output: M, the most specific match

    for g_initial ← Σ lsr_i downto 0
        M_I ← {Gen_{I_j,D_j}(i_j, d_j) ∈ C | Matches(Gen_{I_j}(i_j), I),
                                            Σ i_{j_k} = g_initial}
        if M_I ≠ ∅ break

    for g_desired ← Σ lsr_i downto 0
        M_D ← {Gen_{I_j,D_j}(i_j, d_j) ∈ M_I | Matches(Gen_{D_j}(d_j), D),
                                              Σ i_{j_k} = g_desired}
        if M_D ≠ ∅ break

    return the element of M_D most recently added to the case database
```

Table 3.1: The RETRIEVE-BEST-CASE algorithm

Another way to write this is that a generalization vector of S', $(g_1, \ldots, g_N)$, matches S if and only if $\forall (Min, Max)_{i,g_i}, Min \leq x_i \leq Max$. Note that many generalizations match S by this definition. The best match to a state is the most specific generalization. This is because, in general, the most specific generalization is the most similar to the state and therefore is most likely to be relevant. A generalization g is more specific than g' if and only if it includes more regions.

Definition 6 $Specific(g) = \sum g_i$.

For example, the most specific possible generalization is $g_i = lsr_i$. Reducing any g_i corresponds to removing a maximally specific region from $Gen_S(g)$, so the generalization becomes more general. Now we are ready to precisely define the case database.

Definition 7 *The* case database C *is a set of previously experienced generalized problems* (I_j, D_j), $\{(Gen_{I_j}(i_j), Gen_{D_j}(d_j))\}$ *where* i_j *is the generalization of initial state* I_j *and* d_j *is the generalization of desired state* D_j. *For notational compactness,* $(Gen_I(i), Gen_D(d)) = Gen_{I,D}(i, d)$.

3.2.1 Match to Case Database

SPLICE uses these definitions to select the best match M from the set of stored cases $C = \{Gen_{I_j,D_j}(i_j, d_j)\}$ for the current problem (I, D). The English-level description of the matching process is that SPLICE matches the case with the most specific matching goal of the cases with the most specific matching initial states. Control variables in the initial state can be ignored because they can be set to any value. The precise algorithm for computing the best match is in Table 3.1. The algorithm chooses the most recently added case to compensate for non-stationary environments.

Case	Most Specific		P_1	P_2
	Initial	Desired		
$A = Gen_{5,40}(3,1)$	0–31	0–127	(0,40,8)	(0,30,6)
$B = Gen_{5,65}(3,4)$	0–31	64–79	(5,30,5)	(5,25,4)
$C = Gen_{20,65}(1,5)$	0–127	64–71	(0,40,8)	(20,90,14)

Table 3.2: Sample case database for the driving example.

In the driving example, the problem P is $((x_{speed}, x_{pedal}), y_{speed}) = ((0,0), 65)$. Suppose the case database contained three cases as illustrated in Table 3.2. All three cases match the initial state, but A and B are more specific than C, so $M_I = \{A, B\}$. However, since the desired generalization for B is more specific than A, the best match $M = B$.

SPLICE uses the SCA inductive learning technique [36] to implement case retrieval. SCA matches cases in parallel by first checking if any maximally specific initial state matches the current initial state. Next SCA generalizes the current initial state one step and again checks for matches. SCA continues until it finds a matching initial state or concludes there are no matching initial states. SCA follows the same procedure for desired states. If multiple cases are best, SCA utilizes annotations on the cases, where each case points to its previous case. SCA selects the case to which no case points. This linked structure is possible because SCA learns by creating new cases out of old cases, as described in Chapter 4.

3.2.2 The Domain Model

There may be no best match for a problem P. This could occur either because SPLICE has not experienced anything in this environment and has an empty case database, or because SPLICE has never received a goal for one of the current Desired Variables before. For this problem, SPLICE must resort to internal approximate knowledge of the domain. This knowledge takes the form of a qualitative physics model. The task of the qualitative physics model is to determine what controls affect the Desired Variables, and what the qualitative relationship is between these controls and the Desired Variables. To accomplish this, SPLICE sequentially adjusts all the controls it can access, and qualitatively simulates the results using a model such as that in Figure 3.3. This model of a heat exchanger, based on an example [59] in the work of Williams et al., describes the effect of input temperatures ($T_{1,3}$) and flow rates ($f_{1,2}$) on output temperatures ($T_{2,4}$). Normally, the objective of the heat exchanger is to cool down a hot fluid such as steam by running a cold fluid such as ice-water next to it to draw off the heat. Although there are four input variables, the operator normally only has direct control over the flow rate of the cold leg, f_2. The other variables depend on environmental factors. The final temperature of the hot leg, T_2, is usually the variable to be controlled. As the cold leg flow increases, the exchanger draws more heat from the hot leg, lowering T_2.

If a Desired Variable changes in the same direction as the control adjustment, the relationship is positive. If a Desired Variable changes in the opposite direction of the control adjustment, the relationship is negative. In the heat exchanger example, the qualitative reasoner deduces a negative relationship between f_2 and T_2 by raising f_2 and noting that T_2 falls. If no control movement affects all Desired Variables, the qualitative reasoner begins combining controls together and continues simulating. If two controls have contradictory effects, the qualitative simulator labels the relationship "ambiguous." When it can predict that some combination of controls will have some relationship to all the Desired Variables,

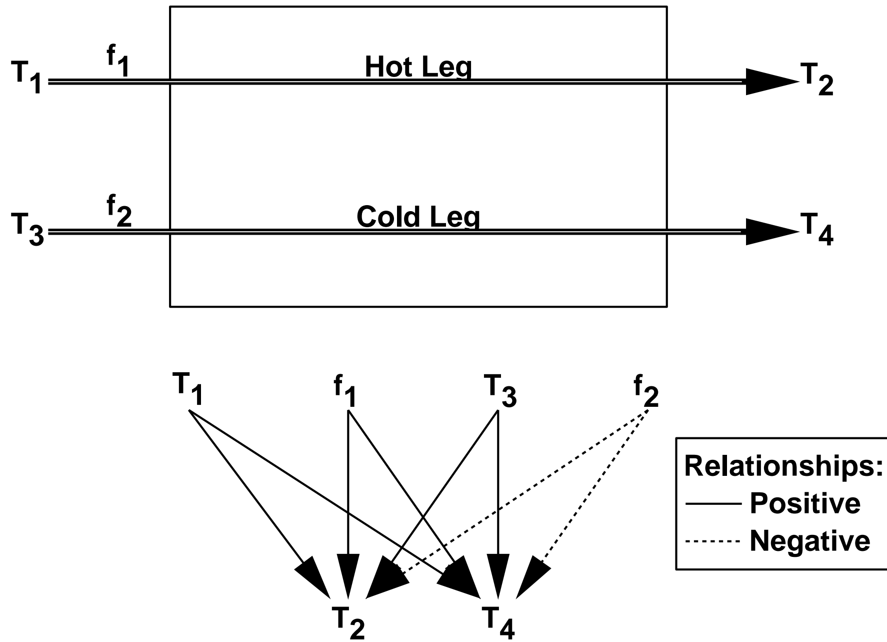

Figure 3.3: Qualitative model of a heat exchanger.

the reasoner stores this information for use now and in the future. Although this forward chaining process may seem overly complex for this model, the qualitative model may take the form of any directed acyclic graph, so it is necessary to find a path from the controls to the desired variables.

As an implementational restriction, although the qualitative model is fully general for any number of simultaneous controls and desired variables, later stages of processing assume that there is only one control to be adjusted, u, and one desired variable, d. This is primarily because of the credit assignment problem in learning. From this point, the presentation will assume that there is a single Desired Variable, which is affected by a single control adjustment. The Discussion (Chapter 8) covers issues concerning multiple controls and multiple Desired Variables.

3.3 Evaluate for Current Problem

Once the PE finds the best match M, it uses that match to solve the current problem P. Rather than directly using the action taken to solve M, SPLICE attempts to generalize the information retrieved from M to P. This information is in the form of points of the form $P = (I, F, U)$ associated with M. These points were either acquired during learning, as described in Chapter 4, or computed from the domain model. The domain model generates two points by examining the relationship between controls and Desired Variables. If the relationship between a control C and a desired D is positive, the points are (Min_D, Max_D, Max_C) and (Max_D, Min_D, Min_C). In other words, a maximum control adjustment will take the minimum Desired Variable value to the maximum value by the deadline, and the inverse. The domain model switches the control adjustments for a negative relationship. Although these points are almost certainly incorrect in the environment, they are by definition not overestimates of the correct control, because, for example, it is impossible for the agent to try more than the maximum control to change an minimal initial value to the maximum

for that variable. Also, the learning mechanism will ensure that the PE only uses domain points for the initial stages of learning.

Given these points, there is one case where SPLICE does not need to generalize and can directly use the match. This is where one of the recalled points exactly matches the initial conditions and desired conditions. In this case, SPLICE immediately takes the recalled control.

If none of the points exactly matches the problem, SPLICE linearly interpolates over the points and the current problem to find a control adjustment. The interpolation approximates a response function $f : I \times D \mapsto U$ relating the current and desired states to control adjustments. SPLICE assumes that f is approximately linear over sufficiently small "patches," and the matched case represents one such patch. A linear interpolation requires finding a coefficient for every function parameter and a constant term, requiring $N+2$ coefficients to approximate a function with N state variables and 1 desired variable. Since the current implementation limits itself to exactly two points, SPLICE does not solve for so many free variables. Instead, SPLICE only considers the initial value of the desired variable, i, giving the equation

$$u = f(i, d) = \alpha + \beta i + \gamma d.$$

Chapter 8 explores possible approaches to eliminating this restriction.

This function approximation technique is similar to many numerical approaches described in Chapter 2, such as spline fitting and regression. The difference is that the total response function is distributed among an arbitrary number of cases, instead of defined by a fixed set of parameters. SPLICE is closer to nonparametric techniques [54] that build a number of local models instead of a single monolithic global model.

SPLICE computes the parameters α, β, γ with the two points associated with the patch,

$$P_1 = (i_1, f_1, u_1), P_2 = (i_2, f_2, u_2),$$

and some general knowledge. SPLICE needs some extra knowledge to find u because two points gives two equations in three unknowns. SPLICE transforms the problem into two equations in two unknowns by interpolating over a new variable x_j, defined by translating each point to the initial value for this problem,

$$x_j = f_j + (i - i_j).$$

Since the x_j is assumed to be the result of applying control u_j at i, the response function is reduced to two free parameters,

$$u = f(d) = \alpha + \beta d,$$

So, solving for α and β and simplifying, the equation relating current and desired states to control adjustments is

$$u = f(d) = \frac{-u_1\, d + u_2\, d - u_2\, x_1 + u_1\, x_2}{-x_1 + x_2}.$$

A sample interpolation is in Figure 3.4. In this figure, the agent translates P_1 and P_2 to the initial value of the desired variable, then linearly interpolates to find the intersection with the desired value. The intersection point gives the interpolated control value for this problem.

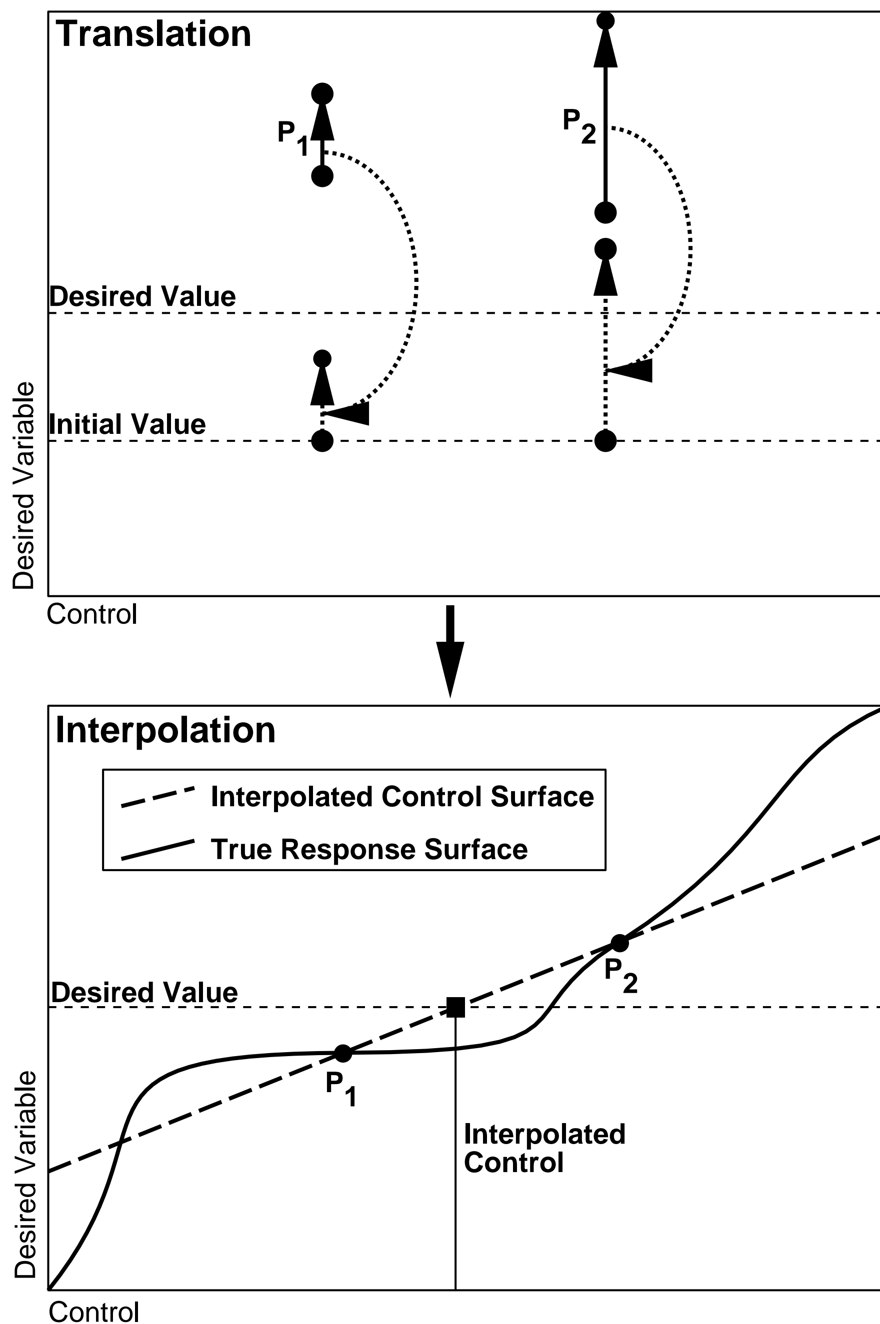

Figure 3.4: Sample translation and interpolation.

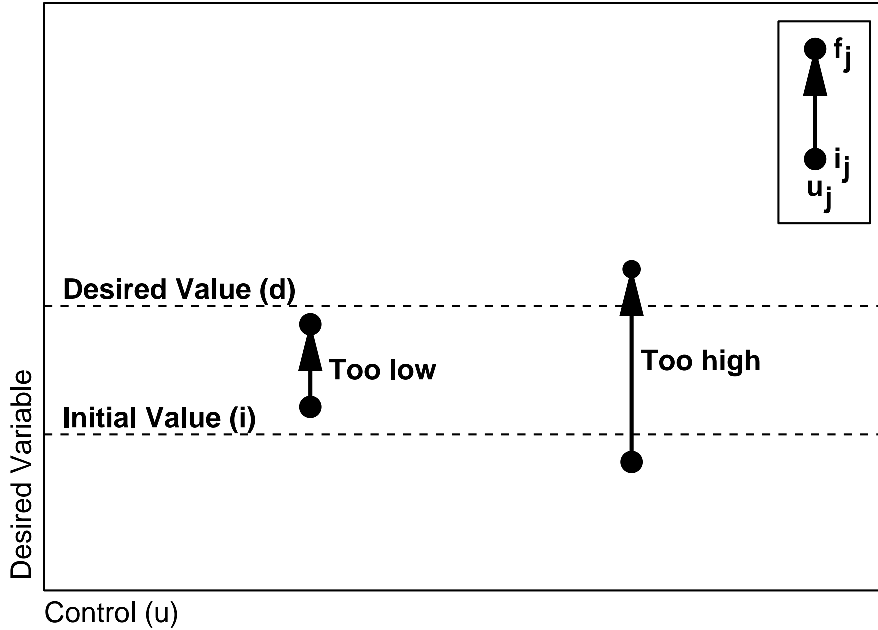

Figure 3.5: Graphical depiction of the control heuristics.

Although the resultant control signal u is a linear interpolation for the current state and goal, it may not be an allowable control adjustment. It may exceed the maximum or minimum possible control settings, which means u needs to be clamped to the maximum or minimum. Some control variables in some environments also may only allow settings at whole numbers or other restrictions, which means that u needs to be rounded to the nearest acceptable value.

After clamping and rounding, the candidate control u may match a previously experienced control u_j. If so, it may be possible to qualitatively predict the outcome of the control movement. Assuming a positive relationship between control and desired variable, if the desired value d is less than the final value f_j and the current value i is greater than or equal to the initial value i_j, the control u will certainly be too high and SPLICE decrements the control to the next acceptable lower value. Similarly, if $d > f_j$ and $i \leq i_j$, then u is too low and SPLICE increments it. If the relationship is negative, the heuristics change to comparing $d > f_j, i \geq i_j$ and $d < f_j, i \leq i_j$. If one heuristic applies for P_1 and the other applies for P_2, it means than the goal is unachievable for that control resolution, and SPLICE chooses the control with the closest final value. Note that these heuristics are only useful if $u = u_j$, because the linear interpolation otherwise ensures that u is qualitatively correct. Figure 3.5 illustrates the heuristics. Note that these heuristics also assume that the response function is monotonic. Chapter 8 covers relaxing the monotonic restriction.

When SPLICE settles on a control adjustment, either from a previous success, linear interpolation, or a correction to the interpolation based on a predicted qualitative failure, it executes the control action in the environment and waits for the deadline.

3.4 Knowledge Requirements

SPLICE was designed to be implementation-independent and domain-independent, but it needs a certain amount of knowledge for a particular implementation and a particular

domain. This section will summarize the operation of the Performance Element from the perspective of necessary knowledge.

The first type of domain-specific knowledge is the agent goals in a particular domain. The variables available in a domain and the possible values of those variables restrict the possible agent goals, and the higher-level oracle or user objectives define the exact goals SPLICE is to achieve. For example, a user may want to achieve a particular sequence of states. In this case, as soon as the agent achieves a state, the user presents the next state in the sequence as the next desired state. Or, the user may want to obtain a complete response surface for the environment. This means that the user must prepare a subset of problems distributed over the entire set of possible problems.

When the agent discretizes the initial and desired values, it needs some domain-specific knowledge for control. This is collectively the *representation scheme* for each variable, and includes information such as the range of possible values for the variable and the most specific level of detail necessary for each variable. This is necessary not only to prevent infinite region sets, but also to avoid dealing with the environment where it becomes nondeterministic and random. For instance, in the car example, the least significant region for speed is 5, which translates to a minimal range of 8. This may be because the speedometer is too imprecise for better control, or because inherent noise in the environment makes better control impossible.

Although the case-indexing strategy is fully-specified, domain knowledge is a critical component of the qualitative reasoner. Domain knowledge provides the qualitative model that predicts the effects of control adjustments. The model is a set of binary qualitative relationships, either positive or negative, between a causal variable and an effected variable.

The final domain-specific knowledge requirements are the characteristics of the control variables, which are similar to the representation schemes of the state variables. SPLICE needs to know the maximum and minimum possible values for each control, and the rounding function, if any.

With the specification of the above domain knowledge, the Performance Element of SPLICE is capable of performing in the environment. Its ability to achieve its goals depends on the accuracy of the linear model of the matched case. For complex non-linear environments, a particular local model is only accurate for a small area of the problem space. This implies that the agent performs best with specific cases containing accurate models for the covered portion of the problem space. The learning mechanism described in the next chapter deals with creating more specific cases with more accurate models.

CHAPTER 4

Learning

When I grow up, I wanna write a book,
But I wonder if I'll know how.
I guess I'll just keep learning a little at a time,
And I think I'll start right now!

The previous chapter presented the Performance Element for SPLICE. The PE uses a case database containing generalized problems, and points for interpolation associated with these problems. Although it can generate some cases from the qualitative domain model, SPLICE requires a Learning Element to incrementally add cases to the database as the agent gains experience in its environment. Since cases consist of both a generalized problem and two points for interpolation, learning a new case involves creating a new generalization ($Gen_{I,D}(i,d)$) and associating it with two points. Figure 4.1 shows the place of the LE in SPLICE.

4.1 Creating Cases

After the PE executes a control adjustment in the environment, SPLICE begins the countdown to the deadline. At the deadline, SPLICE observes the final state of the world F. The completion of the deadline presents a new point from the environment, $P = (I, F, U)$[1]. This point will be associated with the new cases because SPLICE needs data from the environment to improve its performance. Executing an action in SPLICE actually allows the agent to learn information related to two problems:

1. How to solve the original problem (I, D).
2. How to solve the "false sense of accomplishment" problem, (I, F), where SPLICE thinks it really meant to achieve F and was successful.

This information is summarized as two cases, called the "goal-directed case" (GDC) and the "false sense of accomplishment case" (FSAC). Of course, when SPLICE successfully achieves its goal ($F = D$), the two cases collapse into one. Creating a new case requires choosing a generalization for the current problem and two points. If the generalization is more specific than an existing case and it matches some of the problems also matching the

[1]Since this point is only valid for the given deadline, it should not be used as is for a different deadline. Chapter 8 discusses the necessary changes to allow varying deadlines.

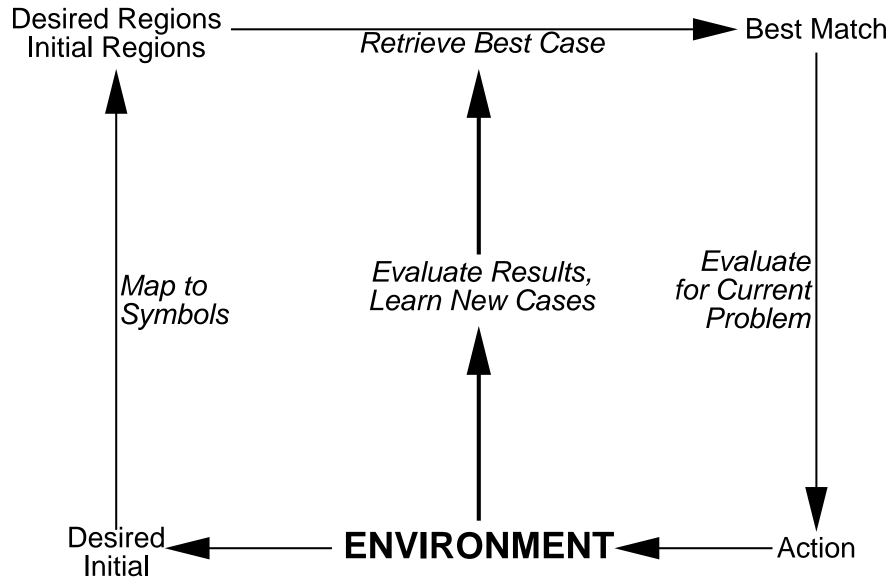

Figure 4.1: Overview of SPLICE with learning added.

existing case, the generalization *masks* the existing case for those problems. There are a number of constraints on the generalization:

1. Since the case should apply to many problems, the generalization should be as general as possible.

2. Since the PE chooses the most specific matching case M, the generalization must be more specific than M. If it was not more specific, M would mask out the new case for all problems and learning would not affect performance. This is because the matching algorithm selects the most specific case matching the current problem, so it would select M over any more general case.

3. The generalization should be general enough to mask out incorrect knowledge, and specific enough not to mask out correct knowledge. Although it is impossible to unequivocally determine from a single experience what knowledge is correct and what is incorrect, it is possible to measure to what level of generality the current knowledge achieves the goal, and only learn beyond that level of generality. This is explained in more detail in the following sections.

4. To ensure that the new case is as general as possible, the generalization will be the same as M except adding some new regions to one selected variable.

Since learning a case is actually a specialization of one variable of the most specific match M, all the LE needs to do is find M using the region subdivision procedure of Chapter 3, and select a variable region to specialize. Each type of case performs specialization differently, and they are described separately in the following sections. However, each type chooses associated points in the same way, so the final section of this chapter describes choosing points. Another commonality between cases is that the LE will not retain the case if the experienced point P is identical to one of the points associated with M, P_1 or P_2. This can happen if the agent experiences the exact situation in which it learned M, or if the generalization for the FSAC includes the original problem (I, D).

4.2 The False Sense of Accomplishment Case

In the FSA case, SPLICE imagines that the final state is actually the desired state, and proceeds to learn a case to ensure that if the agent is ever presented the problem (I, F), it will solve it correctly. Moreover, case generalization will allow this case to match problems similar to (I, F), so the performance can also be expected to improve for problems similar to (I, F). After gaining enough experience so that the problem space is sufficiently dense with cases, every problem will be similar enough to some case so that it can be achieved.

Most adaptive control algorithms emphasize learning (I, F). Statistical regression recomputes the fitting function parameters by including the points (I, F, U). This can be an expensive process, but the resulting functional representation is usually compact and efficient for evaluation. Nearest-neighbor approaches simply add the new point to the table of points. This is very inexpensive, but later queries require searching the entire table for the nearest points. SPLICE has superior algorithmic complexity because it both learns in constant time and retrieves cases in constant time. Learning is constant time because SPLICE retrieves the best match, adds a feature, replaces a point, and stores the case. Retrieval is constant time because SPLICE represents the cases as Soar rules which match in parallel.

To summarize from the previous section, the selected region must be as general as possible, yet not so general that it conflicts with its best match M. This implies the measure *Most General Difference*, or MGD.

Definition 8 *The most general difference between two values of the same variable is the most general region in which the two values differ,* $MGD(x_i, y_i) = \arg\min\{(x_{i,j}, y_{i,j}) | x_{i,j} \neq y_{i,j}\}$.

Since the best match M is a generalization of an individual point experience P intended to correct or improve the local model for P, the new case may mask part of M but not P. The MGD regions, as illustrated in Figure 4.2, for each variable in P are the maximum generalization not masking P. Since it is only necessary to specialize one variable, the selected variable s and selected region r should be the most general of the MGD's, where the current variables are $x_i = value_i \in I$, $x_0 = d \in D$, and the matched variables from $M = (I_M, D_M)$ are $y_i = value_i \in I_M$, $y_0 = d \in D_M$. SPLICE computes this as

$$\begin{aligned} s &= \arg\min(MGD(x_0, y_0), \ldots, MGD(x_N, y_N)) \\ r &= \min(MGD(x_0, y_0), \ldots, MGD(x_N, y_N)). \end{aligned}$$

The only situation where the new case completely masks M is when all variable values for the current and past points are within the same least significant region. This means that the entire case M is incorrect and needs to be updated.

The new case FSAC becomes the best match specialized by the most general MGD, $FSAC = M \cup \{x_{s,j} | j \leq r\}$. This process essentially selects the variable which is most different from the best matching case. To prevent the selection of irrelevant variables, SPLICE limits selection to variables known to be relevant from the domain model. More formally, a variable v is defined to be relevant to a control u and a desired variable d if v lies on a path from u to d in the qualitative domain model for the environment.

Note that the agent solved (I, F) using its current knowledge, so one may ask why the agent needs to learn anything if it can already solve the problem. The first response to this objection is that the agent solved the problem while trying to solve (I, D), which may have

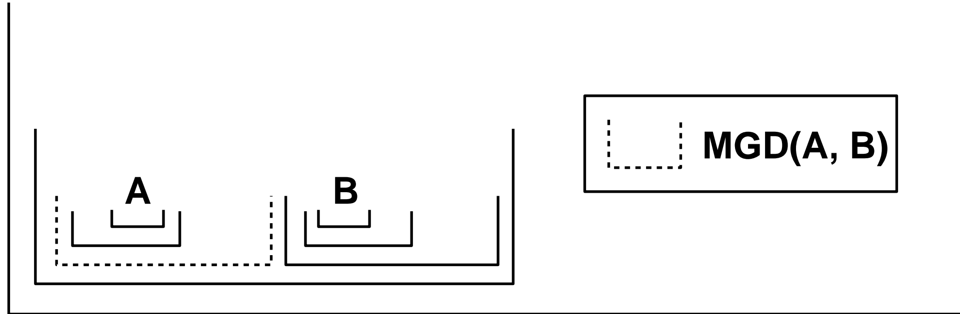

Figure 4.2: Illustration of Most General Difference.

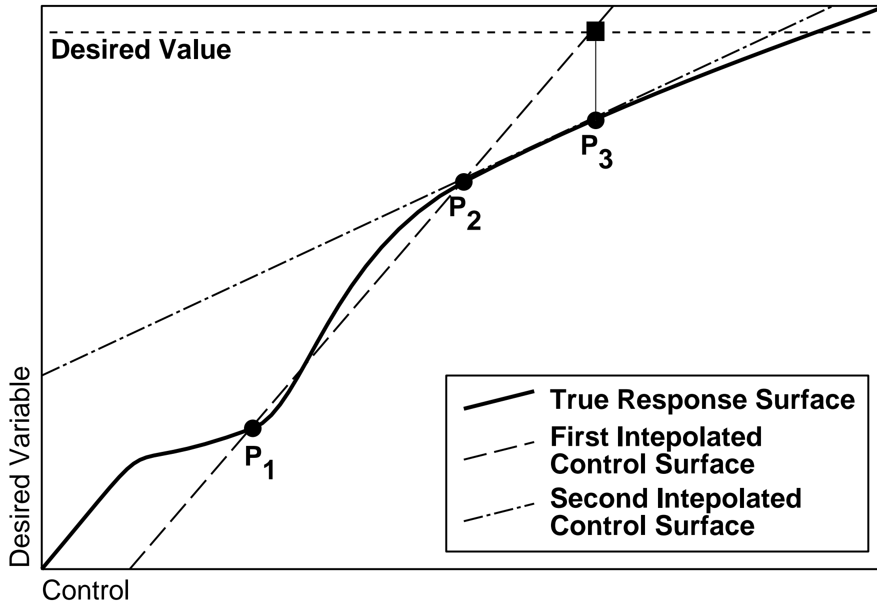

Figure 4.3: Change in response surface upon learning a new point.

been very different. In this case, the LE must at least transfer the necessary knowledge to the region around (I, F). The second response is that learning the point (I, F, U) will still allow the agent to solve (I, F) (now without interpolation), and it may be more accurate for problems similar to (I, F), because the new point is more local for these problems. Figure 4.3 displays the advantage of learning a new point around F (For readability, this figure assumes I is constant). After retrieving points P_1 and P_2 for a desired value, the agent interpolates and finds P_3. Now the interpolation (P_2, P_3) is much more accurate for the region around P_3.

Recall that in the car example, the initial speed was 0, the desired speed was 65, the pedal pressure was 10, and the final speed was 50. The FSAC problem to generalize is $(0, 50)$. From the cases in Table 3.2, the nearest match is case $A = Gen_{5,40}(3, 1)$. Since $MGD(0, 5) = \emptyset$, because they are the same down to the least significant range, and $MGD(50, 40) = 4$, because 48–63 $\neq$ 32–47, the new case $D = Gen_{0,50}(3, 4)$.

4.3 The Goal-directed Case

The GD case allows the agent to perform differently the next time the agent is presented a similar problem P. Without the GDC, the agent would continue using the same incorrect linear model for P unless the generalization of the FSAC happened to include P. The GDC is an explicit attempt to improve the agent's performance for problems similar to P. Computing the GDC is a critical difference between SPLICE and other adaptive control algorithms. Although function approximation methods incorporate the experienced point to improve the response surface, they do not explicitly consider the goals of the agent.

Computing the GDC is similar to computing the FSAC. Again, the case should be general enough to apply to unsolved cases similar to P, but not as general as the best match M. If the agent fails to achieve its goal using match M, at least one of two things must have gone wrong:

1. The points (P_1, P_2) were not close enough to P to be an accurate linear interpolation (Achievement Failure).

2. One or more variables in the Initial State I changed enough to force U to fail to achieve D (Expectation Failure).

The LE first tests for an expectation failure, and if there is no expectation failure it assumes an achievement failure.

4.3.1 Expectation Failures

Expectation failures occur when the monotonic assumption is violated.

Definition 9 *The monotonic assumption is that the relationship between U and F is either monotonically increasing or decreasing (based on the qualitative physics analysis) for approximately equal initial conditions.*

If it is not possible to draw a monotonic curve through the projections onto the (U, F) plane of the recalled points P_1, P_2 and the experienced point P, the curve is inconsistent with the monotonic assumption. Assuming that the environment is monotonic, the initial values of the experienced point and the recalled points must not be similar enough, according to the definition. Therefore, if an experienced point is in the grey region of Figure 4.4 (assuming that the relationship is positive), there has been an expectation failure.

In the example, the best match for the problem $P = (0, 65)$ is case B from Table 3.2. The points associated with this case are $P_1 = (5, 30, 5)$ and $P_2 = (5, 25, 4)$. The projection of P_1 is (5, 30) and P_2 is (4, 25). The projection of the current point (U, F) is (10, 50). Since $10 > 5$ and $50 > 30$, the point lies in the upper right consistent region of Figure 4.4, so it is consistent with the monotonic assumption and there is no expectation failure. If P_1 were $(40, 65, 5)$, there would be a monotonic violation because $50 < 65$, so the agent would conclude that the initial speeds were not similar enough, and the new case would specialize the initial speed.

Once the LE finds an expectation failure, it must specialize some initial state variable to distinguish it from the initial state of the matched case. The LE first prunes all nonrelevant variables, then finds the variable with the most general MGD $x_{s,r}$ like for the FSA case. The new GD case is the best match M specialized by $x_{s,r}$.

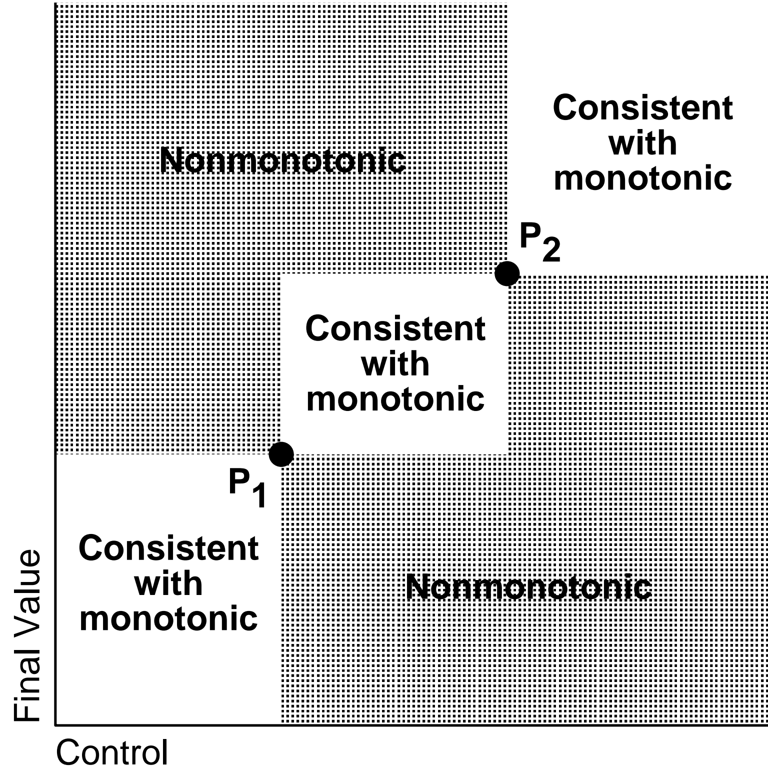

Figure 4.4: Detecting nonmonotonic trends.

4.3.2 Achievement Failures

If there was no expectation failure, but the agent did not satisfy its goal, the agent must use the point it has experienced to improve its performance the next time it is presented a similar problem. However, the agent only needs the additional point to solve the part of the goal that it failed to solve with the knowledge it already had. This means that the new GD case with a new point should only apply in cases where the goal is too specific to be achieved with the current database. Therefore, the new GD case is the best match M specialized by the most general region which was not achieved, $MGD(d, f)$, where f is the final value of the desired variable. If $MGD(d, f)$ is more general than the level of generality of the desired variable for the current best match, the new case keeps the same level of generality, effectively replacing the best match. In this situation, the best match was not able to achieve its most specific goal, so it is not useful and should be discarded.

Returning to the car example, $MGD(65, 50) = 2$, because 64–127 $\neq$ 0–63. Since the best match is $Gen_{5,65}(3, 4)$, the agent replaces this case with $E = Gen_{0,65}(3, 4)$.

4.4 Choosing the Action

Once the LE has built a new case, it must associate a pair of points with it. One point must be $P = (I, F, U)$ to allow progress to the goal. The other point may be either one of the points associated with the original match to the problem, $P_1 = (I_1, F_1, U_1)$ or $P_2 = (I_2, F_2, U_2)$. The LE goes through a series of filters to choose which point to keep.

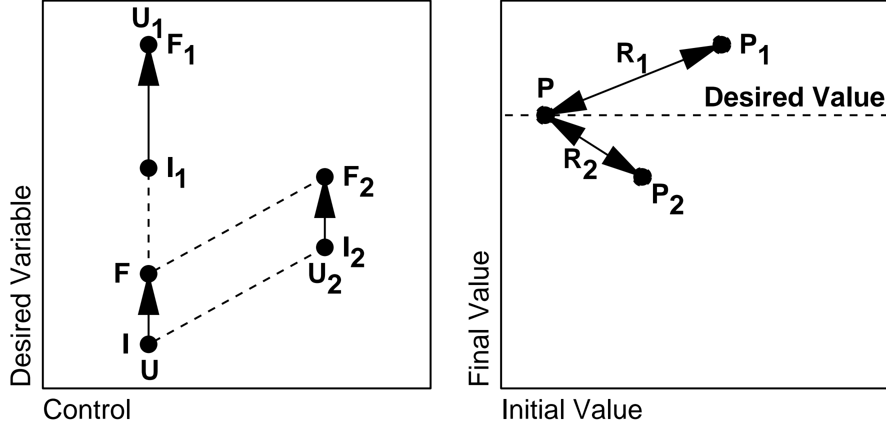

Figure 4.5: Point selection heuristics. (left) Points failing the delete filters. (right) Computing the residuals.

1. If (without loss of generality) the domain model generated P_1 and the agent directly experienced P_2, the agent will retain P_2. This filter is necessary to ensure that the PE only uses domain model points in the absence of directly-experienced points.

2. If $F - I = F_1 - I_1$, and $F - I \neq F_2 - I_2$, the agent will retain P_2. This filter avoids later interpolation problems. Since the agent interpolates over state differences (Section 3.3), if both points have the same difference the line through the points will be horizontal and not intersect with the desired value unless the desired value is $D + F - I$. Point P_2 in Figure 4.5 matches this filter.

3. If $U = U_1$, $U \neq U_2$, and the domain model did not generate P_2, the agent will retain P_2. This filter also avoids later interpolation problems. If the actions of both points associated with a case are equal, the line through the points will be vertical, and the interpolated action will be the same for any goal. The only case where it may be better to have two identical actions is if the other option is to have a domain model point. In this case, the linear interpolation would be worthless, but the high and low heuristics described in Section 3.3 should be able to predict failure and try the next higher or lower control. The LE will include this action in the next cases, and the next time the agent receives a similar problem, it will do a valid interpolation because the actions will be different. Point P_1 in Figure 4.5 matches this filter.

4. If both points are usable for linear interpolation, the agent chooses the point that seems most likely to reside on the same linear patch of the response surface as the current point P. The agent computes a residual error

$$R_i = \sqrt{(I_i - I)^2 + (F_i - D)^2} \tag{4.1}$$

   and retains the point with the lower R_i. This ensures that the agent will retain the overall closer point to the initial and desired states. Figure 4.5 illustrates the residual computation.

It is theoretically possible for one point to fail one filter and the other point to fail the same filter or another one, for example if $F - I = F_1 - I_1 = F_2 - I_2$. If this occurs the

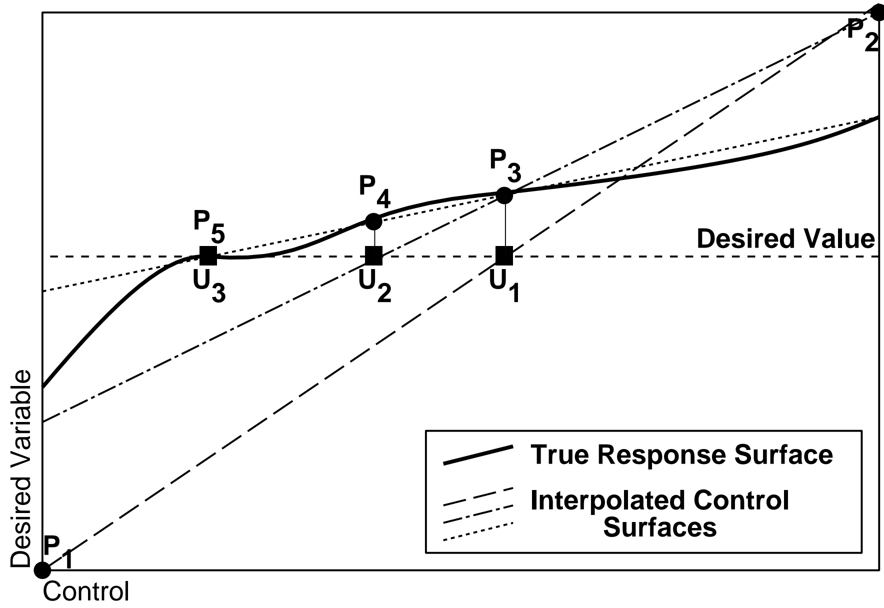

Figure 4.6: Progress toward achieving a goal.

agent is unable to learn a case and make progress toward its goal. Although this was never observed in the experiments, the possibility can be removed in a number of ways. The most promising is to decrease sensitivity to individual points by retaining a number of points for each case. This is covered in Chapter 8. In practice the agent's typical progress is illustrated in Figure 4.6. P_1 and P_2 are domain points. The agent interpolates to find U_1, but after executing this control in the environment receives P_3. After again failing to achieve the goal with U_2, the agent interpolates again and finally achieves the goal with $U_3 = P_5$.

As the final step in the car example, the agent must choose whether to retain $P_1 = (5, 30, 5)$ or $P_2 = (5, 25, 4)$ along with $P = (0, 50, 10)$. Neither point is from the domain model, neither difference is 50 (they are 25 and 20), and neither action is 10 (they are 5 and 4). The residuals for the goal-directed case ($D = 65$) are $R_1 = \sqrt{5^2 + 35^2} = 35.355$ and $R_2 = \sqrt{5^2 + 40^2} = 40.311$, so the agent retains P_1. The residuals for the FSA case are computed for $D = 50$.

This learning technique is based on the assumptions that the dynamics of the environment will not change over time, and that generalization of problems (initial state and desired state) to future similar problems is helpful in solving future problems. In environments where these assumptions hold, SPLICE is able to improve its performance to the point where it can solve any problem (I, D), meaning:

- If it is possible, SPLICE can eventually achieve D within the least significant region of D.
- If the resolution or the range of the controls do not allow the agent to achieve D, the agent will get as close as it can.

SPLICE is even able to perform to some degree in environments where the above assumptions are not true. If the environment changes over time, SPLICE can compensate by its natural removal of points, so the agent can clean out old points no longer valid in this environment. If the environment is so unpredictable that it is not always possible to

generalize to other problems, SPLICE still remembers how to solve problems it has solved in the past, so it can still improve its performance.

4.5 Proof of Convergence

This section shows that the performance and learning mechanisms described in this thesis guarantee that SPLICE will converge on a solution given a few assumptions about the environment. The first step is to define the environmental assumptions for this proof. The next step is to define the behavior of SPLICE over a number of cycles. The final step is to show that this behavior leads to a reduction in error to an arbitrarily small ϵ.

4.5.1 Environmental Assumptions

The environment for this proof includes one desired variable and one effector. We assume that the initial state is the same for each cycle so it is irrelevant. The effect of a control adjustment u is the final value of the desired variable $y = f(u)$. To ensure that there is a solution u^* for any desired value y^* such that $f(u^*) = y^*$, f is invertible. This also implies that f is monotonic and continuous. We also assume that there is no rounding or range limitation on controls u. Note that f is not necessarily smooth. Piecewise linear functions, for example, are acceptable.

4.5.2 Time Behavior of SPLICE

SPLICE begins its first cycle with two domain points. Because of the heuristics for retaining points, both domain points are eliminated by the third cycle. By this time, the most specific case for desired value y^* contains two points on the function f, $(u_1, f(u_1)), (u_2, f(u_2))$. The agent interpolates to find a new control u_3 and executes it to get a new result $f(u_3)$. The agent retains the point $(u_3, f(u_3))$ and $(u_j, f(u_j))$ such that $j = \arg\min\{|f(u_1) - y^*|, |f(u_2) - y^*|\}$. In other words, for each cycle $n > 3$, the agent interpolates with the last point $(u_{n-1}, f(u_{n-1})$ and the closest point $((u_j, f(u_j))$ such that $j = \arg\min\{|f(u_j) - y^*|\}, 1 \leq j < n-1$. This sequence is similar to the Newton-Raphson root-finding algorithm (e.g., [13], p. 193), with a slight difference in calculating the derivative.

4.5.3 Proof

Given the environmental function f and the behavior of SPLICE, we will prove that SPLICE converges to a final result y such that $|y - y^*| \leq \epsilon$. We will prove this by showing that the error $e_i = |y_i - y^*|$ for cycle i always decreases after some number of cycles. Refer back to Figures 4.6 and 4.3 for graphical displays of some possible situations. First we prove two lemmas.

Lemma 1 *For any two points $(u_1, y_1), (u_2, y_2)$ where $y_1 < y^* < y_2$, the linear interpolation at y^*, u_3, will lead to a result $y_3 = f(u_3)$, where $e_3 < e_1$ or $e_3 < e_2$. Also, if $e_3 > e_j$, y^* is between y_3 and y_j.*

Proof of Lemma 1: WLOG assume f is monotonic increasing. Then $u_1 < u_3 < u_2$, and $y_1 < y_3 < y_2$. If $y_3 < y^*$, $e_3 < e_1$ because $y_1 < y_3$. If $y_3 > y^*$, $e_3 < e_2$ because $y_3 < y_2$. For the second part, if y_3 is on the same side of y^* as y_j, $e_3 < e_j$ because of monotonicity. By contradiction, y^* is between y_3 and y_j.

Lemma 2 *For any two points* $(u_1, y_1), (u_2, y_2)$ *where* $y_1 < y_2 < y^*$*, the linear interpolation at* y^**,* u_3*, will lead to a result* $y_3 = f(u_3)$ *either closer to* y^* *than* y_1 *and* y_2*, or greater than* y^**. The same is true for* $y_1 > y_2 > y^*$*.*

Proof of Lemma 2: WLOG assume f is monotonic increasing. Then $u_1 < u_2 < u_3$, and $y_1 < y_2 < y_3$. If $y_3 \leq y^*$, $e_3 < e_2 < e_1$. Otherwise $y_3 > y^*$. For $y_1 > y_2 > y^*$, $u_1 > u_2 > u_3$, and $y_1 > y_2 > y_3$. If $y_3 \geq y^*$, $e_3 < e_2 < e_1$. Otherwise $y_3 < y^*$.

Proof of theorem: On cycle n of SPLICE, it will interpolate using two points: its last result P_{n-1} and its best result so far P_{best}. Given any two points, the desired value will either be less than both final values, greater than both final values, or between the final values.

If the desired value is between, the result y_n will be better than y_{n-1} or y_{best} by Lemma 1. WLOG suppose $e_{n-1} < e_{best}$. If $e_n < e_{n-1}$, the error is decreasing. Otherwise $e_{n-1} < e_n < e_{best}$. For the next cycle $n+1$, the points are P_n and P_{n-1} and the desired value is between the points, by the second part of Lemma 1. If the error for this cycle $e_{n+1} < e_{n-1}$, the error is decreasing. Otherwise $e_{n-1} < e_{n+1} < e_n$. As the sequence continues, e_{n+m} approaches e_{n-1}, so eventually $e_{n+m+1} < e_{n-1}$ and the error is decreasing.

If the desired value is on one side of both final values, the interpolation will either be better than both final values, or on the other side of the desired value, by Lemma 2. If the interpolation is better, the error is decreasing. If the interpolation is on the other side, SPLICE will store this point along with the better of the two points, so the desired value will be between the final values in the next cycle, which has already been proven.

4.5.4 Relaxing Assumptions

It is impossible for any root-finding algorithm to be provably correct for discontinuous functions, and any root-finding algorithm may fall into a local minimum for nonmonotonic functions, so SPLICE is forced to make these assumptions. However, the domain model points provide hints for the correct starting position, and the interpolation may propel the system arbitrarily to different parts of the function, so it may be possible for SPLICE to solve these problems, in contrast to gradient methods which are unable to climb out of local minima. SPLICE also reasons about varying initial and desired states, so it is capable of generalizing knowledge learned from one problem to helping solve similar problems. It is interesting to note that linear interpolation is not strictly required for this proof, any method of selecting controls that satisfy the basic properties of a linear interpolation is sufficient. The interpolation lesion in Chapter 5 finds that in practice, other techniques can be successful.

4.6 Knowledge Requirements

Since learning in SPLICE is almost entirely empirical, it requires no domain knowledge beyond that already required for performance. The primary knowledge source utilized by the LE is the most specific case matching either the actual result (I, F) or the desired result (I, D). Since SPLICE learned this case in an earlier session, it improves its performance by bootstrapping from the initial qualitative physics domain model. SPLICE specializes this case by finding the most general MGD among its relevant parameters. The relevant parameters are also a part of the qualitative domain model. Finally, SPLICE retains one point from its best match, which may be a domain model point, and adds the data point it

experienced from the environment.

4.7 Properties of the Learning Component

The learning component continually adds more and more specialized cases, beginning with very general cases which may not solve many problems. As the agent finds general cases which fail on specific problems, the agent adds a more specific case for the problem on which the general case succeeded (the FSAC), and a more specific case for the problem that failed (the GDC). These more specific cases are more accurate for the problems they cover, so the overall accuracy of the agent improves as specific cases cover old less accurate cases. Learning finally halts when SPLICE stops seeing new data points, but performance can continue as long as the user presents goals to the PE.

In many systems that perform tests of differing generality, such as decision trees [46], more specific cases match slower because the algorithm requires more tests. However, the SPLICE implementation is based on SCA [36] in Soar [29]. In SCA, more specific matches are actually faster because less abstraction is necessary. The only possible source of slowdown in performance is the growing number of cases, represented as Soar chunks. The state-of-the-art Soar matching facility has empirically found that match cost does not suffer when learning up to 100,000 chunks [12]. Since no SPLICE agent has ever learned more than 1000 chunks in its lifetime, the number of rules is not a significant factor in execution time. Over time, the learning component possesses two desirable properties: increasing accuracy and decreasing execution time.

CHAPTER 5

Analysis of SPLICE Execution Properties

And when we grow a little older we can do more things,
Because I'm growing, and so are you!

Now that the description of SPLICE is complete, it is possible to examine and analyze SPLICE in action. The last component necessary for experiments is an environment. Although SPLICE was designed for complex domains not easily represented by an analytical response function, experiments on simpler domains allow more complete control. This chapter describes three artificial domains and SPLICE's results with them. We concentrate on analyzing SPLICE through aggregate data, which requires specifying the methodology for presenting training data and evaluating performance. This chapter covers four experimental methodologies for training and evaluating SPLICE. The analysis itself consists of presenting the performance improvement of SPLICE in each of the domains using each of the methodologies, and a study of secondary effects such as the development of the case database. This chapter concludes with comparisons to a series of lesions and design alternatives that demonstrate the source of power for SPLICE's performance. Although this chapter is concerned with aggregate characteristics and not how SPLICE learns to solve particular problems, there are traces in Appendix A that give a detailed description of how SPLICE operates.

5.1 Artificial Domains

Adapting SPLICE to a specific domain involves choosing a domain SPLICE is capable of learning and creating the necessary initial domain-specific knowledge. SPLICE is liberal on the types of environments in which it can operate. Several heuristics assume the desired variables are monotonic, but it can still interpolate and perform in nonmonotonic environments. Like all function approximation algorithms, it does not guarantee success for discontinuous environments, but its representation does not assume continuity. SPLICE's initial knowledge requirements are the name and allowable range for all possibly relevant variables in the environment, a way to communicate with the environment, and the qualitative model of the environment. These values all have intuitive meaning for the user, so they do not constitute an unusual burden of initial knowledge.

The artificial domains chosen for these experiments were variations on controlling an automobile. SPLICE had control of the gas pedal and had to achieve a desired speed after a specific time interval. It was possible to move to a negative pedal position to decelerate and achieve negative speeds. Direction, wind, load, and terrain were all constant, so the

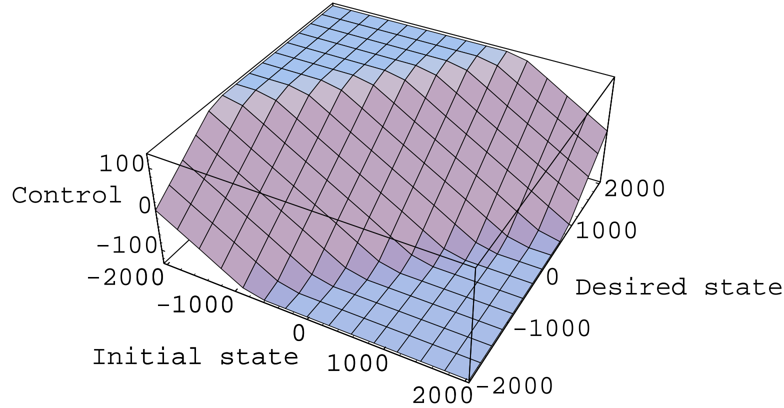

Figure 5.1: True response surface of Domain 1 (linear).

only relevant variables were the pedal position and the speed. The SPLICE agent computed a pedal position based on its initial and desired speed, kept that pedal position constant until the end of the interval (the deadline), and evaluated its results. Its initial domain knowledge was that the pedal position and speed are positively correlated, i.e. a positive pedal increases speed.

5.1.1 A Linear Domain

The first test domain was a simple linear relationship. After the interval Δt, the speed $s(\Delta t)$ was defined to be 10 times the control p plus the initial speed $s(0)$, or

$$s(\Delta t) = s(0) + 10p.$$

Figure 5.1 graphs the true response surface of this domain. The extremes of the domain are not linear because the throttle is limited to a range of -128 to 127. This domain was intended to be extremely easy for SPLICE because it already uses a linear interpolation over the response surface, so it just needs to learn the correct coefficients.

5.1.2 A Non-linear Domain

The second test domain was a more complex non-linear relationship intended to closer approximate reality. The relationship between the pedal position and the carburetor opening became a function of the square root of the pedal position, and the force acting on the automobile was reduced by simulated wind resistance proportional to the speed of the vehicle. The final speed at the deadline Δt is the initial speed plus the total acceleration. The total acceleration is a function of mass (constant), engine thrust (proportional to the square root of the pedal position), and wind resistance (proportional to the speed of the vehicle). The final domain equation is

$$s(\Delta t) = s(0) + \int \frac{k_1\sqrt{p} - k_2 s(t)}{mass}\, dt.$$

Figure 5.2 graphs the true response surface of this domain. The throttle limits are the same as in the linear domain. This domain was intended to be more difficult than the linear domain because the response surface is curved, but once SPLICE learns enough specific cases, it can tessellate the response surface to planar rectangles with enough accuracy to satisfy the goals.

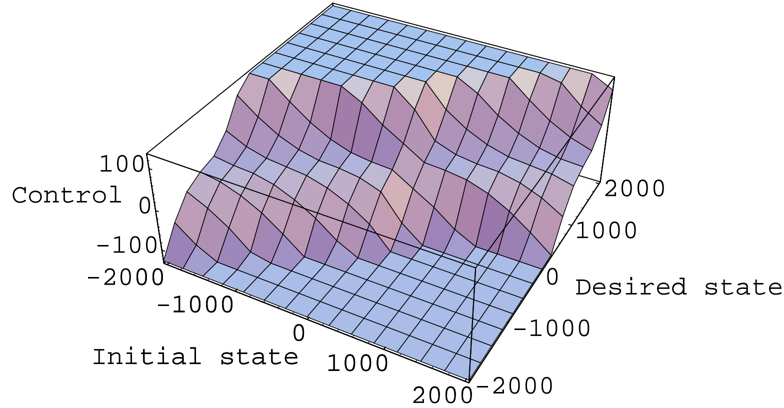

Figure 5.2: True response surface of Domain 2 (non-linear).

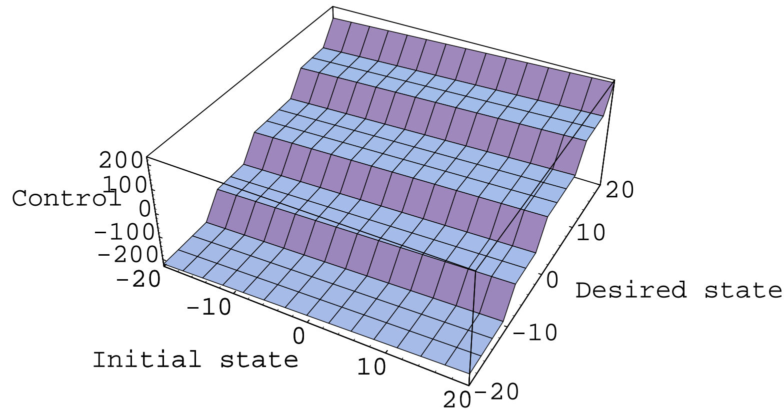

Figure 5.3: True response surface of Domain 3 (discontinuous).

5.1.3 A Discontinuous Domain

The third test domain was discontinuous. Discontinuous domains are common in environments affected by discrete variables, such as the transmission gear on an automobile. However, these environments are difficult for global models because of the abruptness of the discontinuities. The response surface for the domain chosen for testing is a set of planes with a slope of 1 separated by discontinuities. The domain equation is

$$s(\Delta t) = s(0) + 100\lfloor p/10.0 \rfloor + p \bmod 10,$$

as graphed in Figure 5.3. This domain was intended to be extremely difficult for SPLICE because it is explicitly not continuous at all points, so it is not necessarily true that two spatially approximate points share the same local linear model.

5.2 Experimental Methodology

SPLICE operates by receiving an initial state from the environment, deciding on a desired state, computing a response, learning by comparing the result of the response with the desired state, and repeating the process with possibly new initial and desired states. A *cycle* is a single iteration through this procedure. Experiments consist of some sequence of pairs of initial states and desired states. The procedure for choosing these sequences is the

experimental methodology. Methodologies can be characterized along several dimensions. Two are the extent to which the system is challenged, and the extent to which the sequence resembles a "real" task for the system. Realistic methodologies are valuable because they may be similar to real tasks assigned to the agent. Even challenging methodologies that are not realistic are valuable because they expose the limits of the system. For example, automobile manufacturers perform "torture tests" on their components even though they will never be subjected to those conditions in the field.

Like any algorithm, SPLICE can be expected to perform best on problems similar to or the same as problems it has previously experienced, so a sequence where every problem is different would exercise SPLICE more than a sequence where the problems are related, either in initial or desired states. More "realistic" problem sequences would minimize the frequency of "miracles," or going outside of domain laws to manipulate the environment. Measuring the realism of a sequence of desired states is more difficult, since the agent's goals are internal and do not necessarily have to obey physical laws. However, there are two intuitive procedures for changing goals. The first is to only allow one cycle for each goal. This corresponds to a hard real-time system [26] which must go on to the next goal by the deadline, whether the current goal is achieved or not. The logical evaluation metric for this procedure is to measure the difference between the final state and the desired state. The second is to continue to work on a goal until the agent achieves it or decides to give up. This corresponds to a soft real-time system which tries to achieve the goal as soon as possible, but continues working even if it does not immediately achieve the goal. The evaluation metric is the number of cycles necessary to achieve the goal, with infinity or some large value for failure to achieve the goal.

This thesis tests SPLICE on four combinations of the above methodological options. After describing the methodologies I will give them some intuitive meaning using a golf metaphor. The methodologies are:

Random-Hard This methodology gives a different random initial and desired state for each cycle. After a fixed number of cycles, the experiment is over. This method scores high on the difficulty metric because of the large variety of problems, and low in the realism metric because the initial state of cycle n is unrelated to the final state of cycle $n-1$.

Random-Soft This methodology begins with a random initial and desired state, but keeps the same desired state until it is achieved or the agent decides it is unachievable. At that point the agent goes on to a new random initial state and desired state. After completing all goals, the experiment is over. This is slightly less difficult than **random-hard** because the desired state remains constant for several cycles, and it is more realistic because the agent only jumps to a random initial state in the environment when it completes its goal.

Repeated-Hard This methodology is based on a sequence of realistic task-related goals. This sequence can be generated by a planning subsystem, or (as in this work) provided by hand. Since the experimental domains are automotive, the sequence is a series of desired speeds one might have driving to work in the morning.

1. Slowly back up out of driveway.
2. Drive down the street.

3. Slow down for a turn.
4. Return to driving speed.
5. Stop for a red light.
6. Return to driving speed.
7. Accelerate for highway.
8. Decelerate for exit.
9. Drive to parking spot.
10. Park.

The specific speeds varied for each domain, because the speeds were designed so that it is possible to go from one desired speed to the next in the time interval provided. Once the agent tries one goal, it continues directly to the next, without considering if it had achieved its goal or not. The initial state for each goal is the final state of the preceding goal, and the initial state of the entire sequence is stationary (speed = 0). After completing the sequence, the agent returns to stationary and begins the sequence again. To determine if the agent is still learning, the agent records how many cases it learns in the course of each complete sequence. The agent continues repeating the sequence until it does not learn a single case in an entire sequence. This methodology is less difficult than the random methodologies because goals repeat, and it is more realistic because the agent only jumps at the end of a sequence. Additionally, the last goal in the sequence is to park, so if the agent achieves this goal there is no jump at all.

Repeated-Soft This methodology is similar to **Repeated-Hard**, except the agent continues working on a goal until it achieves it or decides to move on. It may be less difficult than **Repeated-Hard** because it considers the same goal several cycles in a row, and it is slightly more realistic because the agent stills jumps at most once per sequence, and the sequences may last longer.

As a final description of the experimental methodologies, consider the game of golf. The initial state is the location of the tee, the desired state is the location of the hole, the effector is the speed, direction, and orientation of the club, and the final state is the location of the ball after hitting it once. For **Random-Hard**, the tee and hole locations are random, and after one swing the players move on to the next hole. The winner is the player with the lowest average distance from the hole for the course. Figure 5.4 illustrates this methodology. For **Random-Soft**, the tee and hole locations are random. The players tee-off and continue to play the ball where it lies until they are successful or scratch. The winner is the player with the lowest average number of strokes for the course. In terms of golf, this is the most realistic of the methodologies. Figure 5.5 illustrates this methodology.

For **Repeated-Hard**, the hole locations are an engineered sequence, and after one swing the players move on to the next hole, but the tee location is wherever the ball landed on the last swing. After completing the course, the players start again. The winner is the first player who gets a hole-in-one for every hole. Figure 5.6 illustrates this methodology. For **Repeated-Soft**, the hole locations are an engineered sequence, and the players tee-off and continue to play the ball where it lies until they are successful or scratch. Then they move on to the next hole, but the tee location is wherever the ball landed on the last swing. After completing the course, the players start again. The winner is the first player who gets

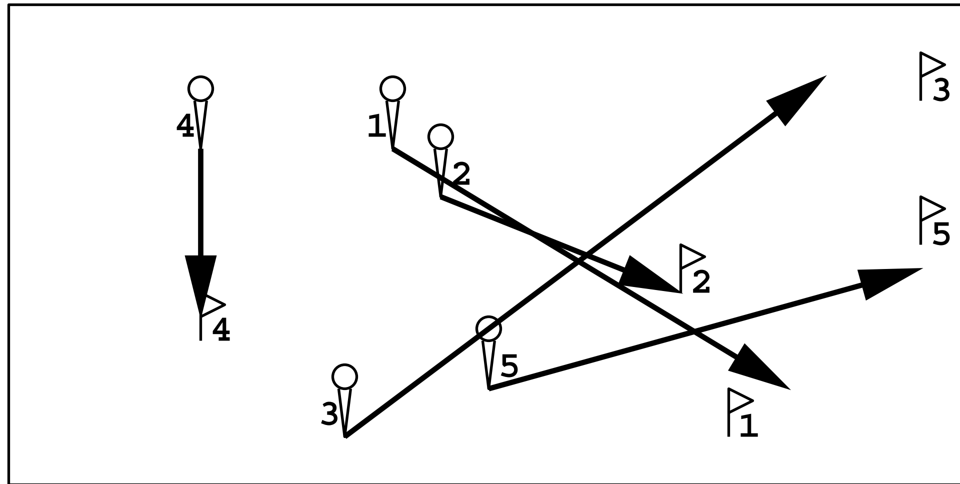

Figure 5.4: Playing golf by the random-hard rules.

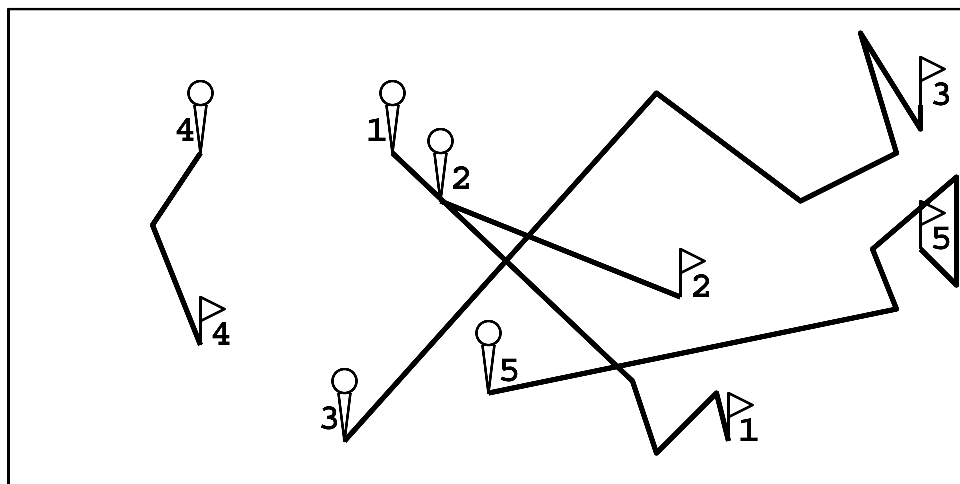

Figure 5.5: Playing golf by the random-soft rules.

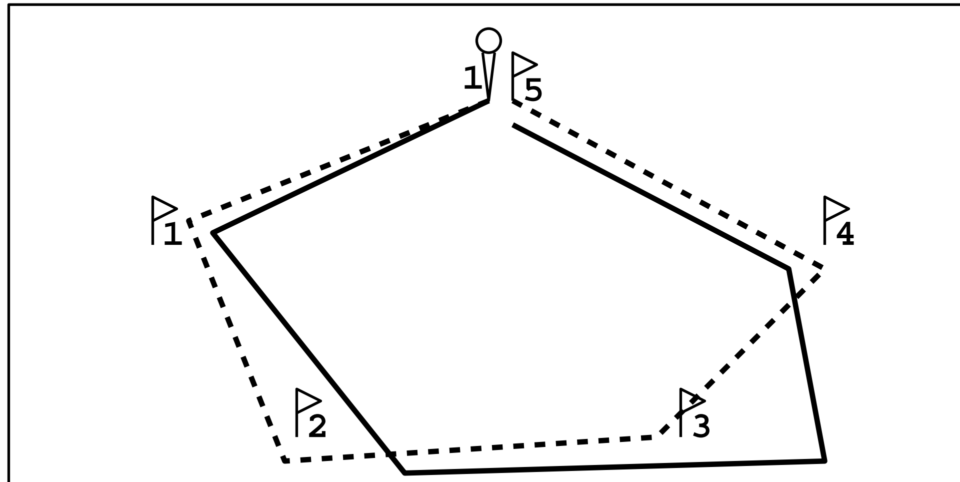

Figure 5.6: Playing golf by the repeated-hard rules.

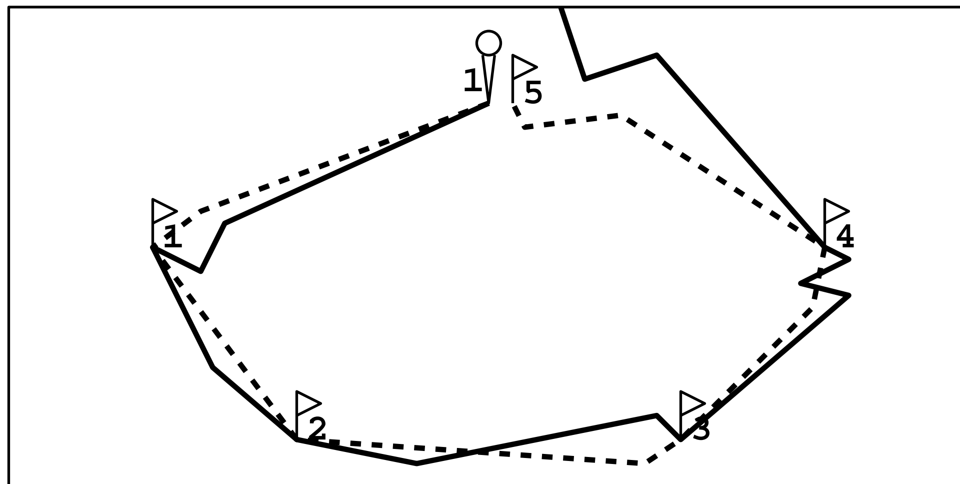

Figure 5.7: Playing golf by the repeated-soft rules.

a hole-in-one for every hole. Figure 5.7 illustrates this methodology. Since golfers see the same holes over and over, these methodologies are easier than the random methodologies (at least for computers). Since the players only pick up the ball at most once per sequence, they are also more realistic than the random methodologies.

There are other permutations of these methodologies that may be worth investigating. Since "random" and "repeating" are not mutually exclusive, one can imagine repeated short sequences of random goals and a single long sequence of task-oriented goals. A final example of a methodology that is both unchallenging and unrealistic is one where the agent goes back to the same initial state after each cycle until it achieves its goal, like "taking a mulligan" in golf.

There is one final non task-oriented evaluation metric. It is a comparison of the derived response surface with the "true" response surface; in effect measuring the error on all possible problems at once. It is popular in traditional adaptive control literature [2], probably because such systems develop easily analyzable response surfaces. Although the focus of this research is on task-oriented performance, it is possible to define a methodology to generate an accurate response surface. Instead of random sampling from the space

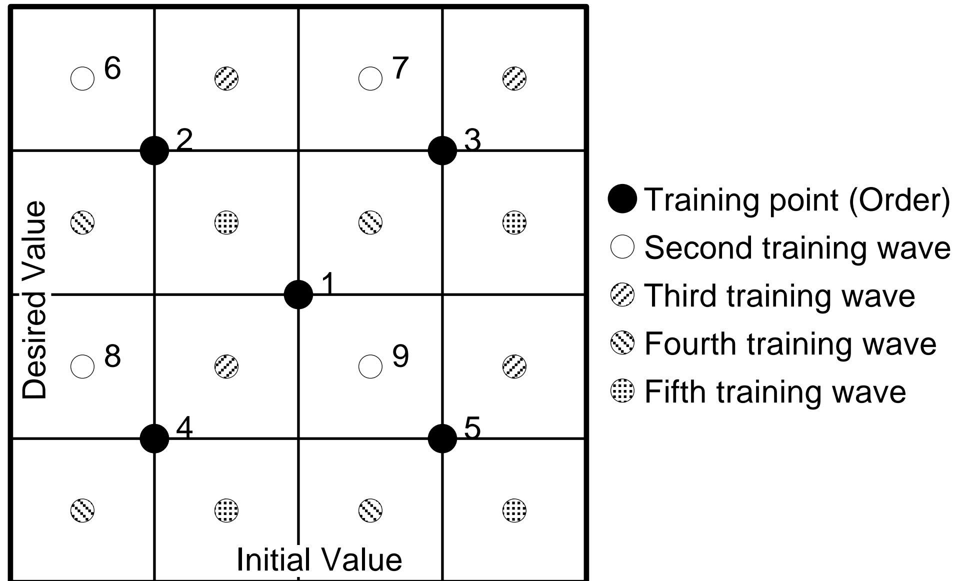

Figure 5.8: The response surface methodology.

of possible problems, the agent selects evenly-distributed problems and solves them. The problems come from dividing the $(Initial, Desired)$ space into quadrants, solving the problem in the middle of the quadrant, then subdividing each quadrant recursively until the response surface is accurate enough, as in Figure 5.8. This is a variant of the **random-soft** methodology. Since it is difficult to derive the analytical response surface from the cases, evaluation consists of simply testing the proposed control at a uniform sampling of problems distributed about the problem space, and comparing with the ideal control.

5.3 SPLICE Results

This section presents the results of the SPLICE experiments. The SPLICE agent was written in Soar, and the environment was simulated in Tcl [40]. The Soar agent included special output productions to change Tcl variables acting as controls, and the Tcl environment updated itself after every Soar primitive sequential control decision. Part of the updating included inserting the current values of sensed variables into Soar's working memory. To avoid inconsistencies between asynchronous environmental updates and real-time deadlines, the deadline was also expressed in terms of control decisions. A SPLICE agent followed each of the four testing methodologies in each of the three domains for a total of twelve learning profiles. The learning profiles provide specific examples of performance as the agent's expertise in the domain increases, insight into the development of the case database, and data on the learning rate.

5.3.1 Random Methodologies

An early observation was that the performance on a specific problem was highly dependent on preceding problem solving, so performance on the random methodologies did not uniformly improve. In an attempt to abstract away the effect of a particular random problem sequence, the final results were a median average of up to six random sequences. The first 100 problems were most variable, so they required more sequences, and the sec-

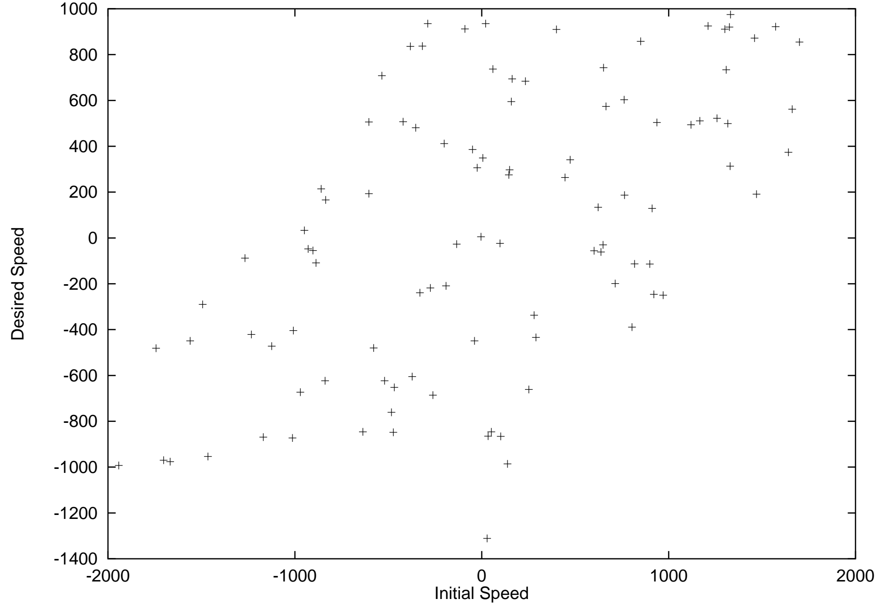

Figure 5.9: Initial and desired speeds for trial 1.

ond 100 problems were less variable, so they required less sequences. To accommodate this, three sequences were 100 problems, and three sequences were 200 problems. Both the **random-hard** and **random-soft** methodologies use the same six sequences, called trials. A sample sequence is illustrated in Figure 5.9. Since these methodologies do not guarantee convergence by the end of training, the section on accuracy on page 51 presents a response surface analysis before and after training.

Random-hard

The **random-hard** methodology, in which the agent only sees each random problem once, was expected to be the most difficult for SPLICE to learn, causing the slowest learning rate. It is also the most unrealistic, since the environmental state changes after every SPLICE cycle in a way inconsistent with the environmental model ("miracles"). However, the agent was able to master all three domains by the end of the training session.

As expected, initial performance was poor. Figure 5.10 shows the actual and desired trajectory for the first 20 problems for trial 1 in the linear domain. Performance improved as the agent gained experience in the domain, as illustrated by Figure 5.11, and was quite accurate by the end of the experiment, in Figure 5.12. The improvement in performance was due to the increasing level of detail in the case database. Figure 5.13 shows the case database after the first three problems. Figures 5.14 and 5.15 show the case database after the agent gains more experience. Each rectangle is a generalized case, and each dot is an experienced point. The more general the case, the lighter the shade of the rectangle. For clarity, the relationship between cases and points is not shown.

For example, the database in Figure 5.13 contains six cases. Case A is maximally general, but the other cases almost completely mask it. Case B is maximally general for the desired speed, but only applies to negative initial speeds. Case C is maximally general for the initial speed, but only applies to desired speeds between 0 and 1023. Case D is similar to case C except it only applies to speeds between 1024 and 2047. Case E only

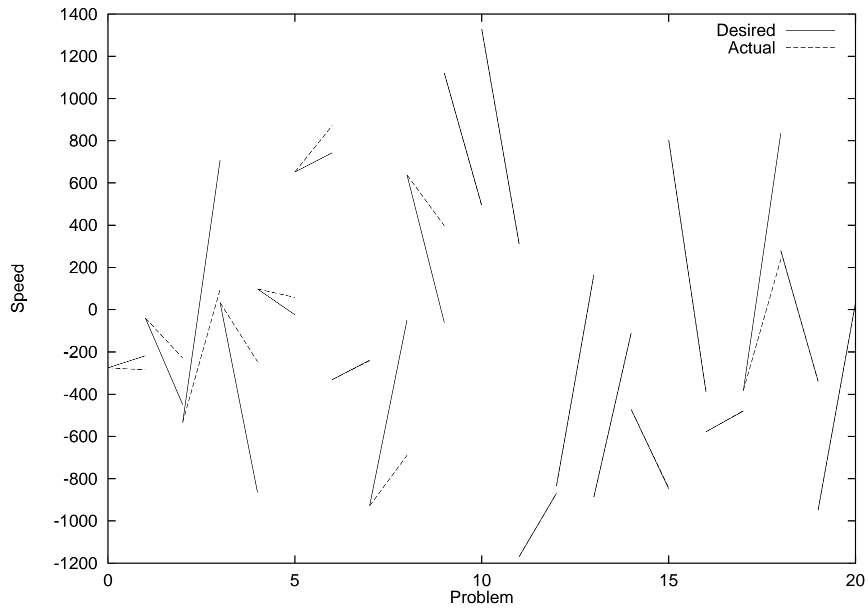

Figure 5.10: Initial performance for the linear domain, random-hard methodology.

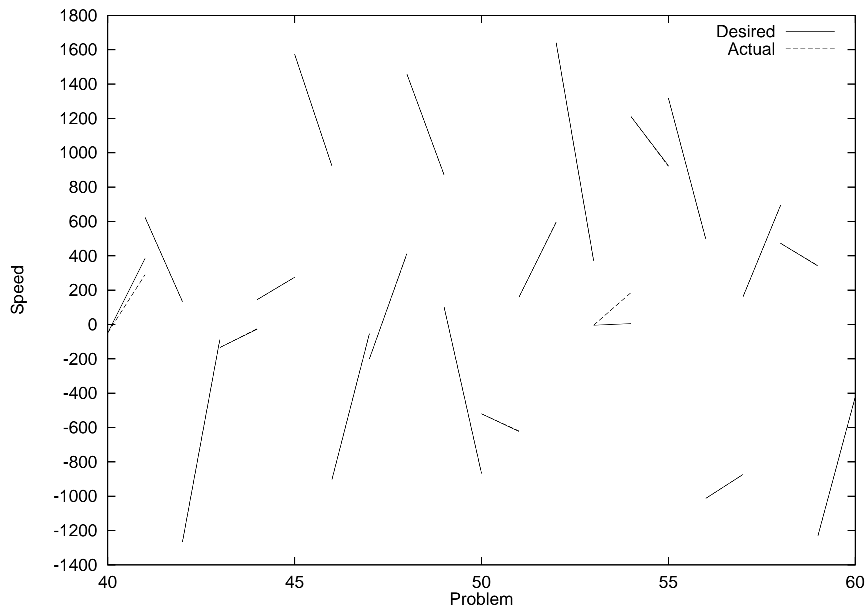

Figure 5.11: Intermediate performance for the linear domain, random-hard methodology.

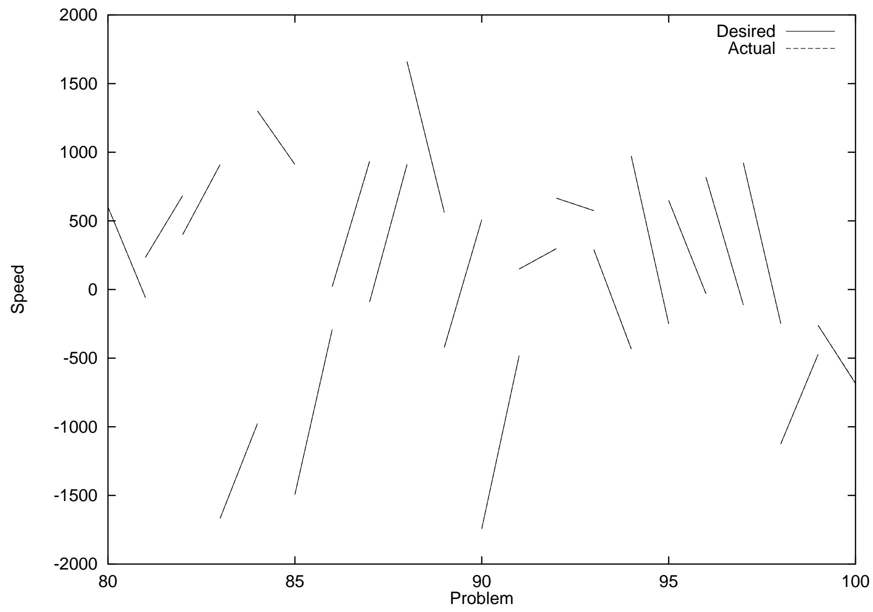

Figure 5.12: Final performance for the linear domain, random-hard methodology.

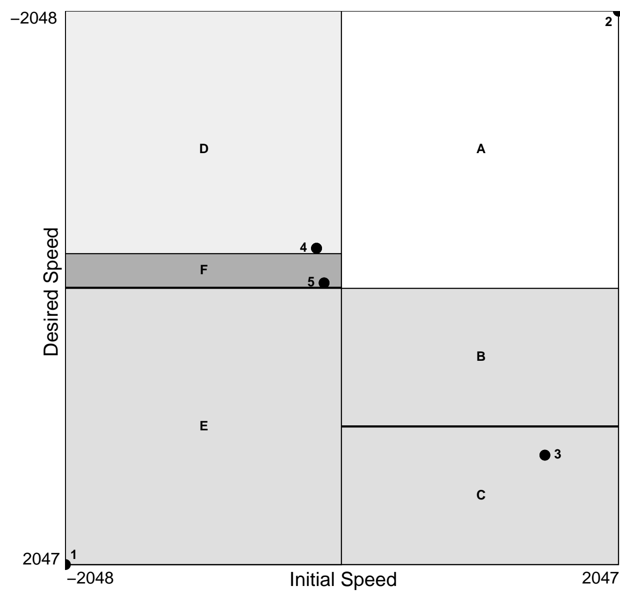

Figure 5.13: Early case database for the linear domain, random-hard methodology.

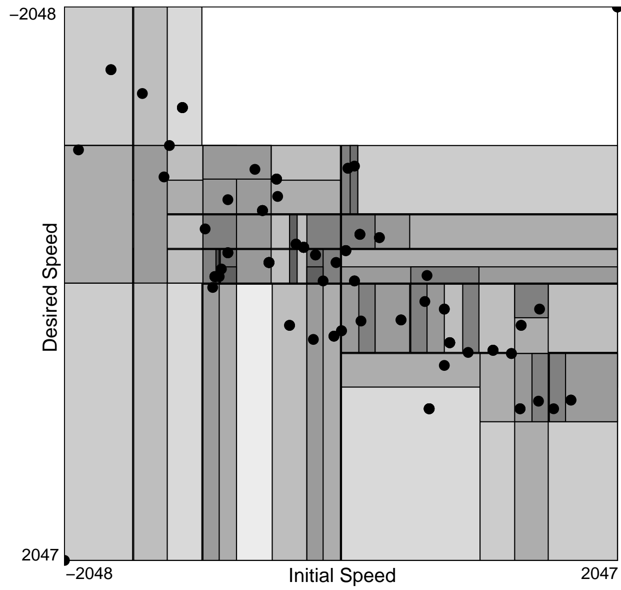

Figure 5.14: Intermediate case database for the linear domain, random-hard methodology.

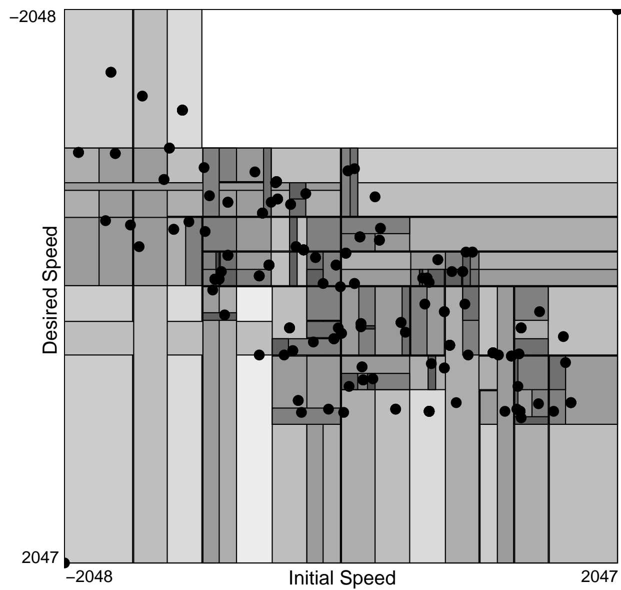

Figure 5.15: Final case database for the linear domain, random-hard methodology.

Case (parent)	Most Specific		P_1	P_2
	Initial	Desired		
A (domain model)	-2048–2047	-2048–2047	1	2
$B(A)$	0–1023	-2048–2047	2	3
$C(A)$	1024–2047	-2048–2047	2	3
$D(A)$	-2048–0	-2048–2047	2	4
$E(B)$	0–2047	-2048– -1	4	5
$F(B)$	-256– -1	-2048– -1	4	5

Table 5.1: Early case database.

applies to negative initial and positive desired speeds, and masks part of cases C, D, and B. Case F only matches negative initial speeds and desired speeds from -1 to -256. It also masks part of B. There are two domain model points and three experienced points. The relationship between cases and points is in Table 5.1. The agent learns the GDC B and the FSAC C after the first experienced point, FSAC D after the second experienced point, and GDC E and FSAC F after the third experienced point. Table 5.1 also shows how each case inherits one point from its parent according to the filters discussed in Chapter 4. More detailed information on how the implementation matches and learns cases is in Appendix A.

This methodology evaluates performance by measuring the difference between the desired value and the final value, so the learning rate is the change in error as the agent solves problems. Figure 5.16 graphs the learning curve. The x-axis is the problem number, and the y-axis is the relative error, calculated as a percentage of the maximum possible absolute difference. Since the interpolation is linear, SPLICE learns the linear domain much faster than the more complex domains. The approximately equivalent performance on the other domains demonstrates SPLICE's ability to perform in continuous and discontinuous environments equally well.

Random-soft

The **random-soft** methodology, in which the agent continues from its current situation until it achieves its current goal, then moves to a new random goal, was expected to be easier than **random-hard**. It is also more realistic, since miracles only occur when the SPLICE agent achieves its goal, instead of every cycle. Again, the agent was able to master all three domains by the end of the training session.

As expected, the agent took many cycles to achieve its goals early in the experiment. Figure 5.17 shows the actual and desired trajectory while achieving the first four goals for trial 1 in the linear domain. Performance improved continuously through the middle of the experiment (Figure 5.18) and the end (Figure 5.19). The case database structure was similar to that developed for the **random-hard** experiments.

This methodology evaluates performance by counting the number of cycles necessary to achieve the desired value, so the learning rate is the change in number of runs as the agent solves problems. Figure 5.20 graphs the learning curve. The x-axis is the problem number, and the y-axis is number of control adjustments made, with one control adjustment per cycle. Like the **random-hard** experiments, the SPLICE agent learned the linear domain faster than the more complex domains, but it occasionally requires exactly three control adjustments after perfectly performing the majority of the problems in a single control adjustment. This is an artifact of the combination of a linear domain, the requirement that

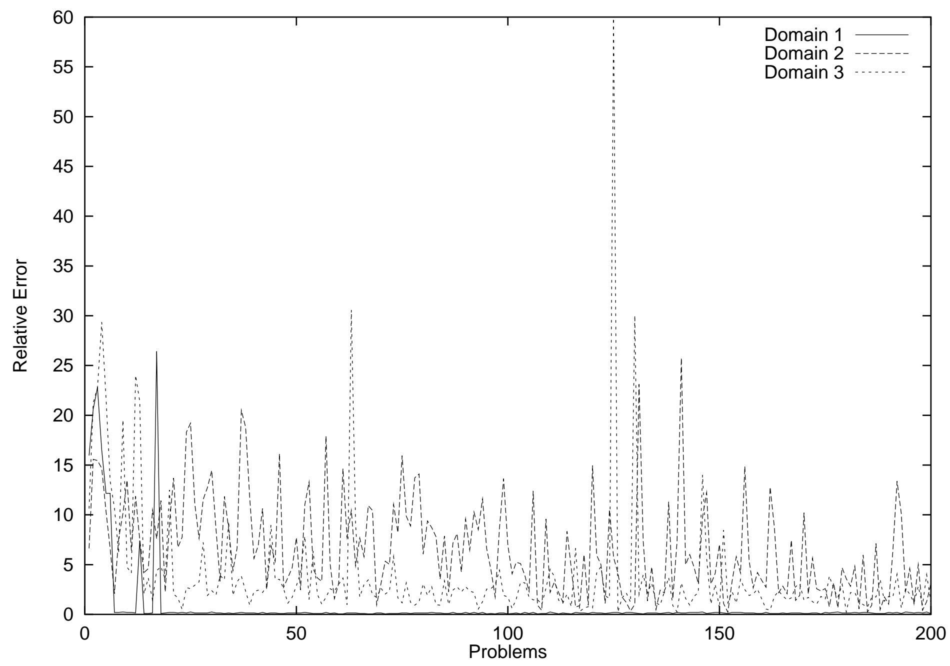

Figure 5.16: Learning curve for the random-hard methodology.

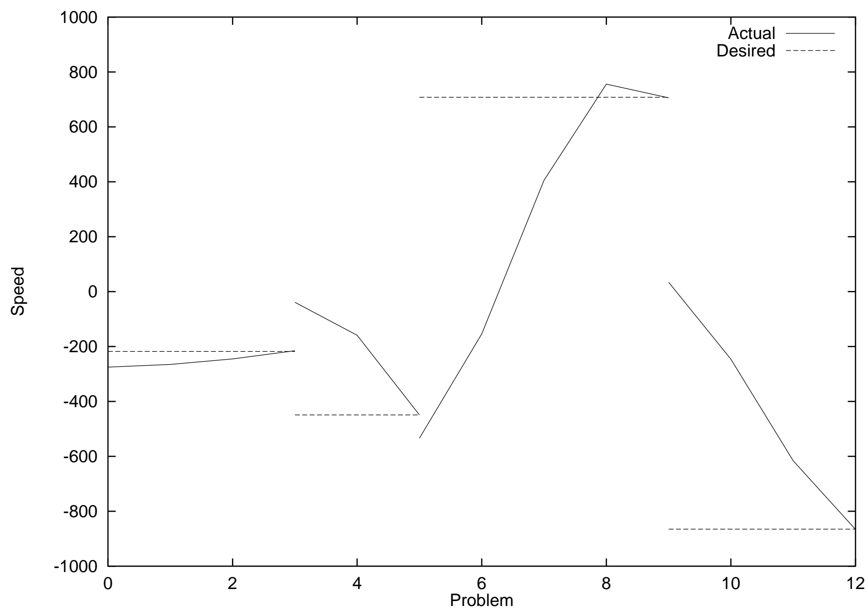

Figure 5.17: Initial performance for the linear domain, random-soft methodology.

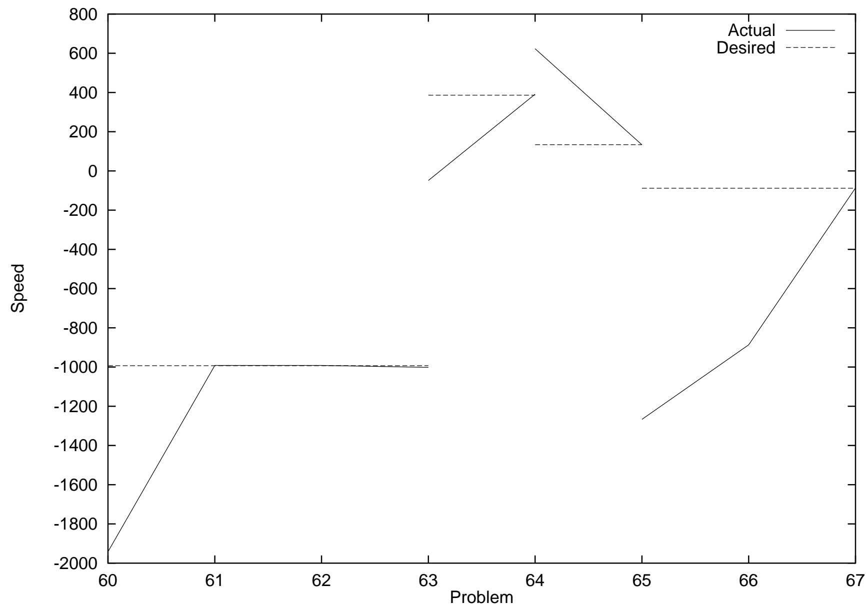

Figure 5.18: Intermediate performance for the linear domain, random-soft methodology.

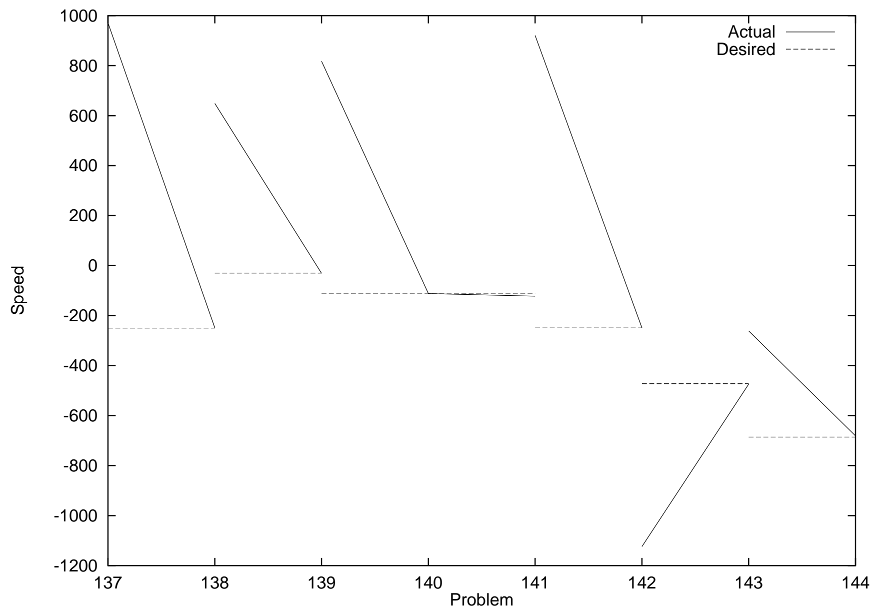

Figure 5.19: Final performance for the linear domain, random-soft methodology.

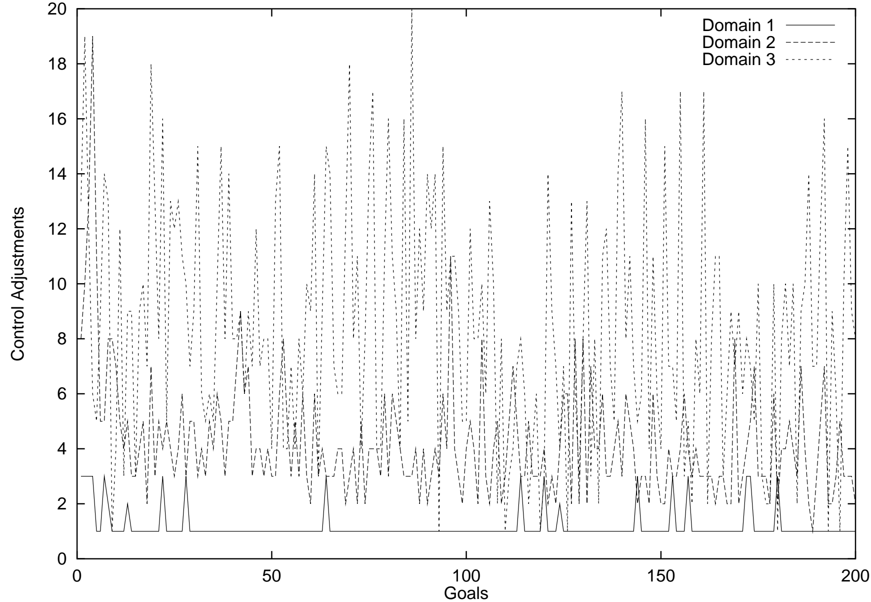

Figure 5.20: Learning curve for the random-soft methodology.

the control be an integer, the linear interpolation scheme, and the definition of achievement. For a problem which the agent had not solved previously, it performs a linear interpolation to calculate the correct control. Since the domain is truly linear, the interpolation is exactly correct, but it must be rounded. This rounding could cause the agent to miss the desired value, and even the most specific desired region. This normally happens when the desired value is near the edge of the desired region, e.g. the most specific desired region for a desired value of 0 is 0–15, so the agent would consider -1 a failure. For the next cycle, the agent must make a very small change in speed. This requires a small control adjustment in this domain, and rounding generally truncates this control to zero. This has no effect at all in this domain, so in the third cycle the agent observes that zero is too low or too high, and tries one or minus one, respectively, which finally achieves the goal. Although it is possible to fix this slight performance artifact, for example by explicitly reasoning about the rounding process, it only occurs in rigidly linear environments under special circumstances, so it is not likely to improve overall performance. Also, the agent records all failures and does not repeat them, so the agent will immediately solve the problem in similar situations.

Accuracy of Casebase

Since the stopping criterion for the random methodologies is simply reaching the end of the problem sequence, there is no guarantee that SPLICE has learned the environment to any degree of accuracy. In fact, it is not clear from Figures 5.16 and 5.20 whether SPLICE has learned the non-linear and discontinuous domains at all. The procedure to measure how well the agent has learned an environment is to evaluate performance on a new problem selected from the distribution of training problems. Since the distribution for the repeated methodologies uniformly covers the entire problem space, the evaluation reduces to the error in the response surface. Since the response surface's representation is distributed among all the cases, it is more practical to sample the response surface along a regular grid. The relative error at sample point is the absolute difference between predicted control and

	Non-linear domain	Discontinuous domain
Initial	22%	19%
Random-Hard	11%	6%
Random-Soft	10%	3%

Table 5.2: Relative error in response surface before and after training.

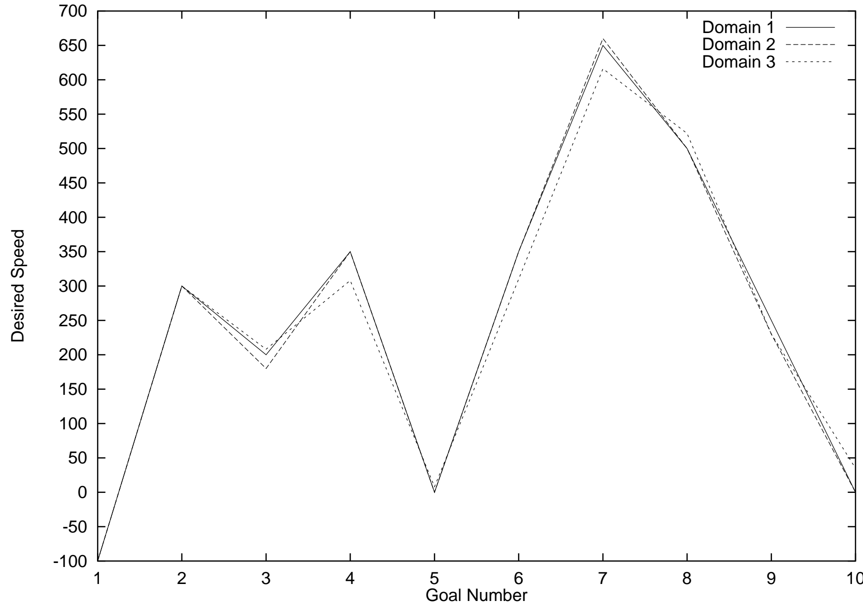

Figure 5.21: The sequence of speed goals for driving to work.

optimal control divided by the maximum possible error. The entire response surface error is the average relative error for a 16×16 grid of queries in the problem space. The set of query points is disjoint from the set of training points for a true measure of generalization accuracy. Table 5.2 shows the reduction in error from the response surfaces derived from the qualitative model and after solving 200 problems for each random methodology in the non-linear and discontinuous domains. The table shows that SPLICE greatly improves its accuracy from its initial knowledge, but requires additional training for perfect accuracy.

5.3.2 Repeated Methodologies

The second pair of methodologies repeat the same sequence over and over again until learning it to a specified tolerance. These methodologies are valuable because they are more realistic than the random methodologies, both because they require fewer "miracles" to change the state of the environment, and because the problem sequence is representative of a real task the agent may wish to achieve, such as driving to work. Since the problems are repeated, SPLICE has the opportunity to learn faster than the random "torture tests." The training set for the repeated experiments is in Figure 5.21. The slight difference for different domains is to ensure that it is possible to go from one goal to the next in a single control movement. Since the repeated methodologies focus on a specific sequence of goals, analysis of the entire response surface is inappropriate for these methodologies.

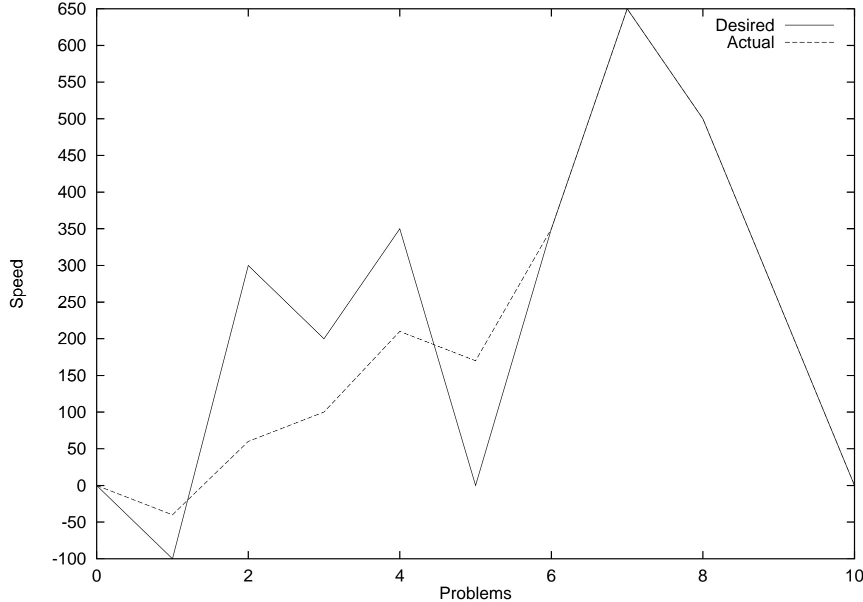

Figure 5.22: Initial performance for the linear domain, repeated-hard methodology.

Repeated-Hard

The hard version of the repeated methodology immediately goes to a new goal after a single attempt at the old goal. However, instead of choosing a new initial state, the agent continues from the final state of the previous attempt. Although performance in the first repetition is shaky (Figure 5.22), the agent quickly masters the task after three repetitions (Figure 5.23). The final case database (Figure 5.24) is much more uneven than in the random methodologies, because knowledge is denser near the task goals.

An interesting property of this methodology is that the agent tends to learn goals near the beginning of the sequence before goals near the end of the sequence. This is because the task of the agent is to learn a sequence of problems $((I, D_1), (D_1, D_2), \ldots, (G_{n-1}, G_n))$. The agent learns to achieve a desired speed D_j fastest when its training problems are from the same initial speed, D_{j-1} or I for this methodology. So, the easiest goal is D_1 because it always starts from the same initial speed, and learning the goal D_{j-1} stabilizes learning for D_j.

The learning rate graph in Figure 5.25 has the same axes as the graph for the **random-hard** methodology, but its performance improves much faster. The more complex domains are again much more difficult than the linear domain. The discontinuous domain requires a full twenty repetitions of the 10-goal sequence. The periodic spikes for these domains correspond to particularly difficult problems in the sequence. The difficulty of a problem in SPLICE depends both on the local nonlinearity in the response surface surrounding the problem, and the history of past learning near the problem.

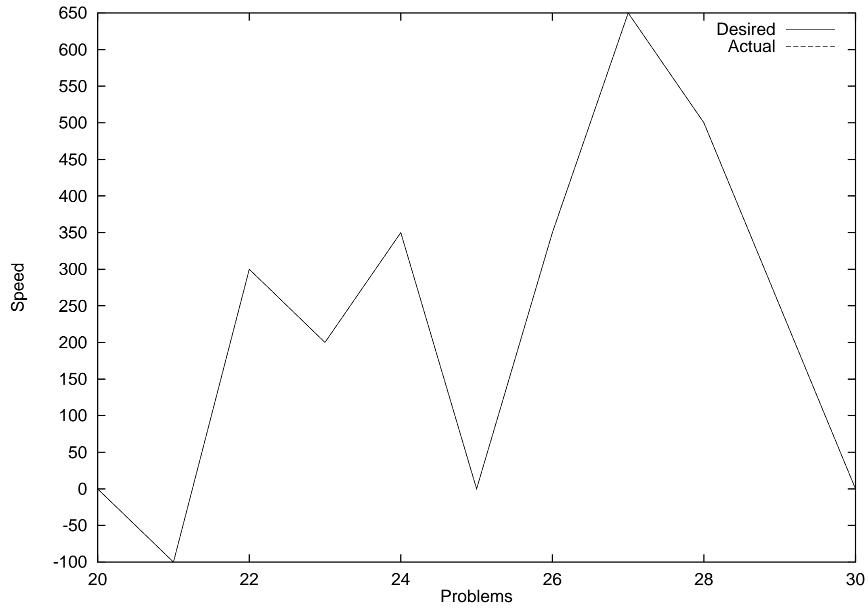

Figure 5.23: Final performance for the linear domain, repeated-hard methodology.

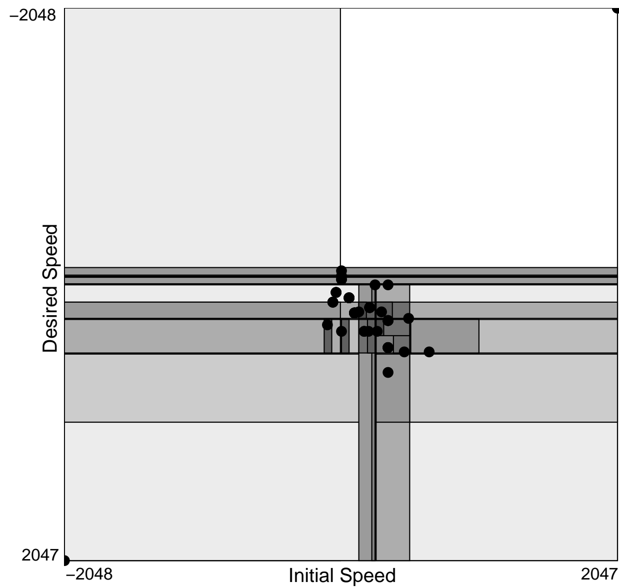

Figure 5.24: Final case database for the linear domain, repeated-hard methodology.

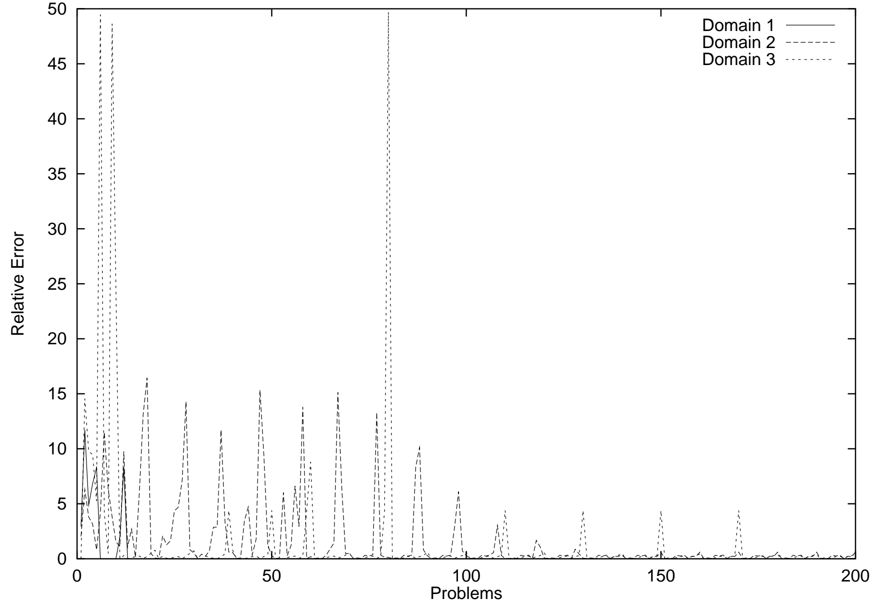

Figure 5.25: Learning curve for the repeated-hard methodology.

Repeated-Soft

The other repeated methodology continues to attempt to achieve goals until successful or the agent decides to give up. Although all problems were chosen to be achievable in a single control movement, it is possible for the agent to find itself in a situation where it cannot achieve the goal. This can occur because previous experience shows the desired value is not within the range of the control from this initial state, or the resolution of the control is too coarse to achieve the desired region, as described in Chapter 3. In this case, the agent selects the control to move as close to the goal as possible and moves on to the next goal for the next cycle. As expected, the agent learned the sequence in fewer iterations than in the **repeated-hard** experiments, because the agent attempted more, and learned more, in each sequence. Sample initial and final performance traces are in Figures 5.26 and 5.27. The case database development is similar to **repeated-hard** and is not pictured.

This experiment is evaluated by the number of control movements to achieve a goal. The results, graphed in Figure 5.28, are similar to **repeated-hard**, except the agent in the discontinuous domain always performs worse than the other domains. This is because the discontinuous nature of the domain makes it difficult to make small adjustments when near the goal.

5.4 Lesion Studies

SPLICE is a combination of a number of design choices, so it is difficult to use the results from the previous section to debate issues because it is not clear what effect one decision had on the results. A common way [32] to demonstrate the effect of some capability is to remove the capability and compare performance of this lesion again the intact system. In the case of knowledge-based systems, a lesion is the result of removing knowledge and possibly replacing it with some less-effective knowledge, just to allow the system to continue to function. This section identifies some relatively orthogonal capabilities of SPLICE and

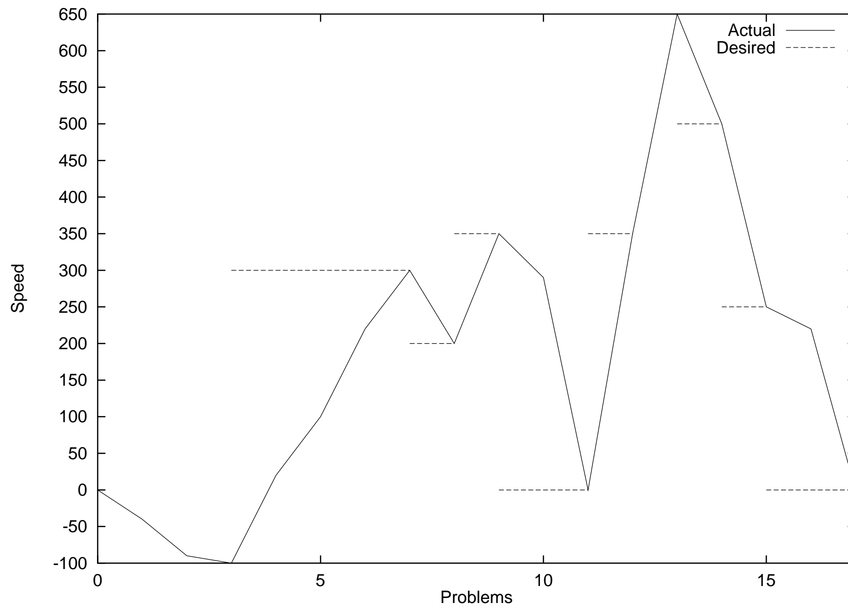

Figure 5.26: Initial performance for the linear domain, repeated-soft methodology.

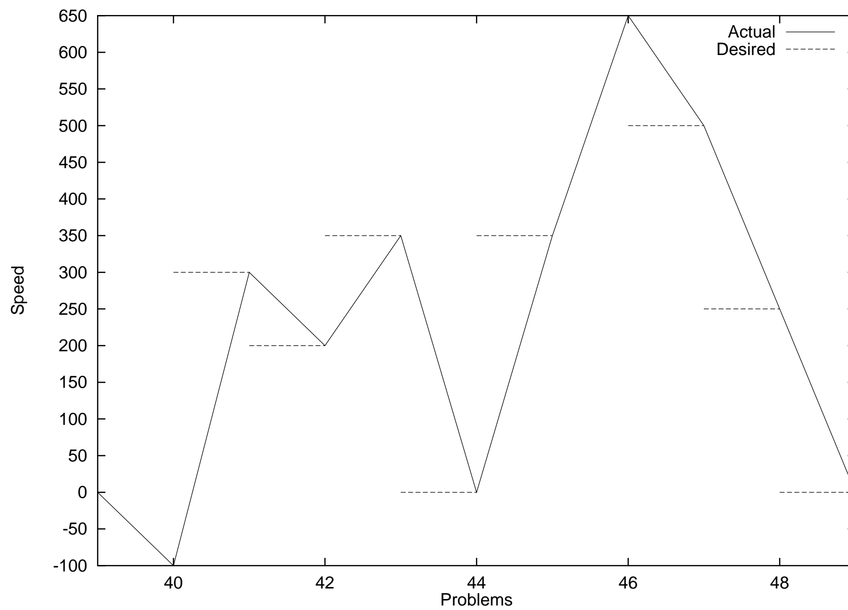

Figure 5.27: Final performance for the linear domain, repeated-soft methodology.

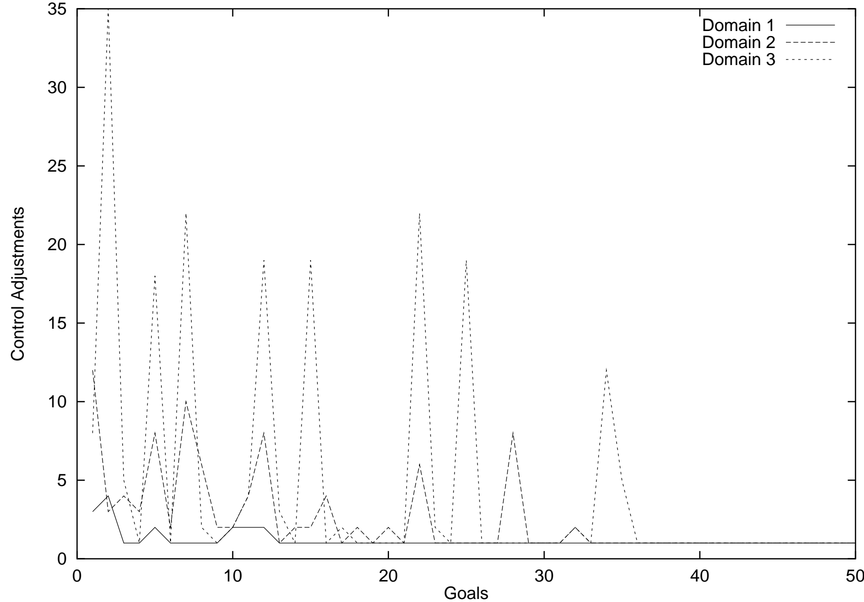

Figure 5.28: Learning curve for the repeated-soft methodology.

analyzes the effect of their removal.

5.4.1 Generalization Capability

One of the most important capabilities of any learning system is its ability to generalize experiences so that it can improve its response to future similar stimuli. The standard approach to generalization in symbolic machine learning has been to ignore irrelevant variables, either through deductive [11] or inductive [46] reasoning. SPLICE adapts this approach to continuous variables by inductively (through differential reasoning) selecting the most specific relevant resolution for the state and desired variables for a particular case, and ignoring details finer than this threshold. However, it is possible to imagine SPLICE without this capability, only able to learn about the current problem. This lesion explores the contribution of generalization to SPLICE's power by removing the differential reasoning knowledge. Instead, SPLICE always specializes to the most specific region. If there is already a case with the same specificity, SPLICE replaces the old case with the new one. Since the architecture is based on the assumption that each new case will only include one new region, the agent randomly picks one of the maximally specific state and goal regions. If we assume an equal probability for selecting each type of region, the expected number of problems in the same maximally specific initial and desired state necessary to create a maximally specific case is three.

The effect of this lesion was expected to depend most on the methodology used to test it. The random methodologies critically depend on generalization, because an agent may never again see the same problem, but must learn from it to improve its performance on similar problems. The repeated methodologies seem less dependent, because the goals are a relatively small repeating sequence, and the relevance of one goal to another is not clear. Two experiments in the linear domain tested these hypotheses, using the **random-hard** and **repeated-soft** methodologies. In the experiments, the agent first derived a completely generalized case from the domain model, then generated new cases by randomly

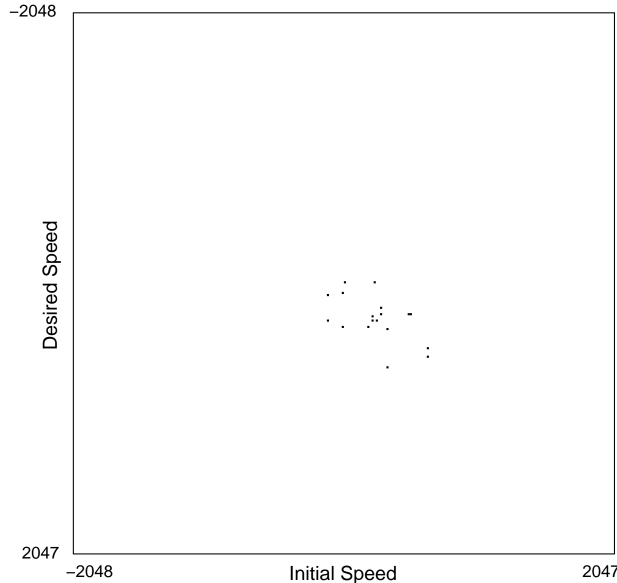

Figure 5.29: Final case database for the repeated-soft methodology, generalization lesion.

specializing the initial or desired speed. By the end of the experiment, the database consisted of many very specific cases. Each tiny dot in Figure 5.29 is a case, in contrast with the superior coverage in Figure 5.24, for example. For readability, only the cases with specialized initial and desired states are included. Figures 5.30 and 5.31 graph the evaluation results. Although the **random-hard** experiments were expectedly poor compared to SPLICE, the **repeated-soft** experiments were very poor as well. Analysis of the trace shows the reason to be poor transfer of knowledge between different initial states because of the extreme specialization. One possible extension to this lesion is to only specialize the goal, but this would lead to failure in domains which behave differently in different initial states, like the non-linear domain. The experiments show that SPLICE's generalization ability is critical to improve its performance.

5.4.2 Serendipity Capability

For an agent to be truly rational, it must take advantage of all its past experiences when solving problems. This includes remembering the problems it has "accidentally" solved in case they are useful in the future. This is the purpose of the "False Sense of Accomplishment" case described in Chapter 4. For simple algorithms such as Nearest Neighbor [18], this is implicit because all data points reside in the same memory, and new problems search the entire memory for the nearest points. For agents living in environments where long life is necessary, this algorithm becomes inefficient. SPLICE compromises by explicitly learning a recognition-based case, where storage and retrieval are not dependent on the number of experiences. However, there is a time and space cost to learn the FSAC, and this experiment was designed to determine if the FSAC improved performance enough to outweigh the cost.

Like generalization, the effect of removing the FSAC was also expected to depend on the methodology. The random methodologies are likely to use an FSA case, because the

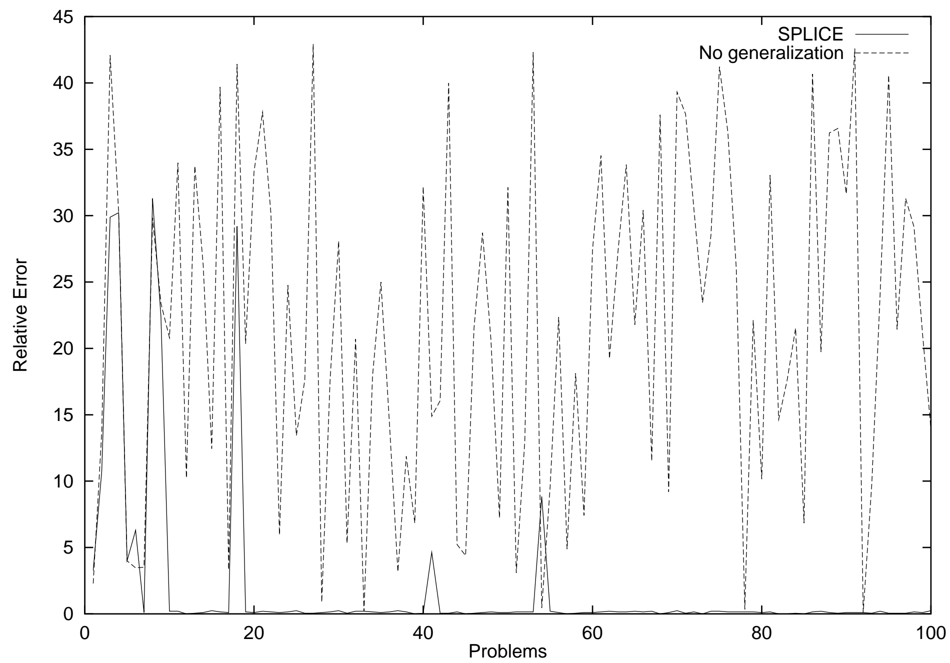

Figure 5.30: Learning curve for the random-hard methodology, generalization lesion.

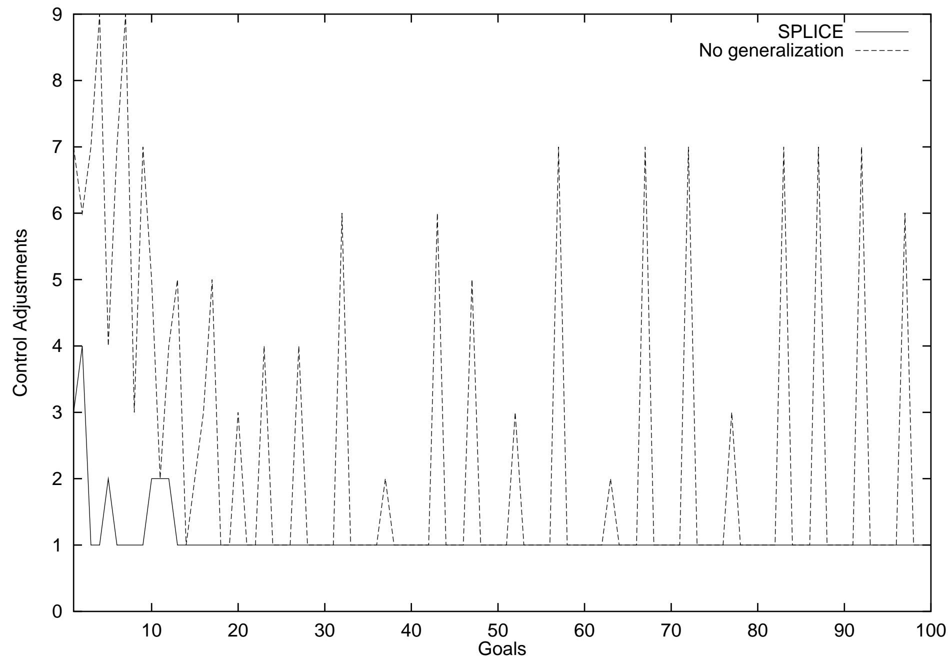

Figure 5.31: Learning curve for the repeated-soft methodology, generalization lesion.

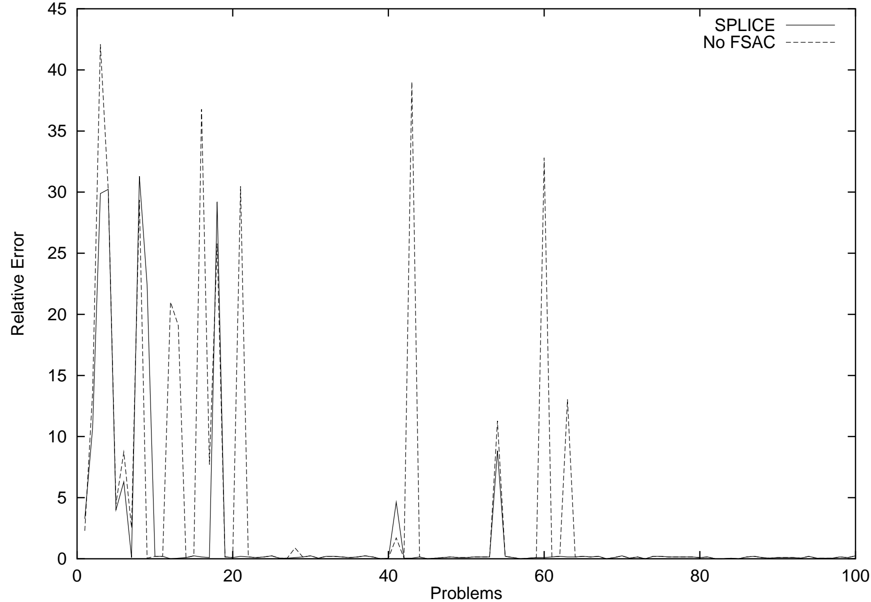

Figure 5.32: Learning curve for the random-hard methodology, FSAC lesion.

variety of goals and initial states implies that some initial and final pair (I, F) may be similar to some later problem (I, D). The repeated methodologies seem less likely to use an FSA case, because learning is faster, so there are less mistakes, and there are fewer distinct goals. Two experiments in the linear domain tested these hypotheses, using the **random-hard** and **repeated-soft** methodologies, like the generalization lesion. As expected, performance was slightly worse for the **random-hard** experiment without the FSAC, as graphed in Figure 5.32. Surprisingly, Figure 5.33 shows performance without the FSAC for the **repeated-soft** experiment was slightly improved. Studying the development of the case database shows that SPLICE attempts to specialize the FSAC as little as possible from its closest match, and this occasionally leads to specializing the initial state. Since the repeated methodologies have a small set of desired states but potentially any initial state, it is more important to learn general initial states than desired states. The GDC always specializes goals unless it finds an expectation failure, so learning only the GDC is slightly better for repeated methodologies. In this experiment, inappropriate specialization of the initial state led to matching cases with one experienced point and one domain model point, defining an inaccurate model. Since the GDC specializes the desired state until the agent achieves its goal, the model of a GDC matching a goal the agent has already solved is likely to be correct. This indicates that the inclusion of the FSAC depends on the intended use of the agent. Agents expecting to experience a variety of problems in their lifetime should improve their performance with the FSAC, while rigidly tasked agents may perform better with a strictly goal-directed configuration.

5.4.3 Point Selection Heuristics

The key knowledge component that allows SPLICE's time complexity to be independent of the number of experiences is the practice of deleting points considered unnecessary. As described in Chapter 4, SPLICE selects two points for use in interpolation for new cases. One point is the current experience, (I, U, F), and the other point is one of the two points

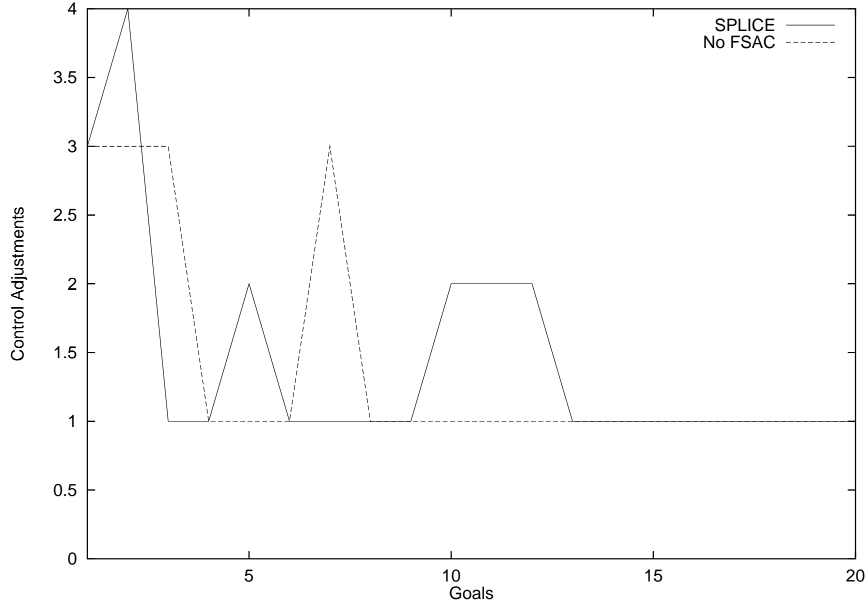

Figure 5.33: Learning curve for the repeated-soft methodology, FSAC lesion.

associated with the current matching case. In this way, every case has exactly two points, regardless of the total number of experiences. However, deciding which point to include in the new case is necessarily uncertain, because an on-line learning system cannot predict future problems to evaluate the effect of each alternative. Instead, SPLICE uses a simple heuristic to select a point: the point with the closest Euclidean distance to the current problem, Equation 4.1. An obvious parameter to introduce is a weighting factor α to bias the agent toward the initial or final states, or toward individual parameters. This gives the parameterized measure $R_i = \sqrt{\alpha \cdot (I_i - I)^2 + (1 - \alpha) \cdot (F_i - D)^2}$.

The two extremes, $\alpha = 0$ and $\alpha = 1$, are lesions because they effectively ignore information available to the agent. These are called the *select final* and *select initial* lesions, respectively. Since these lesions represent different assumptions about the nature of the environment, the experiments tested them on each of the three domains using the **sequence-soft** methodology. For the linear domain, all of the heuristics should work equally well, since the entire response surface has the same linear characteristics. In fact, for the linear domain, any experienced point is equally valid. The non-linear domain requires consideration of both the initial and desired states, so the lesions should perform worse than SPLICE. The discontinuous domain violates the assumption that closer points make better local models than farther points, so it is not clear what the hypothesis is. As expected, neither lesion affected performance for the linear domain, so it is not graphed. Figures 5.34 and 5.35 graph the results for the complex domains. The complex domains are dramatically in favor of regular SPLICE, since neither of the two lesions learn to perform the sequence perfectly in the allowed time. Although it is clear that most domains must make use of both initial and final state information, adjusting α to fit a particular domain may improve performance.

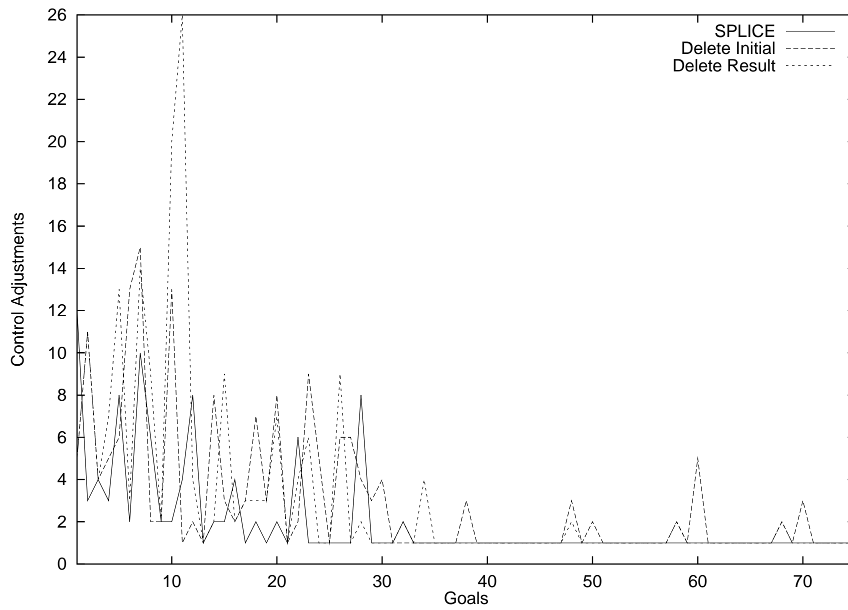

Figure 5.34: Learning curve for the non-linear domain point selection heuristics.

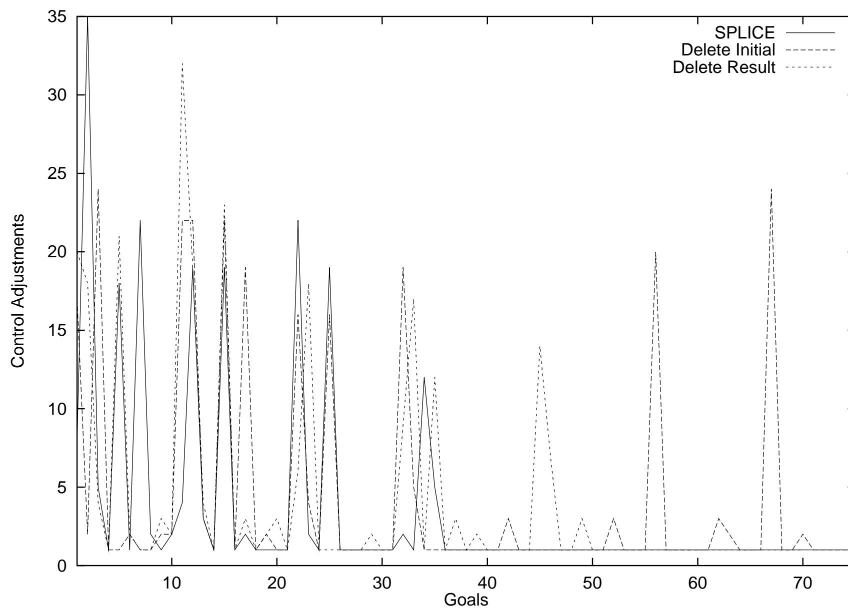

Figure 5.35: Learning curve for the discontinuous domain point selection heuristics.

5.4.4 Interpolation Capability

A final distinct SPLICE capability is linear interpolation, one of the few situations in SPLICE requiring floating point arithmetic. Although SPLICE's floating point math is minimal compared to most adaptive control algorithms, most cognitive theories of behavior do not hypothesize unlimited mathematical ability in high-level decision making. Instead, the interpolation can be replaced with a weaker mechanism for trying new controls to achieve a goal. One straightforward approach is the simple averaging of two numbers. When computing a control for a problem that has not yet been solved, SPLICE can choose two of four controls to average: the minimum control c_{min}, the maximum control c_{max}, the control c_1 from P_1, and the control c_2 from P_2. The choice depends on where the desired value d lies in comparison with the minimum value s_{min}, the maximum value s_{max}, the final value f_1 from the first point, and the final value $f_2 \geq f_1$ from the second point. From these values, we get

$$c = \begin{cases} \frac{c_{min}+c_1}{2} & \text{if } s_{min} \leq d < f_1 \\ \frac{c_1+c_2}{2} & \text{if } f_1 \leq d < f_2 \\ \frac{c_2+c_{max}}{2} & \text{if } f_2 \leq d \leq s_{max} \end{cases} .$$

This equation produces a type of binary search for the correct control. For example, for an initial speed of 0 and a desired speed of 500 in the linear domain, the agent would try the commands 64, 32, 48, 56, 52, and 50 in order each time it encounters this problem, as opposed to the sequence 15, 27, 50 with interpolation. Experiments with the **repeated-soft** methodology showed that the agent had great trouble achieving its goals because poor control decisions put the agent in sparsely-experienced states, making it difficult to return to the same state enough to learn to solve the problem. Experiments with the **repeated-hard** methodology were more successful, because the agent automatically returns to the same initial state after ten cycles. In all three domains, the agent never learned to solve a problem before learning to solve all problems earlier in the sequence, which slowed it down considerably compared to regular SPLICE. Figures 5.36, 5.37, and 5.38 graph the results. The agent was able to learn to achieve the entire sequence in each of the domains, but approximately ten times slower than regular SPLICE. This is a somewhat discouraging result because it implies that a cognitively plausible version of SPLICE using averaging would learn much slower than regular SPLICE.

5.5 Alternative Design Studies

The preceding modifications to SPLICE all dealt with removing access to knowledge in order to study the contribution of this knowledge. This section concerns modifications to SPLICE that access the same knowledge in a different way.

5.5.1 Abstraction Order

SPLICE abstracts features along two dimensions until it finds a match: the initial and desired states. This implies that SPLICE must define an order along these dimensions. The procedure described in Chapter 3 first abstracts specific state features until it finds a match in the state dimension, then abstracts goal features. The intuitive explanation is that the agent first wants to find the closest matching initial state, then find the most specific goal it can achieve from this state. However, there are other approaches. The *goal first* procedure abstracts the goal first, then the state, and the *aggregate* procedure abstracts the goal and

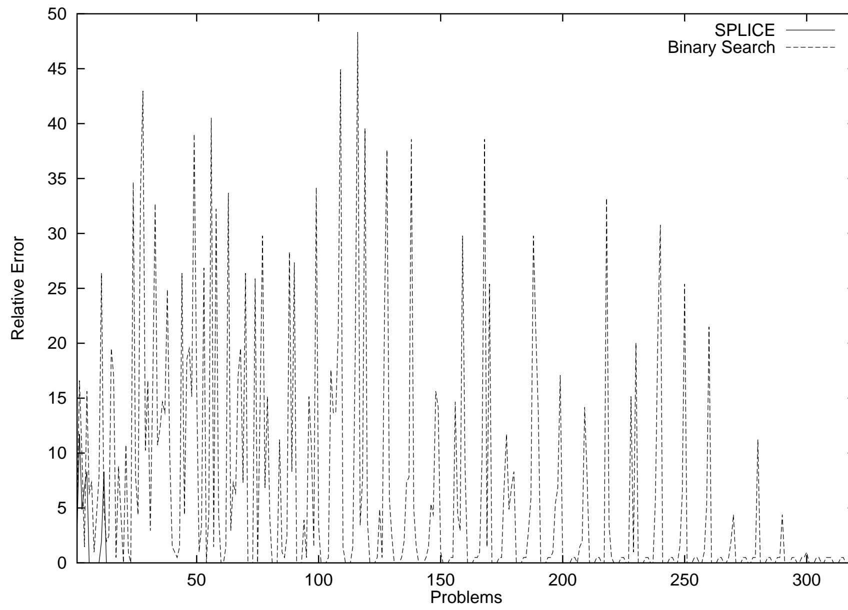

Figure 5.36: Learning curve for the linear domain interpolation lesion.

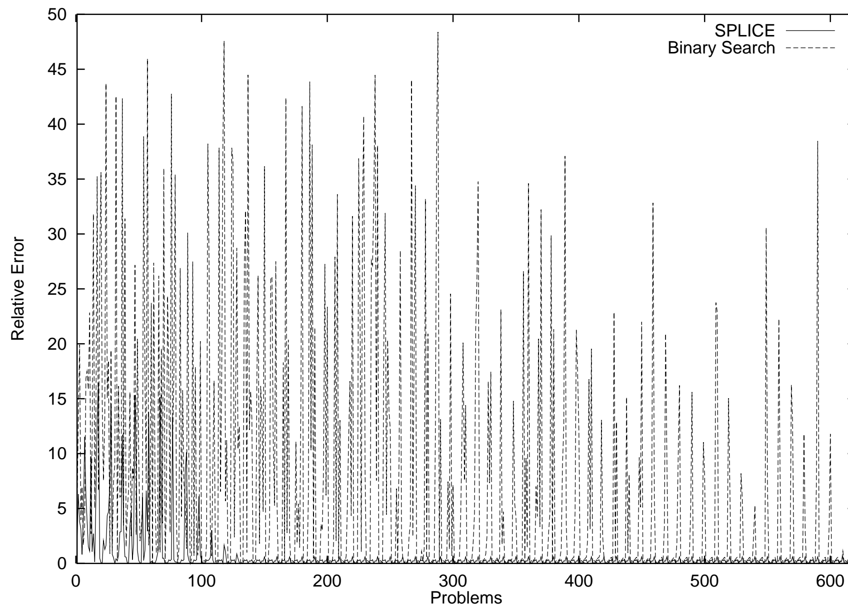

Figure 5.37: Learning curve for the non-linear domain interpolation lesion.

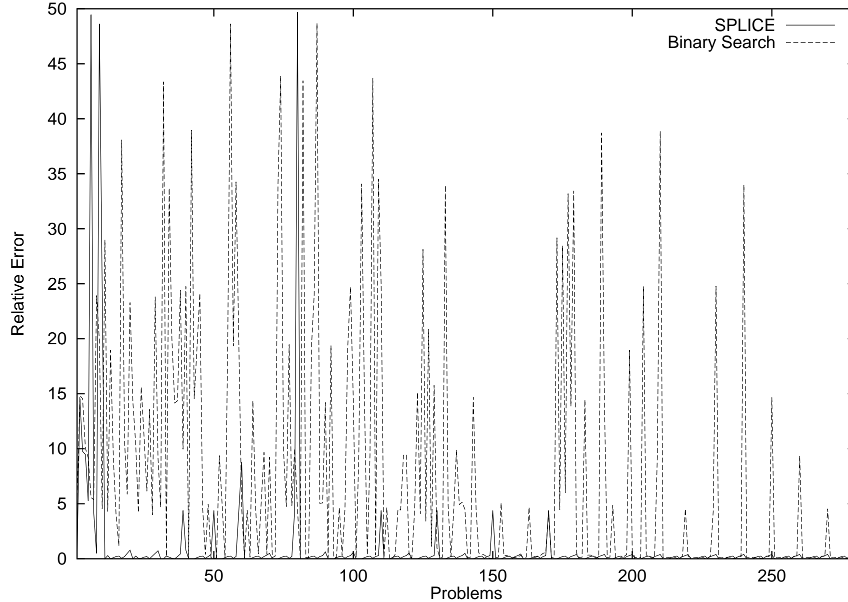

Figure 5.38: Learning curve for the discontinuous domain interpolation lesion.

the state together to find the overall closest match.

Since there is no apparent theoretical distinction between these three abstraction procedures, they were empirically evaluated in the three domains using the **repeated-soft** methodology. The learning curve for the linear domain in Figure 5.39 shows that the goal first and aggregate procedures slightly outperform state first, immediately achieving the second goal of the second repetition, while SPLICE requires two attempts. This is caused by this lesion partially masking the negative effect of including the FSAC for the **repeated-soft** methodology, as explained in Section 5.4.2. Since the lesions do not abstract the state first, some GDC's may match before FSAC's if the desired value is specific enough, even if the initial value of a FSAC is more specific than a GDC. However, like previous modifications, both alternatives perform poorly in the more challenging domains. Figure 5.40 shows that the goal first procedure never completely solves the entire sequence in the non-linear domain, and even the aggregate procedure is slower than regular SPLICE. Finally, the discontinuous domain in Figure 5.41 gives the aggregate procedure a very difficult time, and even the goal first procedure is much slower than regular SPLICE. The repeated methodologies probably perform particularly well with the state first heuristic because there is only a small set of distinct desired values, so it is more important to match the initial state well. Initial experiments with the random methodologies indicate that performance is relatively equivalent amoung the three heuristics.

5.5.2 Feature Selection

A final design alternative concerns improving SPLICE's generalization ability. The best way to illustrate the issue is with an example. Suppose a SPLICE agent matches a case where the most specific initial speed is 256–511 (level 4) and the most specific desired speed is 0–2047 (level 1). When the agent learns an FSAC, it finds the most general difference (MGD) between the current initial and final states and the initial and desired states from

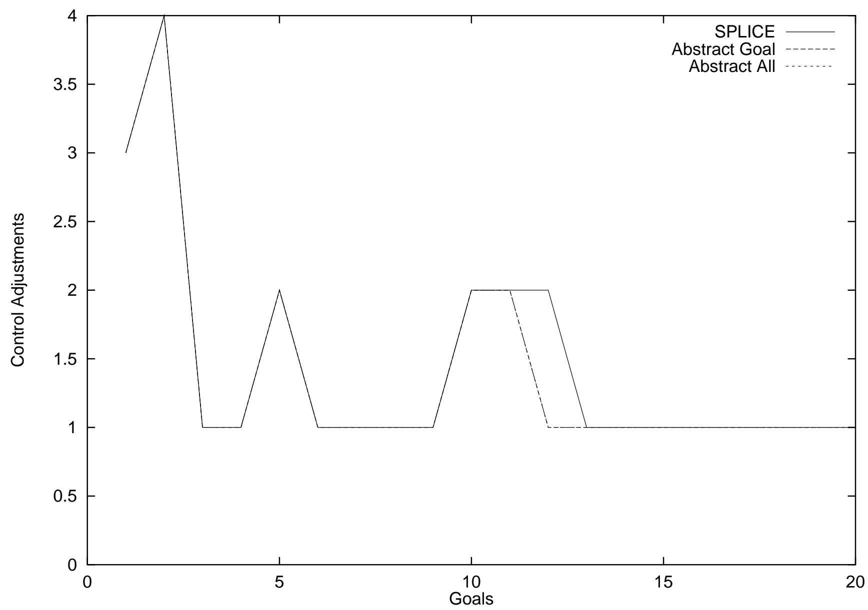

Figure 5.39: Learning curve for the linear domain abstraction alternatives.

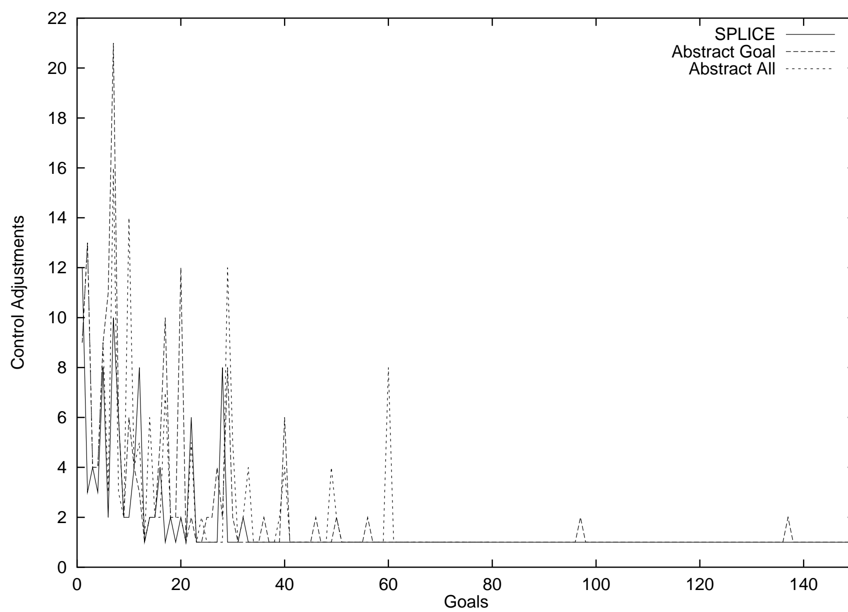

Figure 5.40: Learning curve for the non-linear domain abstraction alternatives.

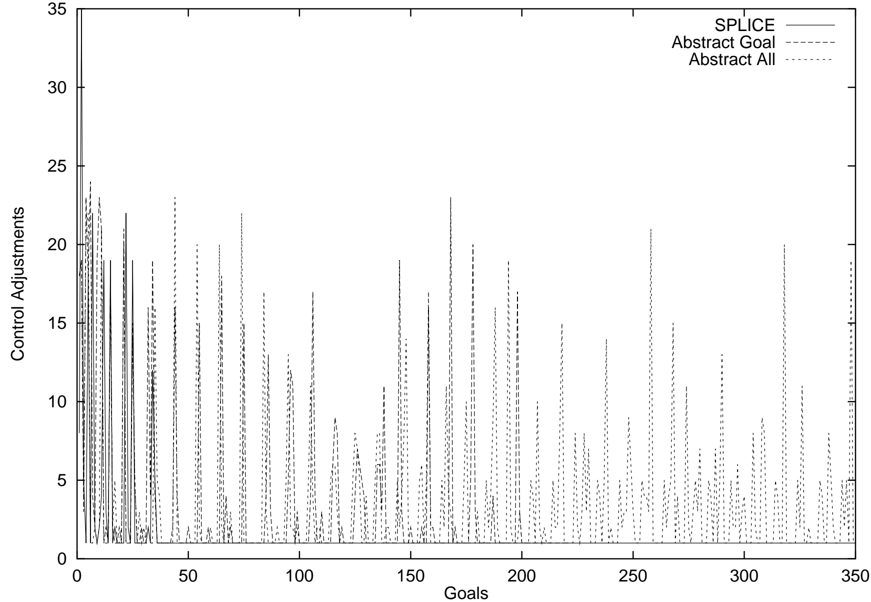

Figure 5.41: Learning curve for the discontinuous domain abstraction alternatives.

the case[1], and specializes the match with the more general of the MGD's. Suppose in our example, which comes from a real problem-solving episode, the MGD for the initial speed is 256–383 (level 5), and the desired MGD is 0–255 (level 4). Since the desired MGD is more general, regular SPLICE specializes the desired, giving a new case covering $256 \cdot 256 = 2^{16}$ possible problems. However, if SPLICE specialized the initial speed, the new case would cover $128 \cdot 2048 = 2^{19}$ possible problems, which is more general by a factor of eight. The key observation is that the criteria for selection is not the absolute MGD, but the change in MGD from the match to the new MGD, called the ΔMGD.

The advantage to the ΔMGD selection heuristic is that it improves overall generality, but the advantage of the absolute MGD selection heuristic is that it tends to create cases with more evenly-specialized initial and desired states. Since different domains may favor one heuristic over the other, experiments were performed in all three test domains using the **repeated-soft** methodology. The results from the linear domain in Figure 5.42 show that ΔMGD performs worse than absolute MGD for most goals. ΔMGD performs even poorer in the more complex domains, as illustrated in Figures 5.43 and 5.44. Like the abstraction alternatives, it may be possible to define a domain where the ΔMGD heuristic is superior to the regular absolute MGD heuristic, but that domain is not likely to be similar to the test domains in this thesis. It is possible that the agent performed better in the test domains because the response surface is more amenable to square tessellation from MGD than rectangular tessellation from ΔMGD, so ΔMGD may perform better for response surfaces that can be approximated by long rectangular strips.

This chapter has introduced three artificial test domains and four testing methodologies. It presents the behavior of SPLICE in these domains and methodologies, and draws insight into the strengths and weaknesses of SPLICE through study of the complete system and a

[1]We compare final and desired states since in the FSAC the agent "pretends" that the final state *is* the desired state.

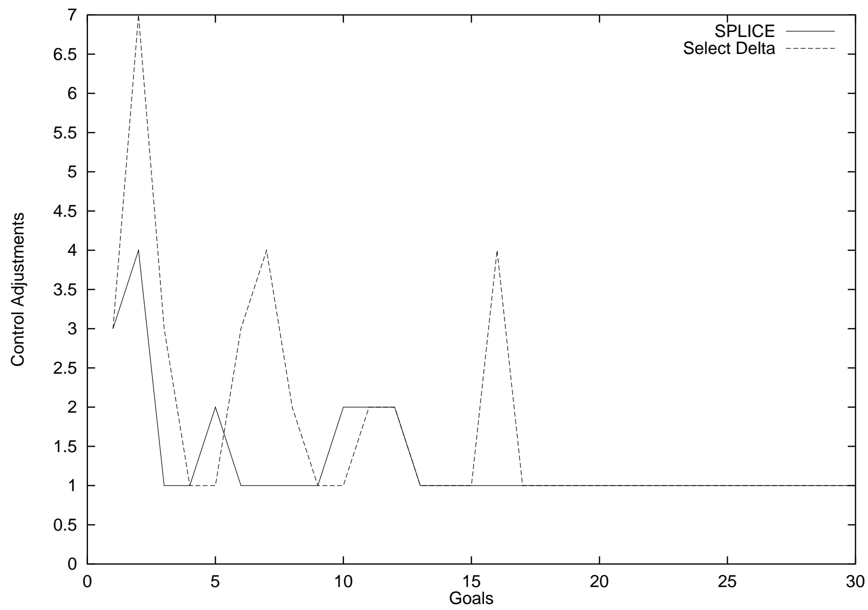

Figure 5.42: Learning curve for the linear domain feature selection alternatives.

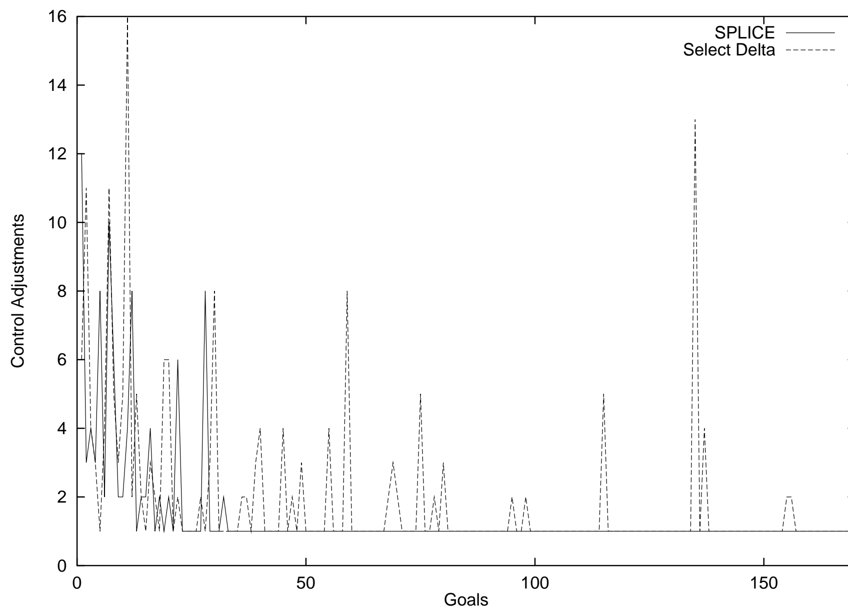

Figure 5.43: Learning curve for the non-linear domain feature selection alternatives.

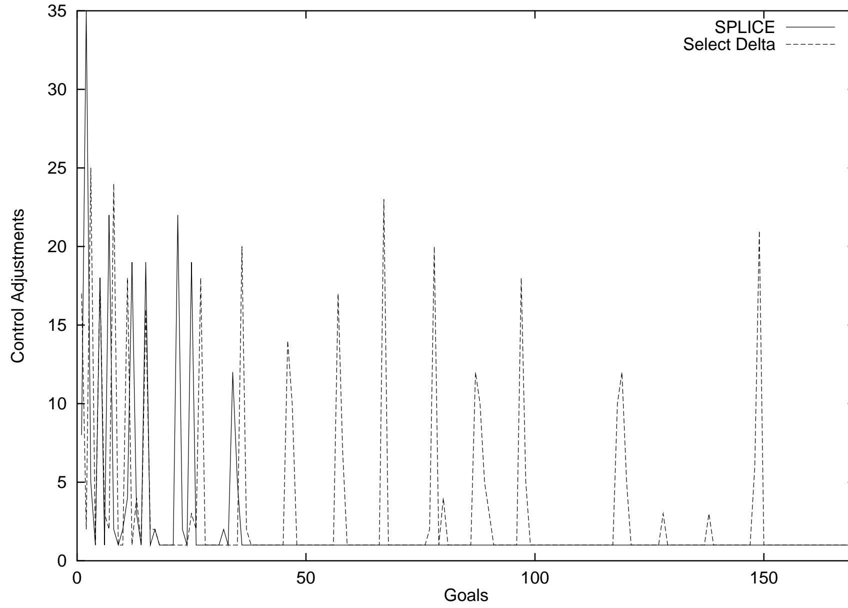

Figure 5.44: Learning curve for the discontinuous domain feature selection alternatives.

number of lesions and design alternatives. Further details on the functioning of the system are in Appendix A. The following chapters pit SPLICE against other algorithms with similar functionality and a more demanding environment.

CHAPTER 6

Performance Comparison with Other Algorithms

. . . we learn big things a little at a time;
To go right up a mountain, first we learn to climb!

This chapter introduces three algorithms also capable of performing in the artificial domains described in Chapter 5 using the methodologies from that chapter. The results of testing on the same domains and with the same methodologies indicate that a symbolic processing approach to adaptive control does not inherently perform poorer than traditional numerical algorithms. This chapter only compares the learning curves for each of the algorithms, since the previous chapter covered the behavior of SPLICE in depth, and the references provide more information on the other algorithms.

6.1 Algorithms for Comparison

SPLICE is a novel algorithm in many ways, primarily because of the integration of a qualitative reasoner with an empirical learning element, and because of its fast, rule-based, incremental learning strategy. SPLICE can function in extremely qualitatively complex domains with the addition of domain-specific qualitative relationships, and it can serve as a component in a larger symbolic system. However, SPLICE can also perform in environments that do not require a complicated model or sophisticated symbolic reasoning. There are a number of adaptive control algorithms specifically designed to perform in these "knowledge-lean" environments, and it is possible to compare SPLICE against them to evaluate SPLICE's base-level performance and empirical learning capabilities.

Finding appropriate comparison algorithms for SPLICE first requires identifying the general class of algorithms similar to SPLICE, then choosing well-known and powerful exemplars. The primary characteristics of SPLICE that distinguish it from other learning algorithms is its ability to handle continuous input and output, and its incremental nature. Certain learning techniques with a history of symbolic representations such as Inductive Logic Programming and most forms of reinforcement learning are immediately excluded because they have no way to generalize a continuous value to similar values. Batch-style algorithms such as decision trees are too inefficient to perform on-line learning. Even incremental decision trees like ID5 [57] perform worse as the training set grows, because the increasing tree depth slows classification. Similarly, raw nearest-neighbor algorithms slow down as their memory grows, and so are not incremental. Finally, it is desirable to select established and well-understood representatives of different approaches to control. Some popular and powerful control algorithms that satisfy these requirements are Proportional-

Integral-Derivative (PID) Control, spline fitting, and Receptive Field Weighted Regression (RFWR), as introduced in Chapter 2. Many other control algorithms are subsumed by these three. For example, (Schaal and Atkeson 1996) shows that RFWR easily outperforms neural networks.

Each of these algorithms approach the problem somewhat differently. External differences are characterized along two major dimensions: restrictions on environmental properties, and initial knowledge requirements. As discussed in Chapter 5, SPLICE works best when the environment is monotonic and continuous, but these are not strict requirements. SPLICE's initial knowledge requirements are information about the sensory variables and effectors, and a qualitative model. The characteristics of the competing algorithms are discussed below.

6.1.1 PID Control

PID control is a very simple non-adaptive algorithm for control [13]. PID is representative of the traditional control theory paradigm. The PID response function $PID(p, i, d)$ is a linear function of the difference between the current value and the desired value p, the integral of the differences over time $i = \int_t p\,dt$, and the rate of change of the difference $d = dp/dt$, so

$$PID(p, i, d) = \alpha \cdot p + \beta \cdot dp/dt + \gamma \cdot \int_t p\,dt.$$

Although PID control is simple, it requires fairly specific initial domain knowledge to set the weights. In the experiments, the constant weights (α, β, γ) were manually tuned to optimize performance over the set of presented problems in each environment. Although PID is linear like SPLICE's linear interpolation, PID uses the same weights for all problems, so it is not expected to perform as well as SPLICE for non-linear environments. In fact, since the PID control strategy is unable to adapt to its environment, it is a strawman measuring minimal performance competency.

6.1.2 Spline Fitting

Spline curves and surfaces are smooth functions with an extremely flexible representation [9]. These are representative of the parametric function approximation approaches. The spline fit algorithm fits a spline surface to the data to explicitly represent the response surface. Spline surfaces are good at representing curved surfaces because their definition is both local and continuous. Splines are local because they are composed of a number of polynomial functions, and changing the definition of one function does not affect the global surface, only a constant-sized region around the center of the changed function. Splines are continuous because the individual functions are continuous, and the merging function between polynomials maintains continuity. By defining a spline surface from a set of data points, the resulting surface will pass through the data points and smoothly interpolate between them.

`Bivar` [2] is a popular algorithm for fitting a spline surface to data. This algorithm is part of the NCAR mathematical algorithms library. The surface fit to the set of points (I, F, U) consists of three major steps:

Triangulation Step 1 divides the problem space (I, F) by finding triangles that have maximal minimum angles.

Partial Derivatives Step 2 estimates the first partial derivatives U_I, U_F at each data point, and from these the second partial derivatives U_{II}, U_{FF}, U_{IF}.

Fit Step 3 fits a fifth-degree polynomial to each of the triangles using the partial derivatives.

Since the spline surface must be completely recomputed after each new data point, it is technically not an incremental method. However, the algorithm did not appear to slow down appreciably in the experiments, and it was desirable to have a traditional parametric approximator. Moreover, it is possible to imagine an incremental version that refines the triangulation to some threshold, then only adjusts one local polynomial for each data point.

This spline fitting method is extremely liberal about initial domain knowledge. Since the algorithm automatically computes the number and shape of local functions, the only necessary preparation for a domain is some prior points. These are similar to the initial points SPLICE derives from its qualitative domain model. The difference is that these points are permanently a part of the data fit to the spline, so if the points are wrong the surface is distorted in those areas. For each case, SPLICE does not retain the domain model points when it gets two experienced points. The nature of spline surfaces imposes specific environmental restrictions. Since the spline is constrained to be continuous, the environment must be continuous as well. Overall splines, like many parametric functions, are good and efficient at fitting data that matches their assumptions, but can fail dramatically if their assumptions are violated.

6.1.3 Receptive Field Weighted Regression

Receptive Field Weighted Regression is a new local function approximation technique [55]. It derives from research on non-parametric modeling methods. The basic data structure of RFWR is the receptive field. A model consists of a set of receptive fields, each of which has an extent and a linear partial model. RFWR makes predictions with the following algorithm:

1. Compute the membership of a query point, in this application the problem (I, D), for each receptive field.

2. Compute the prediction for the query with the linear model for each receptive field.

3. The response is the sum of the predictions weighted by the membership values.

This sequence is very similar to a "fuzzy" version of SPLICE, where the receptive fields function as cases. The extent of the fields is proportional to the generality of the case, and the linear model is analogous to the points associated with the case. Learning a RFWR model is also similar to learning in SPLICE. When the RFWR algorithm receives a data point, it classifies it to the closest receptive field and observes the error of the linear model. If the error is below a noise threshold, RFWR has already accounted for the data and discards it. If the error is above the threshold, RFWR adjusts the linear model to account for the data. If it cannot adjust the model so that it remains accurate for previous points, it splits the current data point and possibly other data points into a new receptive field. RFWR also places weak constraints on the nature of its environment. Since the models are local, the environmental restrictions are similar to SPLICE's: the data structures can handle discontinuous surfaces, but the algorithm is not guaranteed to find solutions in discontinuous environments. RFWR does not have any monotonicity assumptions.

Although there are many similarities between RFWR and SPLICE, there are also many differences. The most notable difference is RFWR's detection of degree of match for each receptive field, as opposed to SPLICE's selection of a single best match. Although fuzzy matching may improve performance for borderline queries, there is an inevitable performance penalty. If we let n be the number of receptive fields/cases, a sequential processor requires $O(n)$ time to find the membership value of each receptive field. Since the SPLICE case database is a set of rules, Soar matches the cases in parallel. This allows the time complexity to be $O(1)$, as described in Chapter 4. Schaal avoids this problem by setting an upper limit on the number of receptive fields n.

Another difference is RFWR's use of linear regression (multiple points). Regression is more statistically valid than interpolation because it is more noise-tolerant and well-conditioned (less likely to compute horizontal or vertical lines). With few points, regression is similar to interpolation, but as the regression algorithm processes more points, it becomes more accurate and stable. Since regression is more computationally expensive than interpolation, the issue is minimizing the cost of regression while keeping the benefits. This is accomplished by evaluating each point on it possible contribution to the regression, an expensive process in itself. Nevertheless, the benefits of regression probably outweigh the drawbacks, and Chapter 8 covers the possibility of incorporating regression into SPLICE.

A major weakness of RFWR is the amount of domain knowledge necessary for operation. SPLICE only requires knowledge of the variables and controls, and a qualitative domain model. In contrast, since RFWR cannot begin to operate without data, RFWR requires an additional performance element to generate data until RFWR learns enough to take over. In Schaal, this performance element is a PID controller. Moreover, the user must specify parameters to control the execution of RFWR itself. There are many many parameters specifying thresholds, learning rates, and other features. In a sense, the user must know something about the environment before applying RFWR to learn the response function. SPLICE places weaker knowledge demands on the user, allowing more autonomous operation. This is a reflection of the different origins of the systems. RFWR is based on statistical tests, which are heavily parameterized, while SPLICE comes out of the autonomous agent paradigm, which requires the agent to learn most domain-specific characteristics.

6.2 Results

All four algorithms, PID, spline fit, RFWR, and SPLICE, were compared under the four experimental methodologies in the three automotive domains. Each algorithm was given control of the gas pedal and tasked to achieve a desired speed after a specific interval. The algorithms were also allowed to use a negative pedal position to achieve negative speeds. Each algorithm received approximately equivalent additional domain knowledge, as individually required. The weights for PID control were tuned to optimize performance. The spline fit was initialized with four points in the corners of the problem space, similar to the output of a qualitative physics reasoner. RFWR's parameters were tuned according to the author's instructions, and a secondary performance element was provided. The recommended performance element, PID control, was not accurate enough in all domains to generate satisfactory data, so RFWR used an optimal control algorithm computed from the internal dynamics of each environment. Optimal control alone did not generate the necessary "cloud" of data points for regression, so a small amount of Gaussian noise was added to the optimal control to generate well-conditioned data. SPLICE simply required

the knowledge that pedal position and speed are directly related. SPLICE also operated under the additional constraints that the pedal range and the resolution of the pedal were limited.

The competing algorithms each required slight changes to some of the methodologies. The soft real-time methodologies require a criteria for abandoning the current goal. Since none of the algorithms have a mechanism for determining when a goal is unachievable, they all have a threshold of 16 attempts before going on to the next goal. The repeated sequence methodologies require a stopping criteria for when the sequence is learned as well as possible. Although the sequence was designed so that it is possible to get from one goal to the next in a single control movement, there is no guarantee that the learning algorithms will find these controls. Also, since the algorithms modify their data structures at many points during each learning experience, it is difficult to monitor what meaningful learning is taking place. This is unlike SPLICE, where learning a case is a meaningful learning event. Instead, the stopping criteria was the change in error on the entire sequence. Since PID control does not learn, there is no change in error and it stops after the first sequence. The spline fit was stopped when the residual error for the entire sequence, $\sqrt{\sum (D-F)^2/n}$, where n is the number of steps in the sequence, 10, did not decrease. Since RFWR at first refused to make any predictions at all (all weights zero), it automatically ran a few sequences, then applied the same stopping criteria as the spline fit. As before, SPLICE ran until it failed to learn a single case in an entire sequence.

6.2.1 The Linear Domain

Since this domain is so simple, all algorithms were expected to learn the domain perfectly by the end of training, but this was not the case for all methodologies. Overall, PID performed optimally because it was manually tuned, and this domain obeys the restriction that the response surface be linear. Since PID performance was completely uninteresting, it is not included in the presentation of this domain. The spline fit was also quick to learn the response surface because the surface is continuous. SPLICE and RFWR learned slower and with more error because they make no global assumptions about the nature of the response surface. The power of these two approaches is evident in the more complex domains.

Random Hard

These experiments were the most difficult for the algorithms for the linear domain. Figure 6.1 is the graph of the change in error in performance as training progresses. As before, the error is the median error for six trials in the first 100 problems and three trials in the second 100 problems. The experiments were particularly difficult for RFWR, because each receptive field requires a certain confidence level before it can make a prediction, and the training data were too scattered to allow much confidence before the end of training. As expected, subsequent repetitions of the training data (not shown) allowed RFWR to increase confidence, and once the confidence passed the threshold predictions were quite accurate. The spline fit generalized over the scattered training points well, effectively learning the response surface after only about 25 problems. SPLICE also generalized well, perfectly learning the response surface after only 20 problems. The unusual spike near problem 20 illustrates how local methods can be locally very inaccurate because of deficient data in the neighborhood, but global methods like the spline fit have a much smoother learning curve in environments that fit their requirements.

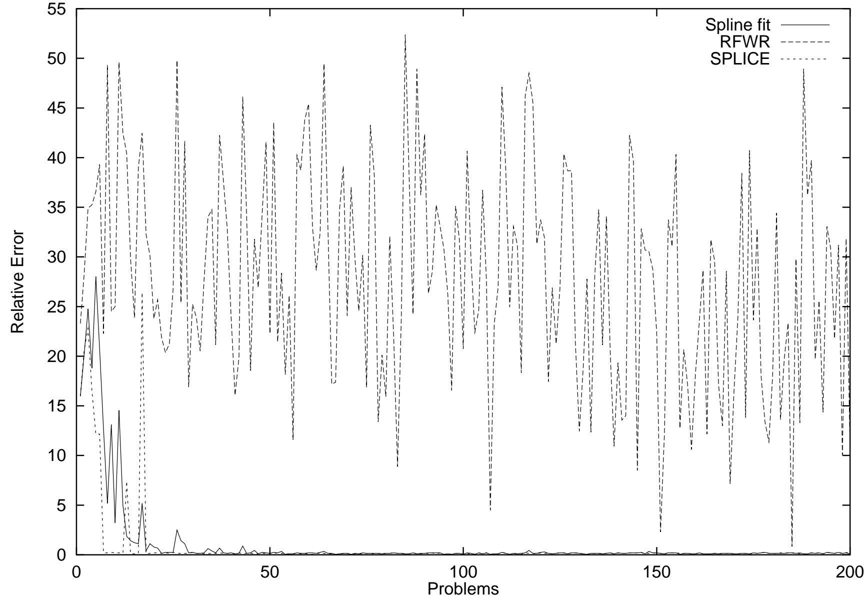

Figure 6.1: Relative error in random-hard, the linear domain.

Random Soft

The next set of experiments for this domain were much easier for all algorithms, particularly RFWR, as illustrated in Figure 6.2. Although still much slower than the others, RFWR is able to demonstrate progress on the domain. Interestingly, RFWR exhibits a kind of neural network "forgetting" effect for this methodology. On the second repetition of the training sequences, RFWR performance suddenly deteriorates (not shown). None of the competing algorithms seem to exhibit SPLICE's occasional need for three adjustments to achieve a goal, as described in Chapter 5. In the case of the spline fit, this is probably because the algorithm defines the acceptable region around a desired value with the desired value in the middle, so rounding errors never cause the final speed to miss the desired region.

Repeated Hard

The repeated methodologies were much less challenging for all of the algorithms. The spline fit and SPLICE again had similar learning curves, except the spline fit curve was again more monotonic than SPLICE. These algorithms learned the sequence perfectly after two repetitions. RFWR was again substantially slower, requiring over twenty repetitions before converging. The graph for this methodology is in Figure 6.3.

Repeated Soft

The results of these experiments, graphed in Figure 6.4, are similar to the other methodologies. The spline fit converges extremely rapidly, never making a mistake after the fifth goal. SPLICE is almost as fast, converging after the twelveth problem. RFWR is again an order of magnitude slower, failing to converge until the seventh complete repetition of the ten-goal sequence.

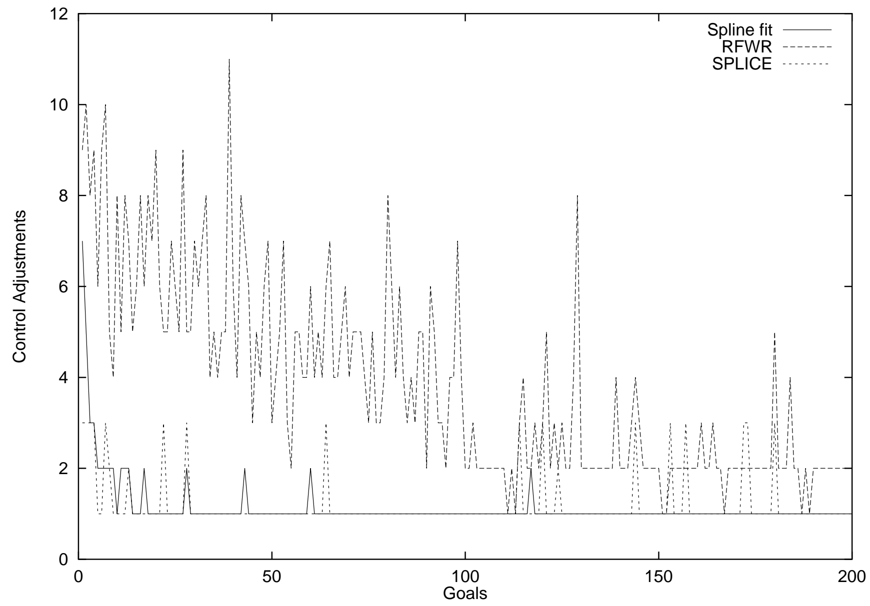

Figure 6.2: Control adjustments in random-soft, the linear domain.

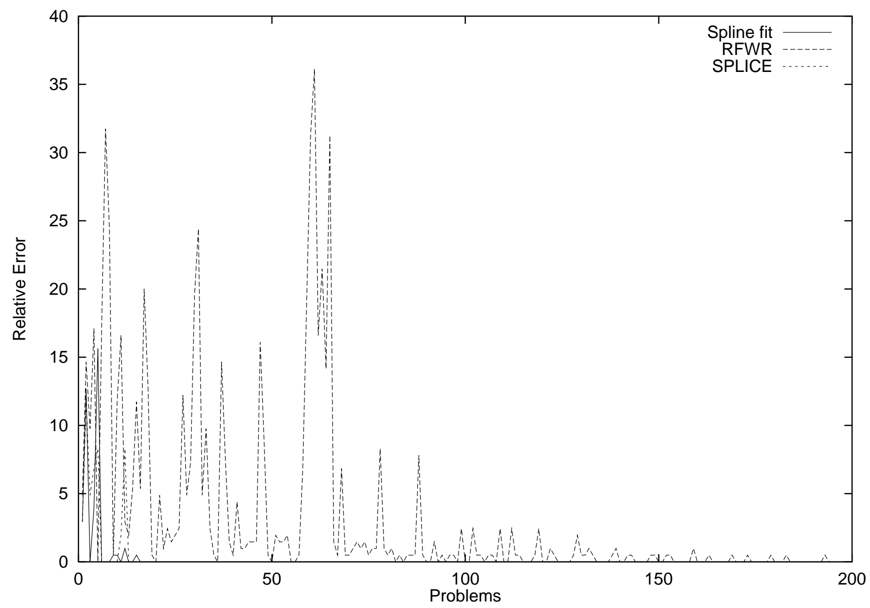

Figure 6.3: Relative error in repeated-hard, the linear domain.

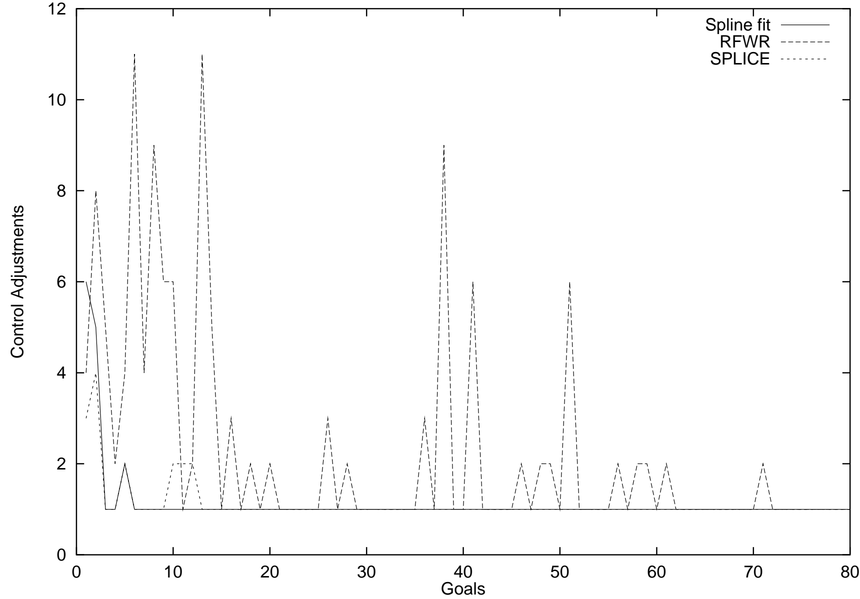

Figure 6.4: Control adjustments in repeated-soft, the linear domain.

6.2.2 The Non-linear Domain

This domain was designed to highlight differences between algorithms for non-linear relationships. Since this domain equation produces a continuous curved surface, PID control was not expected to perform well. In fact, experiments with the PID algorithm revealed that it was not capable of performing at an acceptable level for most methodologies. Therefore, PID control was upgraded to be more highly reactive. The highly reactive version of PID had an opportunity to change its control ten times while attempting to achieve a desired value, whereas the other algorithms were forced to keep the control constant until the time the goal should be achieved. The justification for improving the reactivity of PID is that it is simple enough to run more often than the more sophisticated control algorithms in a real-time situation. After tuning, this version of PID performed well enough to be competitive with the other algorithms. The spline fit was expected to be extremely accurate very quickly, because splines were developed for representing smooth surfaces. SPLICE and RFWR, the local linear techniques, were expected to initially perform badly then improve their performance as their local models became more accurate. Note that even though the underlying domain is non-linear, a sufficiently local linear model can be arbitrarily accurate.

Random Hard

Experiments with this methodology show a uniform decrease in performance compared to the linear domain, as illustrated in Figure 6.5, especially for the spline fit and SPLICE. RFWR is again very slow to converge, but the spline fit converges to be the most accurate algorithm by the end of the training period. PID control performance varies as the appropriateness of the current problem to its weights varies. SPLICE is much slower to converge because it needs at least two points in the neighborhood of a problem to accurately interpolate the control, which is rare in this highly scattered training set.

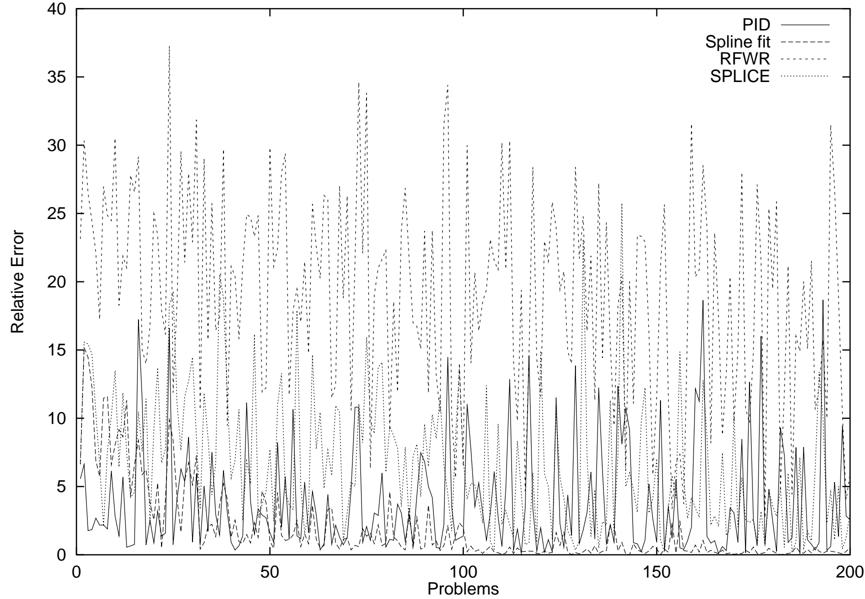

Figure 6.5: Relative error in random-hard, the non-linear domain.

Random Soft

Like the linear domain, RFWR in the non-linear domain, Figure 6.6, showed a marked improvement in performance when given the opportunity to attempt to achieve a goal multiple times. RFWR was competitive with SPLICE, while the spline fit was slightly better, although even spline fit did not ever perfectly learn the domain. PID was surprisingly unable to ever achieve the goal in a number of problems, probably because inappropriate weights sent the algorithm into a situation from which it was impossible to recover. The graphed line for PID looks clipped because the soft methodologies force PID to give up on a goal and go to the next one after 16 attempts.

Repeated Hard

Figure 6.7 graphs the relative error of the four algorithms in the non-linear domain. PID performed well, because its weights were tuned on this exact sequence. Of the three adaptive algorithms, the spline fit was most successful, only generating one error of more than 2% after the first two sequences. RFWR in this experiment learned faster than SPLICE, primarily because of SPLICE's difficulty in achieving the seventh goal, accelerating to highway speed. This is primarily because of two factors:

1. This goal is significantly different (higher) from all other goals, so little transfer of results is available.
2. SPLICE must first solve the previous problem in order to get the correct initial speed.

Since RFWR does not train on its own performance like SPLICE, it uses an optimal control strategy for training. This ensures that the same amount of training data is available for every goal, so no goal should be particularly more difficult than another.

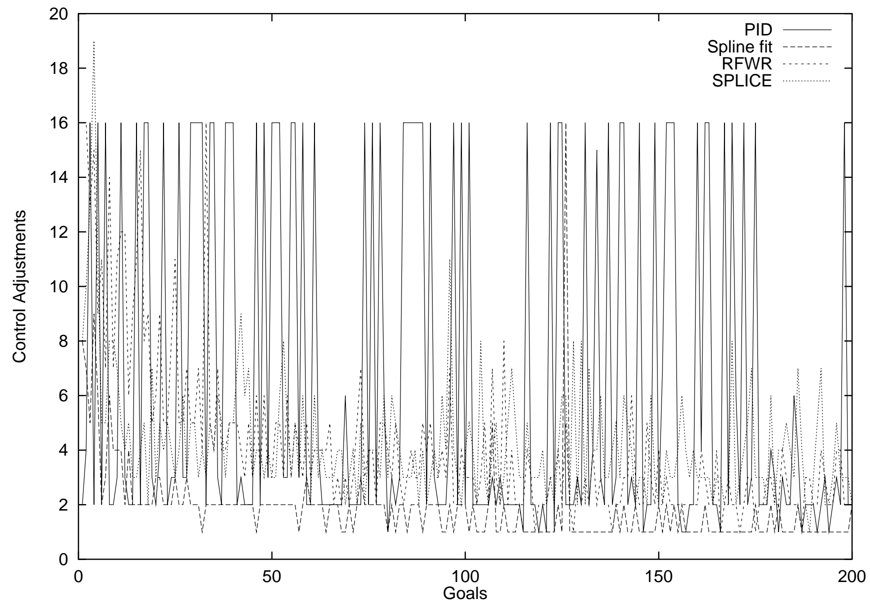

Figure 6.6: Control adjustments in random-soft, the non-linear domain.

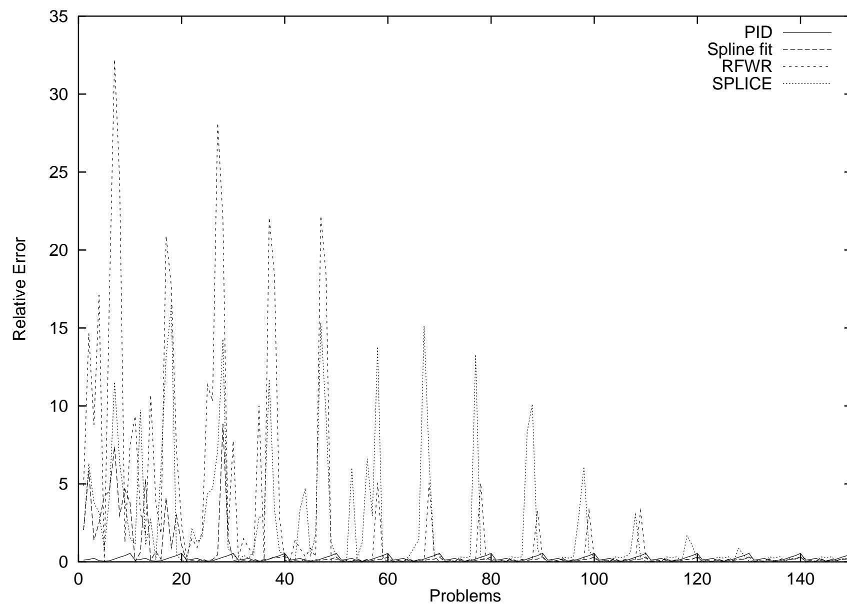

Figure 6.7: Relative Error in repeated-hard, the non-linear domain.

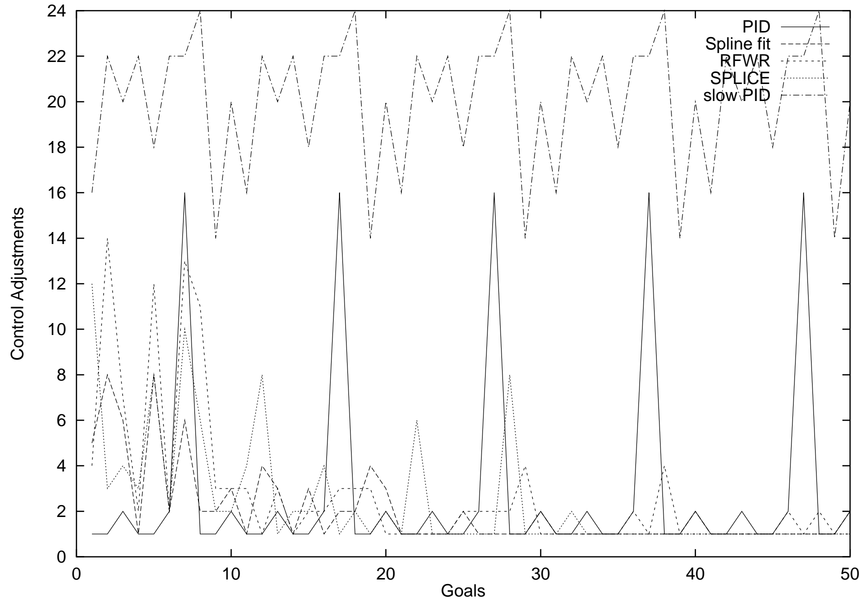

Figure 6.8: Control adjustments in repeated-soft, the non-linear domain.

Repeated Soft

The results for the final methodology are in Figure 6.8. This graph plots not only the regular PID control but also the "low reactivity" PID version, which only gets one opportunity to change its control each cycle. SPLICE converges before any other algorithm in this methodology, because for some reason the spline fit is unable to learn to achieve the final goal in the sequence in a single control movement. RFWR has even more difficulties with this methodology after the ninth repetition (not shown). Due to the remaining error, it splits a receptive field, and "forgets" how to achieve one goal. It finally learns the entire sequence after dozens of repetitions.

6.2.3 The Discontinuous Domain

Since this domain is basically linear, PID control was expected to perform acceptably. However, it is impossible for a spline surface to become discontinuous, so the spline fit was expected to be very poor. The local model algorithms, RFWR and SPLICE, had the potential to learn to perform well, since the domain is composed of linear patches, but since arbitrarily close points could be part of different patches, it was not clear if the algorithms would find the right subdivision.

Random Hard

RFWR continues to perform poorly with this methodology, but the other algorithms are successful, even the spline fit, as displayed in Figure 6.9. The spline fit probably performs acceptably in this domain because the data is sparse enough to hide the discontinuities. The spline surface is only unable to fit groups of data on opposite sides of a discontinuity, and the data for this methodology never collects into any type of pattern.

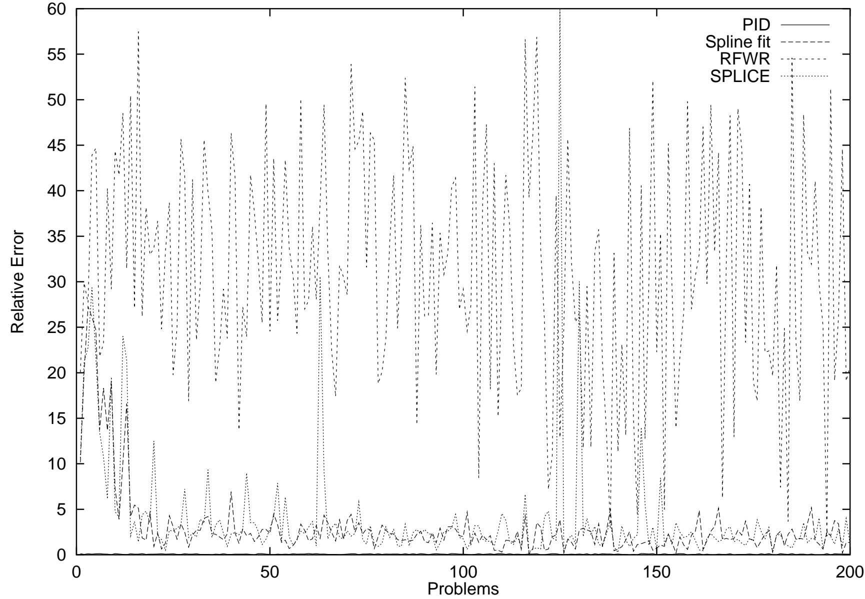

Figure 6.9: Relative error in random-hard, the discontinuous domain.

Random Soft

Conversely, these experiments were extremely difficult for all of the algorithms. Figure 6.10 shows that none of the algorithms were able to perform well. The comparison algorithms frequently reached their limit of 16 attempts before giving up. SPLICE did not have a strict limit for the soft methodologies, as described in Chapter 5, so it occasionally used more than 16 attempts. The algorithms performed poorly because the discontinuities in the domain cause small control changes to have large effects, so it is difficult to monotonically approach a desired value. Since the goals and initial states are random, the local algorithms do not have enough data to construct accurate local models.

Repeated Hard

Figure 6.11 graphs the relative error of the four algorithms in the discontinuous domain. As expected, PID was acceptable, but the spline fit was never very accurate, and became worse as the number of constraint points increased. RFWR and SPLICE both eventually learned the domain. Although both algorithms were relatively perfect after completing the second sequence, SPLICE has a periodic noticeable error until late in the experiment. This is because the last goal was impossible to precisely achieve in one control adjustment after successfully achieving the previous goal, because it was in a discontinuity. SPLICE tried control movement A, which failed badly, then learned a case including the point (I, F, A). The next sequence, SPLICE recalled the failure with A and tried B_i, which failed less severely. However, SPLICE did not select (I, F, A) as a point for the new case, so in the next sequence SPLICE was not aware that A had failed, so it chose A again. SPLICE eventually gets within its margin of error for the desired value because the control B_i is different for each sequence. This problem would not occur if SPLICE stored multiple points for each case, as discussed in Chapter 8.

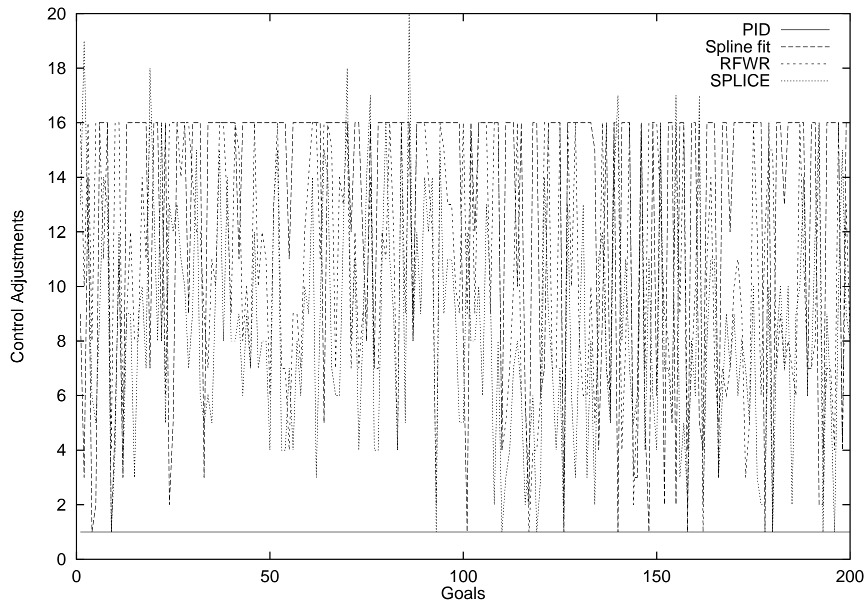

Figure 6.10: Control adjustments in random-soft, the discontinuous domain.

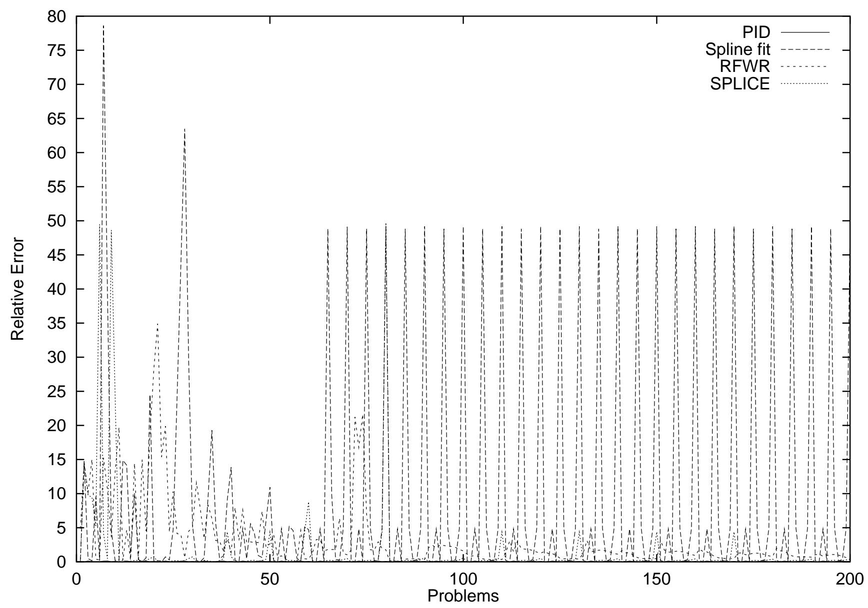

Figure 6.11: Relative error in repeated-hard, the discontinuous domain.

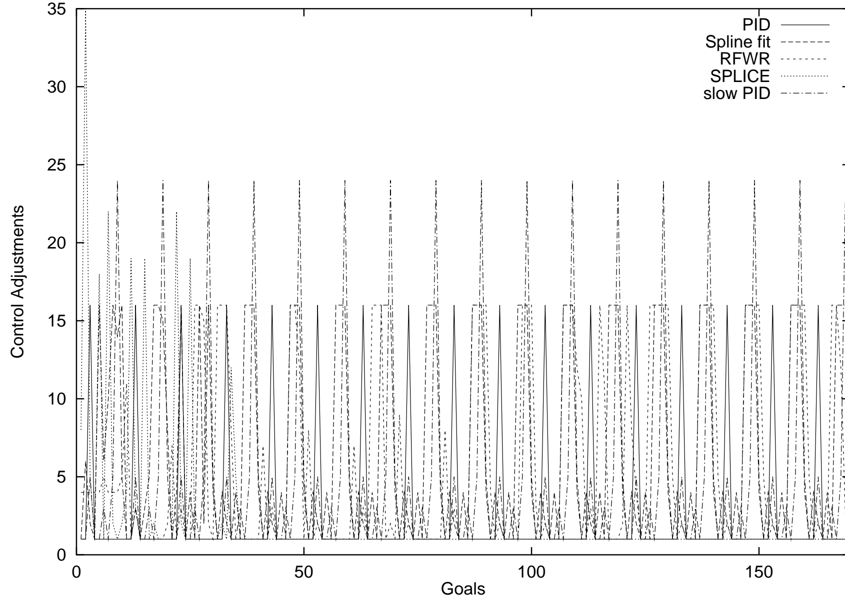

Figure 6.12: Control adjustments in repeated-soft, the discontinuous domain.

Repeated Soft

The final methodology proved to be almost as difficult as the **random-soft** methodology. As Figure 6.12 shows, SPLICE was the only algorithm to learn the entire sequence. Even PID was not able to achieve all the goals in the sequence. The spline fit and RFWR performed approximately equivalently, with neither algorithm able to complete the entire sequence. Subsequent tests showed that a problem with RFWR was the error added to the control for training. Recall that the training procedure for RFWR was to compute the correct control, add some Gaussian error, execute it, and train on the resulting data point. The purpose of the Gaussian error was to distribute the data around query points to improve the accuracy of regression, but for this domain it was more important for the training data to be accurate than well-conditioned.

6.3 Discussion

In general, high-frequency PID performed well in all domain, but low-frequency PID (same frequency as the other algorithms) only performed well in the linear domain and the least challenging methodologies of the discontinuous domain. The spline fit performed well and learned quickly in the linear and non-linear domains. It even approximated the discontinuous domain as well as the other algorithms under the random methodologies, because the data was too sparse for any algorithm to have an opportunity to model the discontinuities well. However, the spline fit was unable to improve its performance in the discontinuous domain for the repeated methodologies because the data was too dense local to the discontinuities, violating its assumptions. Receptive Field Weighted Regression improved its performance in all domains and methodologies, but its learning rate was slow despite careful adjustment of its many parameters according to instructions from the software author. SPLICE also improved its performance in all experiments, and it generally learned faster than RFWR. Additionally, SPLICE was able to train on its own performance,

and it did not require extensive parameter tuning to achieve optimal learning.

One remarkable distinction between SPLICE and the other adaptive algorithms is that SPLICE explicitly attempts to correct its knowledge in order to achieve its goal, while the other algorithms implicitly add data in hopes of getting a more accurate model. A consequence of this characteristic is apparent in the discontinuous domain. In the **repeated-hard** experiments, the spline fit achieves a local minimum error at the fifth sequence, then begins to perform terribly at the sixth sequence. It is unable to reason about how well it is achieving its goals, so it continues performing poorly instead of going back to its previous performance. RFWR has a similar problem in the **repeated-soft** experiments with the last goal in the sequence, and eventually splits a receptive field to handle the last goal, which improves its performance on the the goal but actually increases the overall error.

The most notably poor performance for these experiments are the failure of SPLICE and RFWR in the random methodologies. There are two possible approaches to improving this performance. Both approaches reason about the number of possible maximally-specific cases for SPLICE. Since the most specific interval is 16 units wide, and the range of possible values is 4096, there are 256 possible regions for the initial speed and 256 for the desired speed, allowing up to 65,536 possible cases. One approach is to attempt to cover more of these possible cases by extending the number of random problems to 500. Another approach is to reduce the number of possibilities by extending the most specific interval to 32, reducing the number to 16,384. This approach has no effect on RFWR in the **random-hard** methodology, because it does not reason about success or failure in achieving the goal, and it only affects SPLICE when it determines if recalled points are close enough to the goal to use. The results of these additional experiments are in Figure 6.13. These changes did not measurably improve performance for either SPLICE or RFWR.

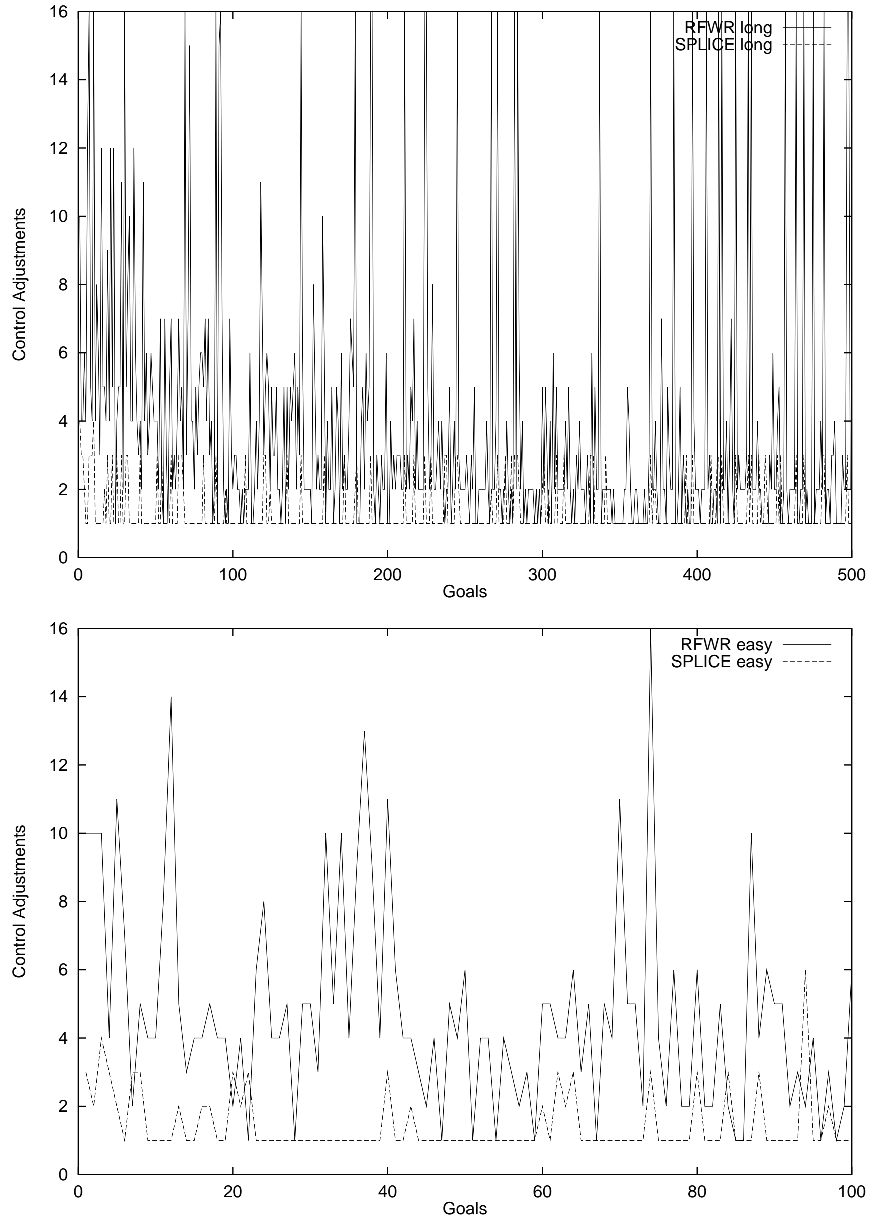

Figure 6.13: Attempts to improve performance for random-soft, the linear domain.

CHAPTER 7

Experiments in a Complex Domain

Flying in an airplane, looking out the window,
Watching the clouds go by ...

Although experiments in simple artificial domains are useful to analyze and compare algorithms, it is also important to apply new architectures to complex domains with a legitimate need for learning. Since these domains are more difficult to analyze, the emphasis of this chapter is on performance evaluation and issues related to the extra complexity in such domains.

7.1 The Flight Simulator Domain

Simulated flight is an ideal challenging domain for continuous-valued adaptive control. The domain dynamically changes over time, there are a number of numeric features available for monitoring, and there are a number of numeric effectors to control the plane's behavior. A qualitative model easily expresses overall domain characteristics, but the specific dynamic behavior is complex enough to require learning.

To ensure that experiments are as valid as possible, the flight simulator that was used was developed independently and distributed as a game demonstration for Silicon Graphics computers. Although the simulator contains a number of plane models, all experiments used the Cessna model. We obtained the source code for the simulator and inserted socket communication code to allow interaction with a Soar process. The communication model between the flight simulator and Soar has two parts: flight information and commands.

Since the communication bandwidth over the network is limited, it is necessary to keep information flow to a minimum, yet still keep Soar current enough for its responses to be relevant. A set of communication parameters modulates the flow of information. The *report* parameter defines which of the many flight variables are tracked. The *delta* parameter defines the minimum change for each variable necessary before the flight simulator sends its new value to Soar. This ensures that the flight simulator will not overload the network with insignificant changes. Finally, the *delay* parameter defines the minimum time delay for each variable since the last update before the flight simulator can send an update. This ensures that rapidly-changing variables will not overload the network. A variable must satisfy all three parameters before the flight simulator sends it on the network.

The commands communication model is simply the same as the user input actions. Possible actions are moving the mouse to control the elevators and ailerons, and using the keyboard to control the throttle and flaps. There are additional high-level commands

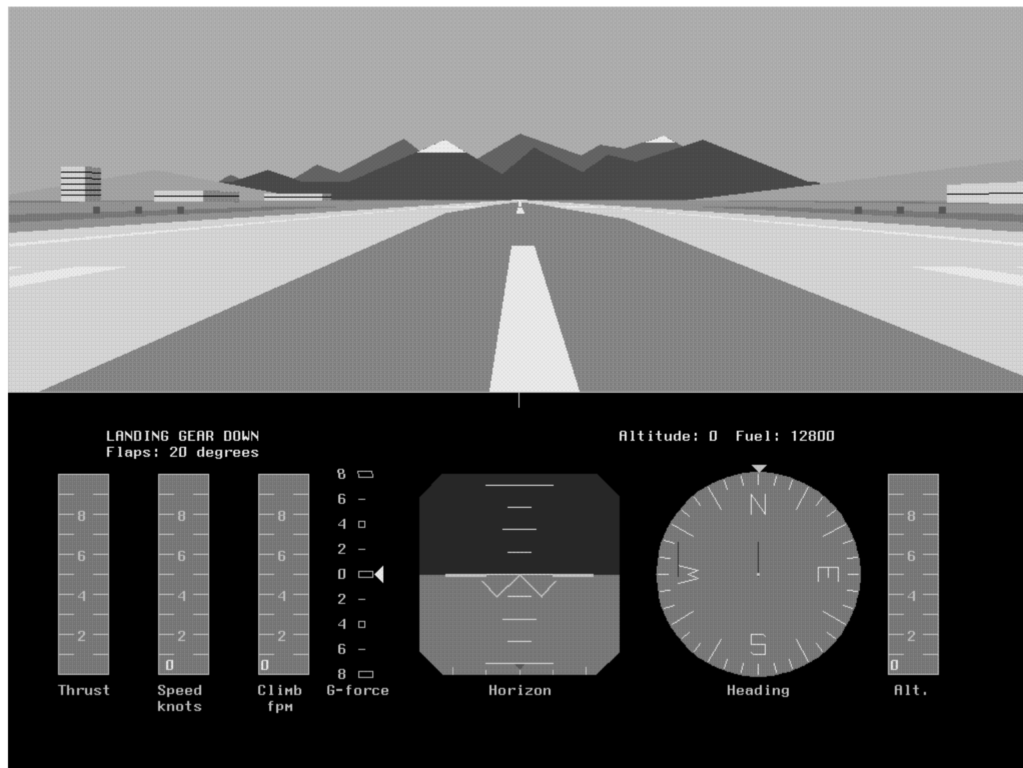

Figure 7.1: Screen view of the SGI flight simulator.

developed for other airplane control research such as the Air-Soar system [43] presented in Chapter 2, but these were not used for this research.

The visual interface for the flight simulator consists of a cockpit and rendered exterior view. The simulator regularly accepts commands from its communication channel or the user and updates the state in quasi real-time. Figure 7.1 displays the flight simulator screen. The World Wide Web site [48] contains more complete information concerning the SGI flight simulator.

7.2 Controlling Speed

The simplest goals to achieve in this domain are the same as the artificial driving domains: speed. The first experiments with the flight simulator were designed to demonstrate that SPLICE can learn realistic, complex response functions. It is first necessary to choose a methodology for the experiments. In this domain, it is important to achieve goals in a timely fashion, but there is no absolute deadline except the requirement to avoid crashing. Since the reward for achieving a goal decreases gradually over time, it fits the definition of a soft real-time domain. Since pilots normally follow the same or similar flying sequences to accomplish a task day after day, a repeated goal sequence seems more appropriate. Combining the soft real-time and repeated sequences requirements, we chose the **repeated-soft** methodology.

The domain knowledge for simulated flight is slightly more complex than the artificial domains. The major differences are in the control interpolation and transmission to the environment. Interpolation is more complicated because the agent must round the throttle

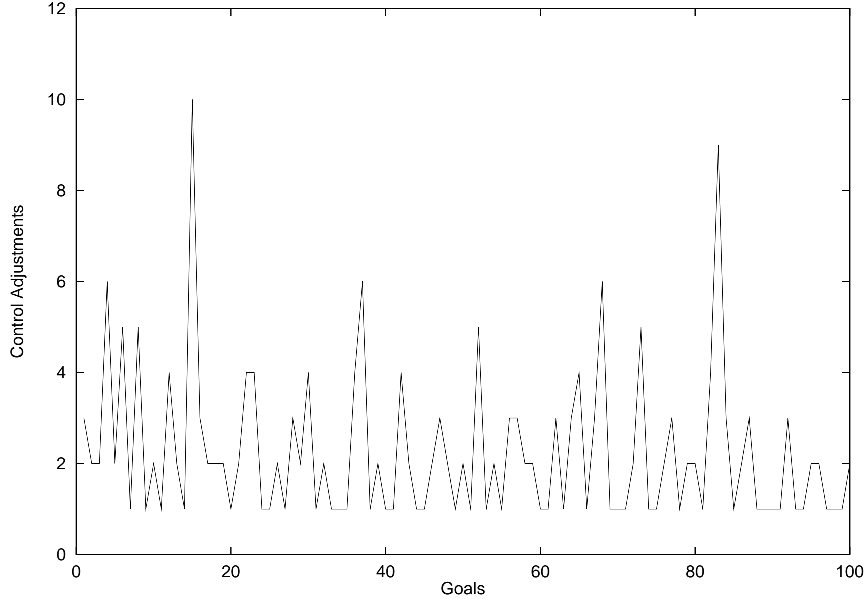

Figure 7.2: Control adjustments for the simulated flight domain.

to a multiple of five, and its range is from -100 to 100. Although the Soar architectural base handles the details of communication, it only accepts commands to increase or decrease the throttle by a increment of five. Effectively, the throttle control level for the flight simulator is the time derivative of the throttle, but the most convenient level for SPLICE is direct throttle control. Domain knowledge corrects this problem by generating the appropriate number of increase/decrease throttle commands based on the current and interpolated throttle positions.

The sequence of speed goals was similar to the sequence in the artificial experiments, adjusted for the change in units to knots. The learning curve for the sequence of ten goals is in Figure 7.2. Note that some of the speeds were impossible to achieve by the deadline, so the agent executed the maximum control movement in the first cycle and a smaller control movement in the second to achieve the goal in two control adjustments.

Besides the additional complexity of the environment, there were several complicating factors SPLICE was forced to handle. One particularly difficult issue was the extra dynamicism of the environment. In the artificial domains, the environment only changed in the interval between sending a control and the deadline. In the flight simulator, the environment was constantly changing. Since the throttle stays constant until changed, a high positive or negative throttle could change the speed greatly from the final state of one cycle to the initial state of the next cycle. For example, if the agent almost achieves a goal with a throttle of 100, by the time it finishes learning, it may already be far past that goal. Although in extreme cases it may be possible for the agent to degenerate into infinite repetitions of 100, -100, this never happened in practice. In practice, the agent was able to treat overshooting the goal as just another problem, and learned how to recover from it. Furthermore, as the agent gained experience with the environment, it encountered situations that it had already learned. In these familiar situations, the learning time was drastically reduced and the agent did not overshoot its goal as severely.

Another issue, which incidentally amplified the previous issue, is that the domain is nondeterministic. This means that, depending on invisible factors like processor and network

load, the environment reacts differently to the same state and control. Since SPLICE never does a global consistency check, this does not greatly affect SPLICE. The only situation where SPLICE needs to deal with the inconsistency is when one of the two points in a case conflicts with the current point (I, A, F). According to the rules for selecting points for a new case, SPLICE rejects old points with the same action as the current action. This is a valid alternative since response times tend to change slowly, so it is better to keep new data than old data. Nondeterminism interacts with the computation time issue because the agent rarely finds situations identical to one of the two points in the matched case, even if the initial state and action are the same. Since familiar situations are rare, the nondeterminism forces the agent to slow down to re-learn the environment.

A final issue is that the plane would occasionally spontaneously take off when its speed gets high enough. This also did not cause great problems for the agent, because the plane would land harmlessly when the speed dropped too low for flight. The only complication is that the flight simulator control model disallows negative throttle in flight. Domain knowledge corrects this by stating that the minimum throttle when the plane is off the ground is 0.

The results for controlling speed on the ground in the flight simulator show that it is possible to apply SPLICE to a realistic domain not specifically designed for testing adaptive control algorithms. However, taxiing on the runway is not nearly as challenging as flying because not nearly as many variables are involved, and there is no possibility (at least in this simulator) of the plane being in an unrecoverable situation such as a crash.

7.3 Controlling Speed and Altitude

The next challenge for SPLICE in the simulated flight domain is actually achieving some flying goals. While getting off the ground was trivial, and even happened accidentally in some experiments, the SPLICE theory of control was inadequate to accurately achieve flying goals for a number of reasons.

Since SPLICE expects a sequence of goals achievable by setting a control and waiting for a deadline, the level of control had to be carefully chosen. The major variables a pilot needs to control are speed, altitude, and heading. As shown in the previous experiment, the pilot controls speed directly with the throttle. Although the pilot does not have direct control over altitude, the position of the elevator control directly affects the climb rate of the plane[1]. The throttle level also affects the climb rate, but this implementation of SPLICE only interpolates over one effector variable at a time, so the domain model only linked the elevator to the climb rate. For SPLICE versions capable of interpolating over multiple controls, an additional link between the throttle and the climb rate will include the throttle in future interpolations. Not including the throttle led to difficulties as explained in the final paragraph of this section. Therefore, the level of control must be expressed in terms of speed and climb rate.

The flying task was to simply take off and land. A more autonomous agent may be able to plan the sequence of low-level goals, but in this experiment the following sequence was provided SPLICE by hand:

1. Achieve a take-off speed of 70 knots.
2. Achieve a climb rate of 300 feet per second.

[1] Although not tested in these experiments, the ailerons have direct control over the turn rate of the plane.

3. Level off (0 feet per second).

4. Achieve a climb rate of -300 feet per second.

5. Touch down and stop the plane (0 knots).

The tolerance for speed was eight knots, and the tolerance for climb rate was 60 feet per second. The deadlines for each of the goals were 10 seconds real time. The agent easily achieved the take-off speed after several cycles, but immediately had difficulties with the first true flying goal. The initial climb rate after the agent took off was highly variable, ranging from 120 to 480 feet per second. The agent used the domain model to find a positive relationship between the elevator and climb rate, and slightly adjusted the elevator. Due to the stability of the Cessna model, such small adjustments have no effect on the climb rate. The agent learned this fact and, on the next cycle, adjusted the elevator to a slightly more extreme position. Generally, this still had no effect. Since both controls had the same effect, the point selection heuristics remove the older point, so the agent correctly continues slightly adjusting the control until it observes an effect. Unfortunately, after the elevator leaves the stable region, climb rate begins to change rapidly. By the deadline, the agent was already in an unrecoverable state. If the agent had been attempting to increase its climb rate, it would stall because the angle of attack was too high, and then enter a steep dive. If the agent had been attempting to decrease its climb rate, it would directly enter a steep dive.

One possible performance improvement is to decrease the deadline so the agent is able to react in time to the rapidly changing climb rate. If we let the minimum response time be r, the learning time for the previous cycle be l, the match time for the current problem be m, and the deadline be d, $d \leq r - (l + m)$. Experimental measures show that the total computation time $(l + m)$ is about 20 seconds, and the time between leaving the stable region and entering an unrecoverable state is about 6 seconds, so computation time must be greatly reduced to accommodate the plane's quick response needs. This is difficult but not impossible, since the SPLICE implementation is not optimized, and there are faster CPUs than the one used in the experiments (an SGI Indigo XS24 150 MHz). For example, lesion studies in Chapter 5 show that the FSA case is probably not useful for the **repeated-soft** methodology, so it can easily be removed to save 50% of learning time for most cycles. Another expensive procedure is copying variable values to subgoals. Since SPLICE only uses a few of the available variables from the flight simulator, eliminating the irrelevant variables would save some time.

However, even if it were possible to reduce computation time by 70% or more, this SPLICE implementation would still not learn to correctly control its climb rate. If we assume that SPLICE is trying to increase its climb rate, SPLICE will learn the elevator setting u that causes the climb rate to start increasing rapidly, then SPLICE may be able to change the elevator back before the plane stalls. Even if SPLICE was able to make the perfect control adjustment $p < u$ to stop the increase at the correct climb rate, SPLICE would not be able to repeat the success the next time the agent takes off. SPLICE would recall that the control p led to success, and use that point to interpolate. Unfortunately, p was executed while the plane was in an unstable state, and the plane is currently in a stable state, so since $p < u$, p will have no effect and the agent will have to start over again. The key observation is that changing climb rates is inherently a sequential operation that involves leaving the stable state, selecting a new climb rate, and returning to the stable state.

Perhaps a version of SPLICE with additional domain knowledge explicitly representing this can learn to perform these maneuvers, and this is a valid direction for future work.

A final difficulty with this task is the limitations of SPLICE's interpolation. Since SPLICE attempts to minimize floating point operations, the interpolation only considers the initial value of the desired variable, the final value of the desired variable, and the control. Of course, many other variables have an influence in flight, especially the throttle setting. It is much harder to increase the climb rate when the throttle is low than when the throttle is high, and after achieving a speed goal the throttle position is unpredictable. For example, if the agent accidentally overshoots the take-off speed, the agent may return to take-off speed with a very low throttle. A weak solution to this problem is to specialize the throttle as well as the initial and desired climb rates when learning cases, so SPLICE will only recall points with a similar throttle to the current throttle setting, and as SPLICE gains more experience the recalled throttle will be more similar to the current throttle. A strong solution to the problem is to add more numerical ability to the interpolation and interpolate over the throttle setting as well. Another approach is to add domain knowledge stating that the throttle should be at maximum when attempting to achieve a particular climb rate.

Acquiring the correct model of control for changing altitudes is the most difficult aspect of this task. Anecdotal evidence shows that people learn to fly using a variety of experimental strategies and gradually acquire the correct control model, then begin to refine their knowledge by working on specific task-related goals. Since autonomously learning a control model is beyond the scope of the current SPLICE implementation, in retrospect it is not surprising that SPLICE had difficulties in tasks where a different model is necessary.

7.4 Discussion

There are many issues to consider when applying a general-purpose learning mechanism like SPLICE to a complex dynamic environment. The most important property is that the learning mechanism is flexible enough to incorporate preprogrammed domain knowledge or discover qualitative properties of the environment, such as the relevant variables when pursuing a particular goal and the most effective control model. In the flight simulator example, SPLICE failed to achieve climb rate goals because it did not consider relevant variables such as whether the plane was in a stable state and the current throttle, and it could not induce the appropriate model for changing the climb rate using the elevator. Since this procedure is comparatively simple for a human to observe and encode, it is most logical for the implementor to include this as domain knowledge for the agent. However, autonomous discovery of such high-level regularities, such as Benson and Nilsson's work on learning teleo-reactive trees [5], are interesting research problems. Another possible approach is to integrate observation of human performance [52] to provide basic capabilities and a generalizing module to learn new tasks.

At the lowest level, learning algorithms in continuous-valued environments need to be able to deal with unique characteristics of the environment. For example, SPLICE needs a more sophisticated mechanism of handling nondeterminism. Although SPLICE reacts appropriately to overtly nondeterministic behavior, such as a conflict in the final state from identical initial states and controls, it confuses nondeterminism with monotonicity violations. If the agent detects a locally nonmonotonic point, as defined in Chapter 4, it assumes that the cause is a disparity in the initial states. If the true cause is nondeterminism,

which can be defined as a change in a hidden state, the agent fails to find a difference and throws out one of the old points. In highly nondeterministic environments, this can cause serious problems as false monotonic violations bog down the learning process. A better approach is to define a new variable n corresponding to the hidden state, and attempt to deduce its value from observations [3]. For example, when $n = 1$, the environment responds in one particular consistent way. If the environment begins responding inconsistently, n transitions to 2, and the agent continues learning. If the environment becomes inconsistent again, the agent can decide to either return to $n = 1$ or switch to a new state $n = 3$. If the environment is never consistent, no learning algorithm based on experience can expect to perform well.

A final issue for learning is minimizing the resource requirements of learning. Learning requires computation time, and incremental learning systems must perform learning while the environment changes. This can lead to serious consequences if the system can drift out of a safe operating state, as observed in the flight simulator. SPLICE was able to learn and perform despite the learning delay because it made no assumptions about the relationship of consecutive cycles. This requires SPLICE to sense the new state at the beginning of each cycle, which may be expensive in some environments. Another approach is to explicitly reason about learning time and predict the new initial state. This requires an accurate model of the algorithm's own processing as well as risking poor performance in early stages when the environment model is poorly understood. In many simulated environments, it is possible to suspend dynamic behavior while the learning agent processes data, but this violates the assumption of a realistic environment.

An interesting possibility for minimizing learning time is to learn from one cycle while the next cycle is waiting for the deadline. Although it may not be possible to completely finish learning while waiting, it ensures that the algorithm is as efficient as possible with its time. A problem with this for SPLICE, and possibly other algorithms, is that performance for a cycle cannot use the result of the previous cycle. For example, if a SPLICE agent tried a control u and did not change its state by the deadline, the next cycle would have the same initial state, so it would repeat u, then learn that u is ineffective, and not make more progress toward the goal until the next cycle.

Although SPLICE was unable to learn to control the airplane at the desired level of expertise, it was an effective tool in exploring control in complex environments and revealed important flaws in the SPLICE control model. Using these results, future versions of SPLICE should have more success in these and other complex, dynamic environments.

CHAPTER 8

Discussion and Extensions

Just imagine, just imagine,
Imagine all the places you can go and see,
Imagination's fun for you and me!

As presented, SPLICE is a functional system for symbolically learning motor control knowledge in continuous-valued domains. I have shown SPLICE to perform comparably to state-of-the-art numerical adaptive algorithms in some simple artificial domains, and learn to perform maneuvers in a very complex domain, simulated flight. However, SPLICE has the potential to apply to a much broader range of domains with several enhancements. This chapter covers some properties of SPLICE and some possible improvements to make SPLICE a stronger learner.

8.1 Contribution of Soar to Functionality

Developing any system on a high-level architecture such as Soar always involves trade-offs. Since Soar is specifically designed to solve problems requiring explicit representation of knowledge, it has several critical properties for this research and many other useful properties. One of Soar's strengths is efficiently handling large amounts of knowledge, and this was very important for SPLICE. Since the number of cases in SPLICE grows as it receives more experiences, the ability to match rules in parallel allows SPLICE to retrieve cases in constant time irrespective of the total number of cases in the database. Also, the learning mechanism generates new cases by backtracing through problem-solving, so it is also independent of the total number of cases. Since neither learning nor performance depends on the total number of cases, SPLICE is incremental. Although it is possible to implement an incremental system without Soar, Soar already contains mechanisms facilitating the incremental capability.

Soar also includes a number of helpful properties that reduced development time and encouraged useful research directions. Representing knowledge as rules, which implements an associative connection between sensed conditions, desired conditions, and local models, is a natural flexible representation for response functions. When no case matches a particular problem, the impasse mechanism naturally leads to the qualitative reasoning subgoal, and resolving the subgoal automatically creates the initial maximally general case via chunking. Restricting access from procedural knowledge helps to force incremental processing, since in general the agent cannot deliberately search all cases in the case database. Finally, the non-sequential nature of rule-based programming allowed easy re-ordering of execution and

other rapid prototyping advantages.

Although helpful in the case of the qualitative model, this research did not extract much advantage from the chunking mechanism. Chunking was designed to automatically summarize the knowledge derived in subgoals so future processing does not need the subgoal, but the most common subgoals in SPLICE were retrieving cases and learning new cases. Since these subgoals store the case database knowledge, it does not make sense to use chunking to avoid using them in the future. Instead, SPLICE inhibits chunking for these subgoals by generating an over-specific chunk which never matches again, as illustrated in the trace in Appendix A. Although chunking was designed to automatically learn from problem solving, the subgoal which adds cases to the database is forced to do problem solving specifically for learning. This violates the assumption that learning through chunking is automatic and instead makes it deliberate. However, it is possible that future versions of SPLICE will include additional knowledge in this subgoal, such as natural language instructions, and the chunking mechanism will automatically backtrace through this knowledge to include it in new cases.

Overall, Soar was found to possess a number of mechanisms and properties that were useful to the implementation Those properties that were not relevant to SPLICE may become more important in future versions, and even the inconvenient chunking mechanism may become more useful in the future.

8.2 Multiple Levels of Control

A major shortcoming of the SPLICE architecture is its inability to learn to achieve goals using more than one control manipulation. For example, SPLICE can learn to control speed by applying pressure to the accelerator over time, but it cannot achieve a particular position and speed without multiple control movements, in general. If we suppose the agent wanted to achieve position x with no speed at time t, the agent would need to accelerate to a cruising speed, travel near the desired position, and decelerate so that the agent stops at the right place and the right time. This is a plan for achieving a complex goal, expressed in steps of smaller subgoals. The plan includes a number of parameters, such as the cruising speed and deceleration time. Although the plan itself is high-level and fairly intuitive, the specific parameters may be difficult to calculate.

This is similar to SPLICE's approach to solving simple goals, where the domain model is intuitive and explicitly presented to the agent, and the local linear models are nonintuitive and automatically learned. For complex goals, the case-based approach is also valid. Cases would contain parameterized plans instead of points, so the agent can learn how to achieve the subgoals using the standard single control movement technique described in this document, and the agent can also learn how to parameterize the plans.

Figure 8.1 illustrates how SPLICE may perform and learn across multiple levels. SPLICE first classifies the current problem to the best match S_1 and retrieves the first step to the solution, accelerating to a cruising speed s_c by time t_1. Since there is no action associated with this plan, Soar impasses, and attempts to solve the subgoal of achieving the speed s_c. This is a goal SPLICE can achieve with a single action in the environment, so the agent finds the best match M_1, computes an action, and tries it out. After waiting t_1 seconds, the agent can evaluate the effects of its action and generate a more specific case from M_1. At the same time, the agent can get the next step in the plan S_2 from matching the new conditions and the goal to the nearest case, maintaining (or achieving) the speed s_c until

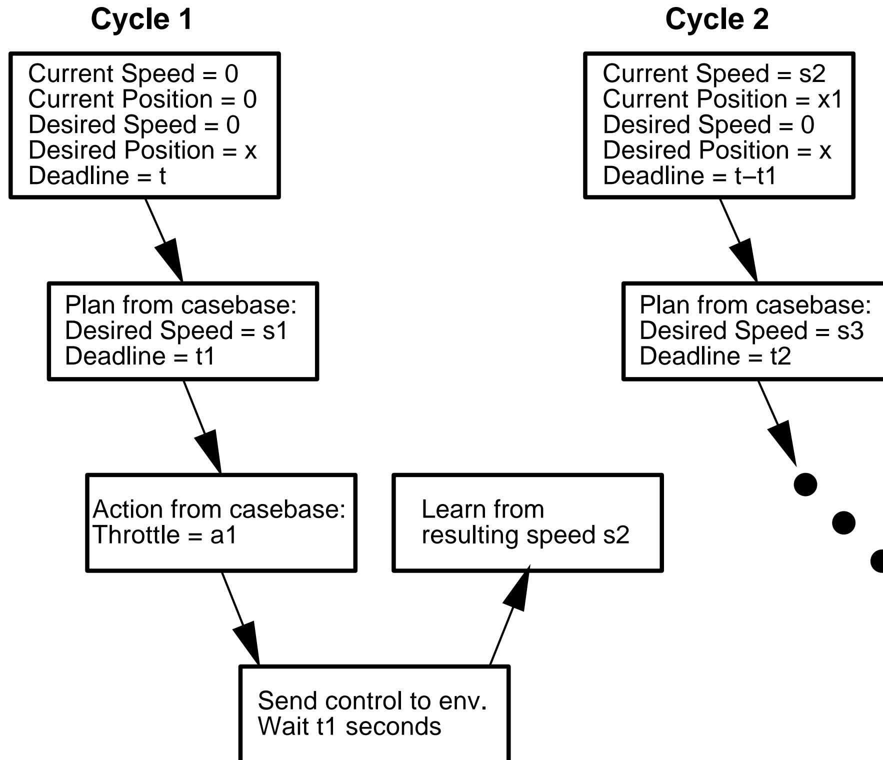

Figure 8.1: SPLICE behavior for multiple levels of control.

it is time to slow down at time t_2. Next SPLICE repeats the low-level action mapping, execution, and learning. Finally the agent retrieves the last step in the plan S_3, slowing to a stop by the overall deadline at the correct position. After the agent executes the entire plan, it has an opportunity to learn about the plan. If all primitive actions were successful but the agent did not achieve the overall goal, there must have been something wrong with the plan, and the agent must learn over the plan steps P_1, P_2, P_3. Even if the plan is analytically correct, as it should be in this simple example, the domain may make achieving the primitive goals impossible. For example, a spaceship may be able to achieve a very large velocity over time by continual thrusting, but a vehicle in an atmosphere has a maximum speed because of friction. There may also be an additional goal relating to fuel expenditure which may require learning at the planning level. The IMPROV system [44] is a recent attempt to autonomously revise planning knowledge in Soar.

The agent can also utilize this framework to experimentally optimize its behavior. For example, it can search for the fastest possible time to move a certain distance without a net speed change. For this task, the agent must solve the problem for different deadlines and declare the smallest deadline for which the plan was successful the optimal time. The case-based performance and execution framework appears to be a powerful and extensible mechanism for symbolic systems situated in continuous-valued environments.

8.3 Region Learning and Partitioning

SPLICE currently automatically classifies variable values into all regions of which they are a part. SPLICE implements this by automatically generating rules for every possible

region. With one rule per region, SPLICE requires approximately 1500 productions for the flight simulator environment. It would be more efficient and more cognitively plausible to automatically generate rules on the fly. This could work by generating an impasse whenever a variable cannot be classified to its most specific region. The subgoal would determine the correct region for the most general unclassified level, and the result would create a chunk identical to the automatically-generated region classification rules now used [45]. This enhancement is not included in SPLICE at this time because we judged action learning to be a more interesting research problem than region learning. Also, region learning is independent of the environment laws, so it would not affect SPLICE's performance in the evaluation.

8.4 Relaxing the Monotonicity Constraint

Although SPLICE was designed with few assumptions about the nature of the environments in which it will execute, it does assume that the relationship between control and desired variable is monotonic. This assumption is necessary to establish whether an expectation failure has occurred, which in turn determines whether an initial state variable or a desired state variable will become more specific, as described in Chapter 4. SPLICE can relax this assumption by simply focusing on the difference between the best match initial and desired states and the current initial and final states. In a manner similar to learning the FSA case, the variable to specialize is the variable most different from the matching case, using the MGD function defined in Chapter 4. Since this modification would not always correctly determine whether to specialize a state or goal variable, performance would be expected to decline for monotonic environments. It is important to note that violating the monotonic assumption may not necessarily disable SPLICE, since the performance element does not depend on the assumption. The major effect would probably be poor generalization to other goals, because the agent would detect expectation failures and specialize state variables more often, forcing goal regions to be very general and inaccurate. Also, the introduction of nonmonotonicity allows the possibility of local maxima, from which no algorithm can be guaranteed to escape. The creation of a suitable nonmonotonic environment for the evaluation of these hypotheses is an important goal for future work.

8.5 Simultaneous Goals and Multiple Effectors

Another major shortcoming of SPLICE is its inability to learn to achieve goals containing multiple desired values. When the desired values are controlled by independent effectors, the problem is no more difficult than the single desired variable case. For example, if SPLICE were controlling a two-armed robot (with possibly different dynamics for each arm), the agent could independently compute interpolations and correct errors. However, if a control affects more than one desired variable, such as speed and position or climb rate and heading, the problem becomes much more difficult.

On the performance side, it may be possible to incorporate multiple interacting desired values by interpolating over all the desired variables, similar to the proposed extension of interpolating over all initial variables. Since interpolating over multiple variables is too computationally expensive to implement in Soar, this would have to be implemented as a Soar external command. Learning over multiple desired variables would simply be a matter of specializing the matching case to the most general difference of all desired variables.

Using multiple effectors at the same time to achieve one or more desired values presents similar difficulties. Again, it is possible to interpolate matching points to find control movements for two effectors. For example, an airplane controller needs to use both ailerons and elevators to maintain altitude while turning, because banking the plane to turn reduces lift and causes the plane to drop. If the agent moves the elevators too little, the agent will notice the plane dropping and try to correct, but it may try to move the ailerons less, which would reduce descent but also reduce turning. The major obstacle to success for these enhancements is "the curse of dimensionality," [4] where each additional input or output dimension causes an exponential increase in the number of possible states. It will probably be necessary to add more domain-specific knowledge, and restrict the experiments to performing specific tasks, rather than deriving the entire response hyper-surface.

8.6 Maintenance Goals

Another possible type of goal not addressed in this thesis is the homeostatic or maintenance goal. For these goals, the agent attempts to keep a variable in a desired range for some time. The goal representation presented in this thesis can roughly approximate maintenance goals by using a number of very short deadlines for the same desired interval, but it is not efficient because learning could potentially occur at each deadline. A better approach is to first achieve the desired interval using a standard deadline, then monitor the environment until the variable value leaves the desired interval. When the value leaves the interval, the agent attempts to learn a control adjustment that would not cause an error in future similar situations, and tries to re-achieve the desired interval. Eventually the agent should learn the correct control responses to stay in the desired interval even as other relevant variables change.

8.7 Generalizing Over Time

In all of the experiments performed for this research, the time deadline was a constant, 10 time steps for the artificial domains and 10 seconds for the simulated flight environment. The current implementation of SPLICE assumes that the deadline is constant, and would not be able to compensate for variable deadlines. Although this was not a problem in demonstrating the functionality of the basic algorithm, it will become an issue for more demanding tasks. It will certainly be necessary for implementing multiple levels of control, where high levels calculate deadlines for lower levels. Fortunately, the basic technique used for learning over other parameters should be equally valid for time. The agent will store the deadlines as an additional component in the points in the case database. The agent will initialize the domain model cases to match any deadline. During performance, the agent will interpolate over time as well as the other variables. This will require an external computation package, as described in the above section. During learning, the agent will specialize the deadline by computing the most general difference, the same as the other parameters. This will cause the agent to behave poorly for different deadlines at first, but gradually build-up experience with varying deadlines and recall cases with relevant deadlines for each new problem.

8.8 Acquiring and Revising the Domain Model

It may be necessary for some reason to investigate control learning without any domain knowledge whatsoever. The problem is impossible without access to sensors and effectors of some kind, but there may be situations where a domain model is unavailable. In these cases, the agent must acquire a domain model by some method. Although there is much previous work on generating a control strategy without using a domain model [55, 23], there is little addressing the problem of finding a qualitative model for a domain. For simple domains like the artificial ones presented here, two observations are all that is necessary to define whether the relationship between the single control and single desired variable is positive or negative. However, for complex domains, it may require many observations to develop a complex and correct qualitative model. Since a qualitative model is conceptually simple and understandable, it may be more effective for an instructor to directly teach the model using natural language. Instructo-Soar [21] is a Soar agent with this capability.

If the agent is able to determine that the domain model is incorrect, it will need to revise itself. IMPROV [44] is a Soar agent with the capability to detect knowledge errors and correct them through experiments. A major future project is the integration of Instructo-Soar, IMPROV, and SPLICE to create an agent that can receive instructions in natural language, correct the knowledge in those instructions, and learn to perform tasks in continuous-valued domains.

The large number and variety of directions for future research show that the problem of symbolic processing in continuous-valued environments is rich with research potential. Perhaps the most promising research area is continuing to apply SPLICE to real-world domains, and letting problems that arise drive development. This would not only lead to interesting and useful SPLICE agents, but also lend insight into the problem of creating domain-independent learning architectures.

CHAPTER 9

Conclusion

I love you, you love me, we're a happy family,
With a great big hug and kiss from me to you,
Won't you say you love me too?

SPLICE is a symbolic architecture for adaptive control in continuous-valued environments. This chapter briefly summarizes SPLICE's motivations, approach, experimental results, and contributions.

9.1 Motivation Summary

The motivational path for SPLICE is as follows:

Control in Continuous-valued Environments This is a significant problem because many real-world domains have the characteristics of these environments.

Adaptive Control Since many of these CV domains are very complex, they are difficult to explicitly model, and the model may became incorrect as the domain dynamics change. This implies that a mechanism to automatically learn control strategies may be more accurate given time in the environment to train.

Symbolic Adaptive Control The advantages to implementing an adaptive controller in a symbolic formalism come from two major points of view.

- **Functional** Symbolic systems can represent knowledge comprehensibly and accept intuitive high-level domain descriptions, which can improve training time in complex environments. Also, symbolic systems naturally handle nominal sensory features, so they are appropriate for hybrid environments.
- **Theoretical** Symbolic controllers can easily integrate with other symbolic capabilities to form more complete autonomous agents. Also, symbolic approaches to CV environments afford the opportunity to test the Physical Symbol System Hypothesis in a broader context.

9.2 Approach Summary

The specific approach of SPLICE is to compute an action in three stages:

1. Convert numerical sensed and desired states to hierarchical sets of regions.
2. Map the initial and desired regions to the most specific stored matching case. If there is no matching case, SPLICE uses a qualitative reasoner to derive an approximate maximally general case.
3. Use the two points in the case to construct a linear model relating control to the initial and final states, and apply to the initial and desired states.

At a specified deadline, SPLICE observes the final state and adds new generalized cases to the case database. SPLICE chooses the level of generalization to maximize the applicability of the new case yet specific enough not interfere with already existing cases where they are correct.

9.3 Results Summary

The performance of SPLICE and the other algorithms tested varied greatly depending on the domain laws and the experimental methodology. In general, algorithms with representations that did not fit the domain laws performed poorly. Algorithms with flexible representations could perform well in any domain with sufficient training. Finally, learning is much slower with random methodologies than repeated methodologies. SPLICE performed well compared to the other algorithms because it learned all the domains using the repeated methodologies, and it generally learned faster than RFWR, the other algorithm capable of learning all the domains. However, SPLICE was unable to learn complex maneuvers in the simulated flight domain because its assumptions about the environment were incorrect.

9.4 Contributions

This thesis makes contributions in both understanding how to apply symbolic processing to continuous-valued environments and the practical problem of control in such domains. Perhaps the major contribution is the overall design and implementation of the first fully symbolic agent for adaptive control in continuous-valued environments. Individual contributions either come from the design of the agent or the results of experimental testing and comparison.

- Contributions from design:
    1. Symbolic systems can function in CV environments given appropriate encoding and decoding functions.

       Besides the static "translation" mappings of quantification into regions and interpolation into controls, all learning and performance computation is concerned with qualitative reasoning and manipulating the case database. One exception is the heuristics for choosing which point to copy into a new case, which is really part of defining the linear model.
    2. Qualitative reasoning provides the top level of abstraction for a response surface for control tasks, and experience with the environment refines the surface.

The qualitative reasoner uses a user-provided domain model to determine the relevant variables and the nature of the relationship between controls and desired variables. This greatly reduces the dimensionality of the problem and speeds up learning.

3. Using symbols to associate problems with local models is efficient.

   SPLICE stores its cases as Soar chunks (rules) with symbolic conditions, and the rete matcher is not sensitive to the number of rules or chunks in memory. This allows match time to remain constant as the agent accumulates cases. In fact, since the chunks are part of an SCA network, more specific cases match sooner than more general cases, so the agent executes faster as its matches become more specific. In an algorithmic complexity sense, SPLICE is $O(1)$ with respect to the number of training examples it experiences. This allows SPLICE to learn incrementally as the agent performs, in contrast with batch algorithms like ID3 that are $O(n)$ at best.

4. Partitioning the space of possible problems into cases of widely varying generality is a novel and compact representation for control and function approximation.

   Although representations like quad-trees and oct-trees [22] are common for spatial occupation, most function approximation algorithms that subdivide the domain space use similar-sized partitions, which is inefficient for some functions. SPLICE begins with the maximally general partition and subdivides based on feedback from the environment. Since SPLICE creates new cases local to where the model from the current case database is incorrect, it bears some resemblance to compression in coding theory [17]. This correspondence may merit further investigation.

5. The learning algorithm is guaranteed to converge.

   Given certain environment restrictions, SPLICE operates much like an iterative Newton-Raphson method. It is guaranteed to converge to within ϵ of a desired value y^* after some number of iterations on the same problem. The generalization ability of SPLICE reduces the number of necessary iterations proportional to the experience the agent has had with similar problems.

- Contributions from testing:

1. Different methodologies make a huge difference in performance.

   In the simplest experiments, SPLICE completed its task perfectly after a few dozen cycles. In the most challenging experiments, SPLICE was still experiencing substantial error after 200 cycles.

2. Generalization and interpolation are key capabilities.

   Although other lesions damaged performance in some methodologies and some domains, losing these two capabilities was devastating to performance in all experiments. In general, the lesion studies isolate individual components and allow independent evaluation of separate parts of the agent. This allows substantiated contributions from individual capabilities in additional to the overall contributions of the agent as a whole.

3. Performance of numerical algorithms is comparable to SPLICE.

SPLICE's learning rate in all experiments and all methodologies is never notably worse than the best numerical algorithm, and SPLICE performs better than any other algorithm for some experiments.

4. Adaptive control algorithms must have very flexible assumptions for success in complex domains.

   SPLICE was able to learn speed goals for taxiing around the runway, but its assumptions about control were incorrect for controlling climb rate. More broadly, implementing any general learning algorithm in a complex dynamic domain must take into account assumptions regarding nondeterminism, computation time, and the general model of control appropriate for this environment. The agent must be flexible enough to modulate these assumptions based on provided initial domain knowledge or direct experience during early exploratory learning.

In the effort to create thinking machines, it will be necessary to combine symbolic and numerical processing to reap the benefits of each. The challenge will be to design an intelligent agent in which both processing styles coexist harmoniously. This thesis is an example of how symbolic and numerical processing can work together to solve a problem.

APPENDICES

APPENDIX A

SPLICE Traces

This appendix presents a trace of SPLICE performing a simple task and gives more detail on the SPLICE implementation. The task is simply achieving a speed of 500 from an initial speed of 0. The methodology is similar to **random-soft** (Chapter 5) with a single goal, because the agent continues working on the goal until it is achieved. The domain is Domain 1 (Chapter 5), the linear domain. The agent solves the task in four control adjustments.

A.1 Explanation of Trace

The execution trace is primarily a sequential list of Soar state and operator decisions, with additional explanatory text. Each decision cycle is numbered and describes the decision made for that cycle. Most decisions are for operators. Operator decisions contain an `O: On`, where `On` is the identifier of the operator, the name of the operator, and optionally some additional operator-specific information. When Soar cannot make a unique operator decision, the agent creates an impasse, depicted by a `==>`, and a new state `S: Sn`. The amount of space between the decision cycle number and the operator or state decision is proportional to the depth of the goal stack at that decision. The agent completes the entire task in 209 decision cycles.

The agent prints notification when it learns chunks and when certain productions fire. These productions are bootstrapping productions that function as pre-learned chunks. Other lines in the trace include communication with the external simulator, goal achievement status, case match status, and the contents of the matched case.

A.2 Control Adjustment 1

SPLICE makes its first control adjustment from a stationary position. First SPLICE checks its case database, finds nothing appropriate (the state no-change), and uses its qualitative model to find the relationship between pedal and speed.

```
     0: ==>S: S1
     1:    O: O1 (^name wait)
** received speed value 0 from simulator **
** received pedal value 0 from simulator **
** Desired speed: 500 **
```

```
Desired 256-511 too low: State = 0-255
     2:    O: O3 (^name sca-action-classify)
     3:    ==>S: S2 (operator no-change)
     4:       O: O4 (^name get-example)
     5:       O: O14 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
     6:       O: O22 (^name switch-context)
Retracting predict-action*propose*switch-context
     7:       O: O23 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
desired count goes from 5 to 4
desired count goes from 4 to 3
desired count goes from 3 to 2
desired count goes from 2 to 1
desired count goes from 1 to 0
     8:       ==>S: S3 (state no-change)
Entering qualitative reasoner.
     9:          O: O35 (^name copy-number desired speed 500)
    10:          O: O34 (^name copy-number initial speed 0
Building chunk-1
    11:          O: O33 (^name copy-number initial pedal 0)
Building chunk-2
    12:          O: O32 (^name controls)
Setting pedal to 63.5.
Positive pedal --> Positive speed
Variable speed has been achieved.
    13:          O: E2 (^name evaluate-state)
Building chunk-3
    14:       O: S7 (prediction)

Match: desired -2048-2047, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (-2048, 127, 2047)

Interpolate: (-128, -4095), (127, 4095) at 500: 15

    15:       O: O39 (^name return-result)
Building chunk-4
Building chunk-5
```

The `abstract` operator increases the generality of the current situation until it matches a case. Unlike the original SCA implementation, generalization does not necessarily follow a prescribed ordering, but matches any collection of regions at the given level of generality. This is an enhancement to SCA most useful for classifying multiple objects presented in

IMPROV [44], another inductive learning system in Soar. The qualitative reasoner needs to explicitly copy numbers to avoid the data-chunking problem, as described in Appendix B. Chunks 1 and 2 identify speed and pedal variables as relevant for future learning. The reasoner determines that the relationship between the pedal and speed is positive, and creates an initial maximally-general case in `chunk-3`. The prediction operator creates a linear model from the domain points and solves for the desired speed to predict a pedal adjustment of 15. Chunks 4 and 5 return this information to the top state and never match again.

After the deadline, the agent learns a FSAC and a GDC.

```
    16:    O: O1 (^name wait)
** received speed value 15 from simulator **
** received pedal value 15 from simulator **
    17:    O: O1 (^name wait)
** received speed value 30 from simulator **
    18:    O: O1 (^name wait)
** received speed value 45 from simulator **
    19:    O: O1 (^name wait)
** received speed value 60 from simulator **
    20:    O: O1 (^name wait)
** received speed value 75 from simulator **
    21:    O: O1 (^name wait)
** received speed value 90 from simulator **
    22:    O: O1 (^name wait)
** received speed value 105 from simulator **
    23:    O: O1 (^name wait)
** received speed value 120 from simulator **
    24:    O: O1 (^name wait)
** received speed value 135 from simulator **
    25:    O: O1 (^name wait)
** received speed value 150 from simulator **
    26:    O: O41 (^name deadline)
Desired 256-511 too low: State = 0-255
    27:    O: O42 (^name sca-learn-action FSAC)
    28:    ==>S: S11 (operator no-change)
    29:       O: O43 (^name get-example)
    30:       O: O54 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
    31:       O: O62 (^name switch-context)
Retracting predict-action*propose*switch-context
    32:       O: O63 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
```

```
desired count goes from 5 to 4
desired count goes from 4 to 3
desired count goes from 3 to 2
desired count goes from 2 to 1
desired count goes from 1 to 0
Firing chunk-1
Firing chunk-3
Firing chunk-2
    33:       O: O71 (prediction)

Match: desired -2048-2047, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (-2048, 127, 2047)

Match Problem: (500, 0)
Current Result: (150, 0)
MGD: desired 0-255

    34:       O: O75 (^name select-feature desired speed 0-255)
    35:       O: O90 (^name replace)
desired count goes from 0 to 4
Retracting chunk-3
Retracting chunk-1
Retracting chunk-2
    36:       ==>S: S12 (state no-change)
    37:          O: O97 (^name copy-number final pedal 15)
    38:          O: O96 (^name copy-number final speed 150)
    39:          O: O95 (^name copy-number initial speed 0)
    40:          O: O94 (^name copy-number initial pedal 0)
    41:          O: O93 (^name copy-number desired speed 150)
    42:          O: O92 (abstract)
desired count goes from 4 to 3
desired count goes from 3 to 2
desired count goes from 2 to 1
desired count goes from 1 to 0
Firing chunk-1
Firing chunk-3
Firing chunk-2
    43:          O: O106 (prediction)

Match: desired -2048-2047, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (-2048, 127, 2047)

    44:          O: O108 (^name distance)
    45:          O: O110 (^name delete-point P2)
    46:          O: O107 (^name return-result)

Learn (150, 0): desired 0-255, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (0, 15, 150)

Building chunk-6
Building chunk-7
    47:       O: O88 (^name return-result)
Building chunk-8
```

As a simplification, the environment only changes between the time the agent sends a control and the deadline. This simplification is not used in the experiments with the flight simulator (Chapter 7). At the deadline, the goal is still not achieved at the 256–511 level of generality. The agent first learns the FSAC. It finds that the most general difference between the matched problem and the current result is 0–255 for the desired speed. It deletes point `P2` and creates a new case with `P1` and the current point. `Chunk-6` stores the new case and chunks 7 and 8 terminate the subgoals and never match again. Note that `chunk-6` still contains one domain model point, so it is only partially correct.

```
    48:    O: O111 (^name sca-learn-action GDC)
    49:    ==>S: S20 (operator no-change)
    50:       O: O112 (^name get-example)
    51:       O: O123 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
    52:       O: O131 (^name switch-context)
Retracting predict-action*propose*switch-context
    53:       O: O132 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
desired count goes from 5 to 4
desired count goes from 4 to 3
desired count goes from 3 to 2
desired count goes from 2 to 1
desired count goes from 1 to 0
Firing chunk-1
Firing chunk-3
Firing chunk-2
    54:       O: O140 (prediction)

Match: desired -2048-2047, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (-2048, 127, 2047)
No monotonicity violation.

Match Problem: (500, 0)
Current Result: (150, 0)

    55:       O: O144 (^name select-feature desired speed 256-511)
    56:       O: O159 (^name replace)
desired count goes from 0 to 4
Retracting chunk-3
Retracting chunk-1
Retracting chunk-2
    57:       ==>S: S21 (state no-change)
```

```
    58:          O: O166 (^name copy-number final pedal 15)
    59:          O: O165 (^name copy-number final speed 150)
    60:          O: O164 (^name copy-number initial speed 0)
    61:          O: O163 (^name copy-number initial pedal 0)
    62:          O: O162 (^name copy-number desired speed 500)
    63:          O: O161 (abstract)
desired count goes from 4 to 3
desired count goes from 3 to 2
desired count goes from 2 to 1
desired count goes from 1 to 0
Firing chunk-1
Firing chunk-3
Firing chunk-2
    64:          O: O175 (prediction)

Match: desired -2048-2047, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (-2048, 127, 2047)

    65:          O: O177 (^name distance)
    66:          O: O179 (^name delete-point P2)
    67:          O: O176 (^name return-result)

Learn (500, 0): desired 256-511,-2048-2047
Points: P1 = (2047, -128, -2048), P2 = (0, 15, 150)

Building chunk-9
Building chunk-10
    68:       O: O157 (^name return-result)
Building chunk-11
    69:    O: O180 (^name reset)
```

The agent does not detect a monotonicity violation for the goal-directed chunk, so it specializes the desired speed at the most general failure, 256–511. **Chunk-9** stores the new case and chunks 10 and 11 terminate the subgoals and never match again. Note that chunks 6 and 9 have identical points and are adjacent, so a pruning algorithm may be able to merge these cases. However, the cases are formed around different desired speeds, so it is not clear what the desired speed for the merged case should be. Also, the merged case will be more general so it will match more slowly. After learning the GDC, the agent resets itself in preparation for the next cycle.

A.3 Control Adjustment 2

In this cycle, the agent matches a partially-correct case, and creates a single completely-correct case.

```
    70:    O: O1 (^name wait)
    71:    O: O182 (^name sca-action-classify)
    72:    ==>S: S29 (operator no-change)
    73:       O: O183 (^name get-example)
    74:       O: O194 (abstract)
```

```
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
    75:       O: O202 (^name switch-context)
Retracting predict-action*propose*switch-context
    76:       O: O203 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
desired count goes from 5 to 4
Firing chunk-10
    77:       O: O207 (prediction)

Match: desired 256-511,-2048-2047
Points: P1 = (2047, -128, -2048), P2 = (0, 15, 150)

Interpolate: (-128, -3945), (15, 300) at 500: 22

    78:       O: O209 (^name return-result)
Building chunk-12
Building chunk-13
```

The agent waits one decision to initialize itself, and generalizes the current problem until it matches the goal-directed case from the last cycle. The new linear model predicts a control of 22 will achieve the goal of 500 from its initial state of 150. Chunks 12 and 13 return the matching case information to the top state and never match again.

```
    79:    O: O1 (^name wait)
** received speed value 172 from simulator **
** received pedal value 22 from simulator **
    80:    O: O1 (^name wait)
** received speed value 194 from simulator **
    81:    O: O1 (^name wait)
** received speed value 216 from simulator **
    82:    O: O1 (^name wait)
** received speed value 238 from simulator **
    83:    O: O1 (^name wait)
** received speed value 260 from simulator **
    84:    O: O1 (^name wait)
** received speed value 282 from simulator **
    85:    O: O1 (^name wait)
** received speed value 304 from simulator **
    86:    O: O1 (^name wait)
** received speed value 326 from simulator **
    87:    O: O1 (^name wait)
** received speed value 348 from simulator **
```

```
    88:    O: O1 (^name wait)
** received speed value 370 from simulator **
    89:    O: O211 (^name deadline)
Desired 384-511 too low: State 256-383
    90:    O: O212 (^name sca-learn-action FSAC)
    91:    ==>S: S33 (operator no-change)
    92:       O: O213 (^name get-example)
    93:       O: O224 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
    94:       O: O232 (^name switch-context)
Retracting predict-action*propose*switch-context
    95:       O: O233 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
desired count goes from 5 to 4
Firing chunk-10
    96:       O: O237 (prediction)

Match: desired 256-511, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (0, 15, 150)

    97:       ==>S: S34 (state no-change)
Building chunk-14
Building chunk-15
Match Problem: (500, 0)
Current Result: (370, 150)
MGD: desired (256-383)
MGD: initial (128-255)

    98:       O: O251 (^name select-feature initial speed 128-255)
Firing predict-action*propose*switch-context
Retracting chunk-15
Retracting chunk-14
    99:       O: O259 (^name test-if-matched)
   100:       O: O260 (^name replace)
initial count goes from 0 to 5
Retracting chunk-10
Retracting predict-action*propose*switch-context
   101:       ==>S: S35 (state no-change)
   102:          O: O271 (^name copy-number final pedal 22)
   103:          O: O270 (^name copy-number final speed 370)
   104:          O: O269 (^name copy-number initial speed 150)
   105:          O: O268 (^name copy-number initial pedal 15)
   106:          O: O267 (^name copy-number desired speed 370)
   107:          O: O266 (abstract)
```

```
initial count goes from 5 to 4
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
Firing chunk-10
   108:         O: O281 (^name switch-context)
Retracting predict-action*propose*switch-context
Building chunk-16
   109:         O: O282 (prediction)

Match: desired 256-511, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (0, 15, 150)

   110:         O: O285 (^name distance)
   111:         O: O286 (^name delete-point P1)
   112:         O: O284 (^name return-result)

Learn (370, 150): desired 256-511, initial 128-255
Points: P1 = (0, 15, 150), P2 = (150, 22, 370)

Building chunk-17
Building chunk-18
   113:      O: O255 (^name return-result)
Building chunk-19
```

The goal is again too low at the 384–511 level. Since chunks 1 and 2 generalize incorrectly, the agent briefly impasses to relearn the relevant variables in chunks 14 and 15. The agent learns a FSAC specializing the initial speed range 128–255. Since the most general differences for the desired and initial speeds were at the same level of generality, the agent randomly chose one. Since the agent is specializing an initial speed for the first time, `chunk-16` proposes changing the context earlier for these initial and desired speeds. Next the agent removes the last domain model point and learns a correct case in `chunk-17`. Again, chunks 18 and 19 terminate the subgoals.

```
   114:    O: O288 (^name sca-learn-action GDC)
   115:    ==>S: S44 (operator no-change)
   116:       O: O289 (^name get-example)
   117:       O: O302 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
Firing chunk-16

   118:      O: O305 (^name switch-context)
Retracting chunk-16
   119:      O: O307 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
```

```
desired count goes from 5 to 4
Firing chunk-18
   120:       O: O311 (prediction)

Match: desired 256-511, initial 128-255
Points: P1 = (0, 15, 150), P2 = (150, 22, 370)
(150, 22, 370) matches current point.

Firing chunk-14
Firing chunk-15
   121:       O: O329 (^name return-result)
Building chunk-20
   122:    O: O331 (^name reset)
```

Since 256–511 matches the goal, the FSAC and the GDC reduce to the same case, so the agent does not need to explicitly learns a GDC. `Chunk-20` notifies the top state that the subgoal is finished and never matches again.

A.4 Control Adjustment 3

Unfortunately, the new initial speed for this cycle does not match the initial speed of 128–255 in the correct case learned in the previous cycle, so the linear model is still partially based on a domain model point, resulting in a slightly erroneous control.

```
   123:    O: O1 (^name wait)
   124:    O: O333 (^name sca-action-classify)
   125:    ==>S: S48 (operator no-change)
   126:       O: O334 (^name get-example)
   127:       O: O345 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
   128:       O: O353 (^name switch-context)
Retracting predict-action*propose*switch-context
   129:       O: O354 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
desired count goes from 5 to 4
Firing chunk-10
   130:       O: O358 (prediction)

Match: desired 256-511, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (0, 15, 150)
```

```
Interpolate: (-128, -3725), (15, 520) at 500: 14

Firing chunk-14
Firing chunk-15

   131:       O: O360 (^name return-result)
Building chunk-21
Building chunk-22
```

The agent again matches the chunk learned in the first cycle. If the agent had randomly chosen to specialize the desired speed in decision cycle 98 instead of the initial speed, the GDC for that cycle would have matched, and the linear model for this cycle would be correct. As usual, chunks 21 and 22 terminate the subgoal and never match again.

```
132:    O: O1 (^name wait)
** received speed value 384 from simulator **
** received pedal value 14 from simulator **
   133:    O: O1 (^name wait)
** received speed value 398 from simulator **
   134:    O: O1 (^name wait)
** received speed value 412 from simulator **
   135:    O: O1 (^name wait)
** received speed value 426 from simulator **
   136:    O: O1 (^name wait)
** received speed value 440 from simulator **
   137:    O: O1 (^name wait)
** received speed value 454 from simulator **
   138:    O: O1 (^name wait)
** received speed value 468 from simulator **
   139:    O: O1 (^name wait)
** received speed value 482 from simulator **
   140:    O: O1 (^name wait)
** received speed value 496 from simulator **
   141:    O: O1 (^name wait)
** received speed value 510 from simulator **
   142:    O: O362 (^name deadline)
Desired 496-511 achieved.
   143:    O: O363 (^name sca-learn-action FSAC)
   144:    ==>S: S52 (operator no-change)
   145:       O: O364 (^name get-example)
   146:       O: O375 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
   147:       O: O383 (^name switch-context)
Retracting predict-action*propose*switch-context
```

```
   148:       O: O384 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
desired count goes from 5 to 4
Firing chunk-10
   149:       O: O388 (prediction)

Match: desired 256-511, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (0, 15, 150)

Firing chunk-14
Firing chunk-15
Match Problem: (500, 0)
Current Result: (510, 370)
MGD: initial (256-511)

   150:       O: O401 (^name select-feature initial speed 256-511)
Firing predict-action*propose*switch-context
Retracting chunk-15
Retracting chunk-14
   151:       O: O409 (^name test-if-matched)
   152:       O: O410 (^name replace)
initial count goes from 0 to 4
Retracting chunk-10
Retracting predict-action*propose*switch-context
   153:       ==>S: S53 (state no-change)
   154:          O: O421 (^name copy-number final pedal 14)
   155:          O: O420 (^name copy-number final speed 510)
   156:          O: O419 (^name copy-number initial speed 370)
   157:          O: O418 (^name copy-number initial pedal 22)
   158:          O: O417 (^name copy-number desired speed 510)
   159:          O: O416 (abstract)
initial count goes from 4 to 3
initial count goes from 3 to 2
initial count goes from 2 to 1
initial count goes from 1 to 0
Firing predict-action*propose*switch-context
Firing chunk-10
   160:          O: O430 (^name switch-context)
Retracting predict-action*propose*switch-context
Building chunk-23
   161:          O: O431 (prediction)

Match: desired 256-511, initial -2048-2047
Points: P1 = (2047, -128, -2048), P2 = (0, 15, 150)

   162:          O: O434 (^name distance)
   163:          O: O435 (^name delete-point P1)
   164:          O: O433 (^name return-result)

Learn (510, 370): desired 256-511, initial 256-511
Points: P1 = (370, 14, 510), P2 = (0, 15, 150)
```

```
Building chunk-24
Building chunk-25
   165:       O: O406 (^name return-result)
Building chunk-26
   166:    O: O437 (^name reset)
```

Since this control adjustment achieved the desired speed to the maximally-specific region (496–511), the agent only learns an FSAC for this cycle. The only discernible difference is in the initial speed, so the agent learns a new case for the initial speed range of 256–511. Chunks 23 and 24 notifies the agent to change context and stores the case, respectively. Chunks 25 and 26 terminate the subgoals.

A.5 Control Adjustment 4

Although the agent would normally terminate when it achieves the maximally-specific region of the goal, this trace includes a final cycle to show how the agent achieves the exact desired value.

```
   167:    O: O1 (^name wait)
   168:    O: O439 (^name sca-action-classify)
   169:    ==>S: S62 (operator no-change)
   170:       O: O440 (^name get-example)
   171:       O: O453 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
Firing chunk-23
   172:       O: O457 (^name switch-context)
Retracting chunk-23
   173:       O: O459 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
desired count goes from 5 to 4
Firing chunk-25
   174:       O: O463 (prediction)

Match: desired 256-511, initial 256-511
Points: P1 = (370, 14, 510), P2 = (0, 15, 150)

Interpolate: (14, 650), (15, 660) at 500: -1

Firing chunk-14
Firing chunk-15

   175:       O: O465 (^name return-result)
Building chunk-27
Building chunk-28
   176:    O: O1 (^name wait)
** received speed value 509 from simulator **
```

```
** received pedal value -1 from simulator **
   177:    O: O1 (^name wait)
** received speed value 508 from simulator **
   178:    O: O1 (^name wait)
** received speed value 507 from simulator **
   179:    O: O1 (^name wait)
** received speed value 506 from simulator **
   180:    O: O1 (^name wait)
** received speed value 505 from simulator **
   181:    O: O1 (^name wait)
** received speed value 504 from simulator **
   182:    O: O1 (^name wait)
** received speed value 503 from simulator **
   183:    O: O1 (^name wait)
** received speed value 502 from simulator **
   184:    O: O1 (^name wait)
** received speed value 501 from simulator **
   185:    O: O1 (^name wait)
** received speed value 500 from simulator **
   186:    O: O467 (^name deadline)
Desired 496-511 achieved.
   187:    O: O468 (^name sca-learn-action FSAC)
   188:    ==>S: S66 (operator no-change)
   189:       O: O469 (^name get-example)
   190:       O: O482 (abstract)
initial count goes from 8 to 7
initial count goes from 7 to 6
initial count goes from 6 to 5
initial count goes from 5 to 4
Firing chunk-23
   191:       O: O486 (^name switch-context)
Retracting chunk-23
   192:       O: O488 (abstract)
desired count goes from 8 to 7
desired count goes from 7 to 6
desired count goes from 6 to 5
desired count goes from 5 to 4
Firing chunk-25
   193:       O: O492 (prediction)

Match: desired 256-511, initial 256-511
Points: P1 = (370, 14, 510), P2 = (0, 15, 150)

Firing chunk-14
Firing chunk-15
Match Problem: (510, 370)
Current Result: (500, 510)
MGD: initial (384-511)

   194:       O: O506 (^name select-feature initial speed 384-511)
Firing chunk-23
Retracting chunk-15
Retracting chunk-14
   195:       O: O516 (^name test-if-matched)
```

```
   196:       O: O517 (^name replace)
initial count goes from 4 to 5
Retracting chunk-25
Retracting chunk-23
   197:       ==>S: S67 (state no-change)
   198:          O: O532 (^name copy-number final pedal -1)
   199:          O: O531 (^name copy-number final speed 500)
   200:          O: O530 (^name copy-number initial speed 510)
   201:          O: O529 (^name copy-number initial pedal 14)
   202:          O: O528 (^name copy-number desired speed 500)
   203:          O: O527 (abstract)
initial count goes from 5 to 4
Firing chunk-23
Firing chunk-25
   204:          O: O538 (^name switch-context)
Retracting chunk-23
Building chunk-29
   205:          O: O539 (prediction)

Match: desired 256-511, initial 256-511
Points: P1 = (370, 14, 510), P2 = (0, 15, 150)

   206:          O: O543 (^name distance)
   207:          O: O544 (^name delete-point P2)
   208:          O: O542 (^name return-result)

Learn (500, 510): desired 256-511, initial 384-511
Points: P1 = (370, 14, 510), P2 = (510, -1, 500)

Building chunk-30
Building chunk-31
   209:       O: O510 (^name return-result)
Building chunk-32
Task finished.
```

The agent matches the case with a correct model, computes the control, and achieves the exact desired value. It again learns an FSAC for the more specific initial speed range 384–511. It chooses to retain P1 over P2 because it is closer to the problem (500, 510). Note that matching time decreases from 16 generalizations in the first cycle to 8 generalizations in the fourth cycle.

A.6 Final Knowledge State

Figure A.1 displays the final knowledge state of the agent. Besides the maximally-general domain model case, the agent has learned five cases. Two cases are maximally general along the initial speed axis and include one experienced point and one domain model point, and three cases are specialized along both dimensions and include two experienced points. The relationship between points and cases is in Table A.1. Since the domain is linear, the model for any two experienced points is correct for this domain. This means that any problem with a desired speed of 256–511 and an initial speed of 128–511 would be solved with a single control adjustment. Since SPLICE makes no global assumptions about the nature

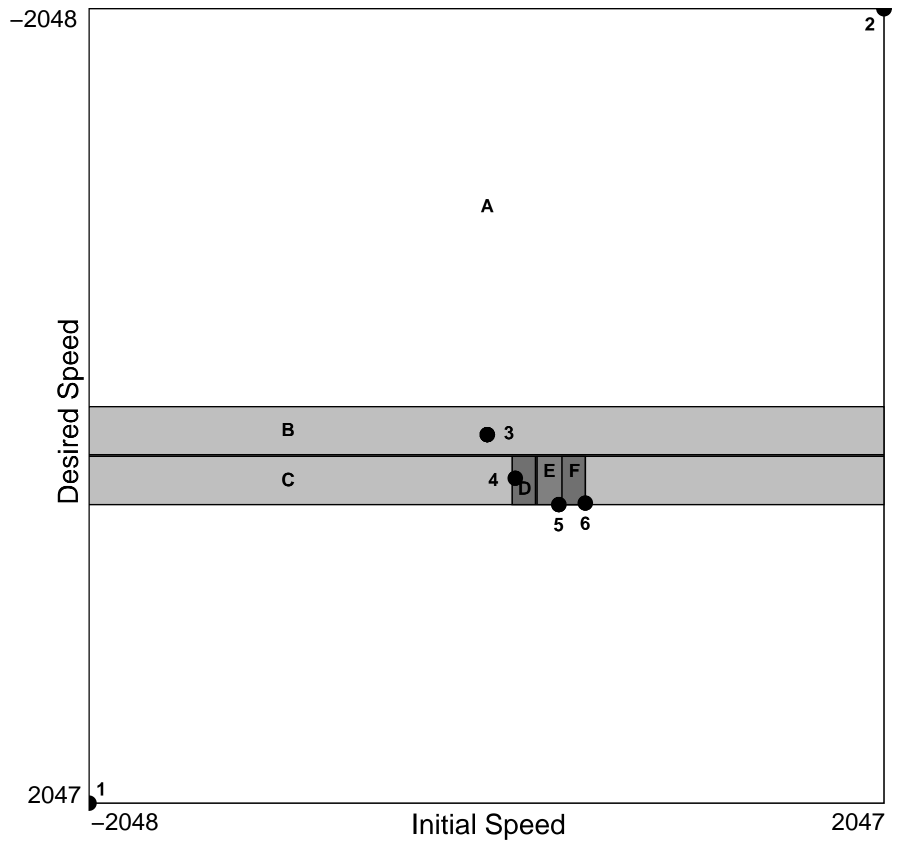

Figure A.1: Final case database for the trace.

of the domain, it does not assume the experienced points are relevant in other areas of the problem space and requires further training.

Case (parent)	Most Specific		P_1	P_2
	Initial	Desired		
A (domain model)	-2048–2047	-2048–2047	1	2
$B(A)$	-2048–2047	0–255	2	3
$C(A)$	-2048–2047	256–511	2	3
$D(C)$	128–255	256–511	3	4
$E(C)$	256–511	256–511	3	5
$F(E)$	384–255	256–511	5	6

Table A.1: Final case database for the trace.

APPENDIX B

Data-chunking Numbers in Soar

This appendix introduces the *data-chunking problem*, describes its unique manifestation in this work, and presents the solution which may be helpful for other Soar researchers working in domains with large external states. Further information about data-chunking is in *Unified Theories of Cognition* [38], for example. The on-line Soar Tutorial [19] contains additional practical information, including a version of this Appendix and code.

B.1 Data-chunking in General

A persistent challenge to Soar agents is the need to remember specific symbols for later use. For example, an agent may want to remember the color of a house so it can recognize it again later. Since Soar represents long-term memory with rules, one would want a rule like:

```
IF <x> is RED
THEN <x> is MY-HOUSE
```

Since chunking is the only way to add rules to memory, the agent needs to create a chunk that matches with the working memory element "`<x> is RED`" and creates the working memory element "`<x> is MY-HOUSE`." Unfortunately, the chunking mechanism includes all working memory elements tested in the subgoal as conditions of the chunk. Since `MY-HOUSE` must be tested for the subgoal to know the relationship, Soar generates the following tautological but unproductive chunk:

```
IF <x> is RED
AND <x> is MY-HOUSE
THEN <x> is MY-HOUSE
```

Since the agent is trying to remember a fact, this is called the *data- chunking problem.* The standard work-around is to use the loophole that working memory tested for search control does not appear in the chunk, under the assumption that search control (preferences) only affects efficiency not accuracy. With this technique, the subgoal *generates* all possible

values for `<x>`, and uses preferences to select the correct value from the supergoal. In this way, the chunking mechanism does not include "`<x> is MY-HOUSE`," so the chunk is correct.

Since exploiting search control is not a theoretically satisfying solution, and since chunks may include search control in future versions of Soar, Rick Lewis and others have developed a new approach, as documented Doug Pearson's web site [42]. Although this thesis used the normal data-chunking approach, the problems and solutions in the next section are equally valid for data chunking without search control.

B.2 Data-chunking Numbers

SPLICE needs to recall numbers for a variety of purposes. The most important are comparing previous experiences to new experiences to detect differences, and interpolating over previous experiences to create local linear models. This requires data-chunking, but data-chunking includes additional complexities for continuous-valued environmental features. The problem is that it is impossible to generate every element of an infinite set like the integers, and impractical to generate every value of a many-valued set like the ranges of integers considered in this thesis.

A possible way to reduce the cost of generation for countable sets is to sequentially enumerate the set and terminate when the value matches. This has the disadvantage of being slow compared to parallel generation in normal data-chunking, and it fails to work for uncountable sets like the real numbers. A better solution is to convert the number to a binary representation and data- chunk that one bit at a time. Since the numbers considered in this thesis were all integers ranging from -2^n to 2^n-1, the binary representation is simply adding an offset of 2^n to the quantity, similar to the excess representation commonly used for exponents in floating point numbers.

To copy a number to a subgoal without creating a data dependency to the number in the supergoal, SPLICE uses an operator named `copy-number`. This operator first temporarily stores the number in binary in the subgoal. Then the operator generates the possible values for each digit, 0 and 1, and selects the correct digit using search control. After the operator fills in all the digits, it recomputes the decimal representation for use in chunks. Since the only connection to the number in the supergoal is through search control, chunks do not refer to the number. Since the operator sequentially generates two values for each digit, its space complexity is $O(1)$ and its time complexity is $O(n)$, where n is the number of digits. Traditional parallel generation is $O(2^n)$ in space and $O(1)$ in time. Sequential enumeration is $O(1)$ in space and $O(2^n)$ in time.

Note that it is possible to represent the number using a string of digits with a different base. With a representation of n digits of base b, the operator can represent b^n distinct values. The `copy-number` operator will generate b possible values for each of n digits, so for approximately equal values of b^n, larger bases require more space but less time.

B.3 Factoring in Data-chunking

The concept of trading off space and time in data-chunking is applicable for any Soar application where the space requirements of generating all possible values at once is a concern. For example, even generating a symbolic state with several possible values for several features is exponential in state space. It is natural to factor the state by features to make generation linear in the number of features, but this may not be appropriate for

precise control of generation time or space requirements. For the most flexible control, it is possible to convert a state description to a string of digits, copy it to the subgoal, and reconvert into the original description. Since the total number of possible descriptions, T, is known, the digit string must allow at least $b^n \geq T$ possible values. Choosing the appropriate b allows the researcher to precisely control generation space and time tradeoffs. Even if this level of control is not necessary, it is always helpful to consider factoring when execution performance is an issue during generation.

APPENDIX C

Soar Production Templates

One disadvantage of Soar is that Soar productions are not a particularly compact knowledge representation. Every rule related to a data structure must explicitly match this structure. This leads to large repetitive parts of rules. Even worse, when a data structure changes, every rule referencing the data structure must be changed. Also, the limited number of knowledge types defined by the Problem Space Computational Model [38] implies that productions of the same type share some structure. Creating a more compact knowledge representation that can easily adjust to new data structures has been a common factor in improving Soar's usability. The integration of Soar with the Tcl [40] macro language gives some additional flexibility in defining knowledge. This appendix will present some past approaches to adding another layer of knowledge representation and introduce the approach used in this thesis.

C.1 Off-line Approaches

One set of approaches defines a separate knowledge representation and "compiles" it into Soar productions. The productions are then loaded and matched normally. In general, the problem with these approaches is that the new knowledge representation is normally not flexible enough to completely specify any Soar system, so it is still necessary to edit raw Soar productions. These techniques are frequently used in conjunction with standard Soar programming for highly repetitive and common sets of rules.

TAQL The most ambitious and comprehensive approach is TAQL [60] (Task AcQuisition Language). TAQL was intended to be a knowledge-level design tool with a compiler for reduction to the problem-space level. Changing data structures only requires changing the high-level representation and re-compiling for a new Soar system. It greatly reduced development time for tasks appropriate for its representation language, but proved to require too much editing of raw production rules for much time reduction for general Soar programming.

SDE A general effort for Soar editing and debugging, SDE [20] (Soar Development Environment) included a flexible function for interactive creation of Soar rules from templates. The user would define a template of one or more partially-instantiated Soar productions. When invoking the template during editing, SDE prompts the user for variable substitutions and instantiates the resulting template in an Emacs buffer. The high-level representation is the set of templates, and changing the data structure

only requires changing the templates. Unfortunately, since instantiating the templates is interactive, the user must manually recompile the productions by re-invoking the templates.

Free-form generation Since Soar production files are just text files, any programming language with file output can generate sets of productions suitable for the Soar interpreter. Most useful for large sets of rules with regular, small variations, the program or shell script itself defines the representation, and changing the representation requires changing the program and regenerating the rules.

C.2 On-line Approaches

A different solution is to actively interpret high-level representations during execution. The simplest way to do this is define data structure abbreviations [56] similar to regular Soar attributes. Instead of directly matching the abbreviation, the interpreter matches against the abbreviation expansion. Changing the data structure only requires changing the definition of the abbreviations. This can be either implemented in the architecture or knowledge. The architectural approach is to extend the rete matcher to accept abbreviations. This may interfere with efficiency. The knowledge approach is to define the abbreviations as elaborations. This may induce race conditions because the abbreviation is updated one elaboration cycle after the actual data structure. Also, it only applies to matching since changing the abbreviation does not change the data structure.

C.3 Templates in Tcl

The integration of Soar with Tcl allows much more flexibility in Soar development. Soar productions can now call arbitrary Tcl functions from the right-hand side, and Tcl code can control Soar execution and provide input to Soar. Defining a Soar production is now a Tcl procedure that accepts variable substitution, so it can now be automated through Tcl control. The approach advocated by Doug Pearson [41] is to create partially-instantiated productions in files. The Tcl procedure loading productions would set Tcl variables and load a template file, so the template is instantiated with the current variable values. This is "load-time" template expansion because the instantiated productions are created as they are being loaded. This approach is similar to SDE templates, except the instantiated productions never exist as text files. Although this saves file space, there is never an opportunity to edit the productions, so the templates must be complete.

The Soar system developed for this thesis had a need for a higher-level knowledge representation because the region-mapping productions, as described in Chapter 3, were "prechunked." This means that SPLICE had to include rules for every possible region at every level of generality for every sensed variable. Especially for the simulated flight domain, this was a large number of productions that would take up substantial file space. Additionally, since the number of variables and their levels of generality changed often during development, it was helpful to regenerate the rules automatically whenever SPLICE loaded. Finally, these productions were good candidates for templates because they were identical except for the name of the variable and the region interval. An alternative to templates is to create the productions on demand through chunking, as described in Chapter 8.

SPLICE uses a generalization of Tcl template files to generate its region-mapping pro-

ductions. Instead of files, each template is a Tcl procedure, and the variables are Tcl procedure arguments. This has several advantages over files. First, a template is instantiated with a single command and arguments. Second, several templates can be defined in one file, or even the same file as other regular Soar productions. Third, the procedure mechanism allows optional arguments and default values. However, using Tcl procedures for templates has the same drawbacks as Tcl files. This was not a problem for their application in this thesis, but the problem of finding a general mechanism for high-level creation and editing of Soar systems remains.

APPENDIX D

Glossary

Action/Control/Effector How the agent influences the environment. *Action* is a general term describing any effect the agent can have on the environment. *Control* is the action of setting a continuous-valued variable to some value. An *effector* is the name of a variable that is adjustable by the agent.

Agent A computational entity that interacts with the environment. The agent can receive sensory information, process it, and take action. The agent attempts to achieve explicit or implicit goals in the environment.

Continuous Mathematically, a continuous set is a totally ordered set where there is always an element in the order between two other elements. The real numbers are continuous. Time is continuous because conceptually there is always a moment between any two other moments.

Continuous-Valued (CV) A continuous-like set. These sets have an ordering and many elements, but not necessarily arbitrarily small differences between elements. Also, CV environments are dynamic environments where some or all variables, including controls, are continuous-valued.

Cycle One iteration of SPLICE through receiving the current state, calculating a response, waiting for the deadline, and learning from the result.

Deadline The amount of time the agent has to achieve a goal after taking an action.

Domain A well-defined sphere of activity.

Dynamic An environment that constantly changes, even while the agent is computing a response. Chess is an example of a static environment.

Environment The source of percepts for the agent and target of actions. An environment is typically the instantiation of a domain. For example, simulated flight is a domain and the SGI flight simulator is an environment.

Final State The state of the environment at the end of a cycle.

FSAC False sense of accomplishment case. At the deadline, the agent knows how to solve the problem of achieving the final state from the initial state. The agent records this in case it ever experiences a similar problem, by replacing the desired state with the final state and learning a case.

GDC Goal-directed case. If the agent does not achieve its goal, it needs to learn from its failures in order to do better in future similar problems. The agent specializes the GDC to exactly the most general level of specificity not achieved by the deadline.

Goal A desirable situation. In SPLICE, a goal is a desired value for some variable. This value is generalized to the most specific region containing this value.

Initial State The state of the environment at the beginning of a cycle.

Learning Improving performance. When the agent evaluates its performance, it can attempt to modify itself so that it behaves differently, and hopefully better, when it experiences similar situations in the future.

Methodology The procedure followed when conducting an experiment. The experimenter can design methodologies to be challenging or not, realistic or not, and a number of other properties. In this thesis, the methodology defines the sequence of initial and desired states the agent is intended to solve, and the stopping criterion.

Monotonic A function whose first derivative never changes sign. Positive monotonic functions always have a positive or zero slope; negative monotonic functions always have a negative or zero slope. An example of a monotonic function is the level of water in an irregular container. As the volume increases, the level must increase, but the rate of increase may be highly variable.

Performance Taking action in a domain. Performance should be relevant to the agent's current state and goals. The agent can evaluate performance by comparing the result of the action to the expected result. In the case of this thesis, the expected result is generally achieving the goal.

Problem The combination of an initial state and a desired state.

Response Surface Similar to a Universal Plan in nominal environments, it is a mapping defining an action for any combination of initial and desired states. For continuous-valued environments, this mapping corresponds to a geometric surface.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, January 1991.

[2] Hiroshi Akima. On estimating partial derivatives for bivariate interpolation of scattered data. *Rocky Mountain Journal of Mathematics*, 14(1), Winter 1984.

[3] L. E. Baum and T. Petrie. Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics*, 41, 1966.

[4] Richard Ernest Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[5] Scott Benson and Nils Nilsson. *Machine Intelligence*, volume 14, chapter Reacting, Planning and Learning in an Autonomous Agent. The Clarendon Press, Oxford, 1995.

[6] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth, Monterey, CA, 1984.

[7] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[8] Y. Le Cun, L. D. Jackel, B. Boser, and J. S. Denker. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989.

[9] Carl DeBoor. *A Practical Guide to Splines.* Applied mathematical sciences. Springer-Verlag, New York, 1978.

[10] Gerald F. DeJong. Learning to plan in continuous domains. *Artificial Intelligence*, 65(1):71–141, January 1994.

[11] Gerald F. DeJong and Raymond J. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[12] Robert B. Doorenbos. Matching 100,000 learned rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 290–296, 1993.

[13] Richard C. Dorf. *Modern Control Systems.* Addison-Wesley, 5th edition edition, January 1989.

[14] Usama M. Fayyad. *On the Induction of Decision Trees for Multiple Concept Learning.* PhD thesis, The University of Michigan, 1991.

[15] Kenneth D. Forbus, Paul E. Nielsen, and Boi Faltings. Qualitative spatial reasoning: The CLOCK project. *Artificial Intelligence*, 51(1–3):417–471, October 1991.

[16] K. Furuta, T. Ochiai, and N. Ono. Attitude control of a triple inverted pendulum. *International Journal of Control*, 39(6):1351–1365, 1984.

[17] Robert Gray. *Source Coding Theory.* The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, Boston, MA, 1990.

[18] Peter E. Hart. The condensed nearest neighbor rule. *Transactions on Information Theory*, IT-14:515–516, 1967.

[19] Peter M. Hastings. The Soar coloring book: A tutorial for soar. `http://ai.eecs.umich.edu/soar/tutorial`, 1996.

[20] Michael Hucka. The Soar development environment. Unpublished manuscript, April 1994.

[21] Scott B. Huffman. *Instructable autonomous agents.* PhD thesis, University of Michigan, Dept. of Electrical Engineering and Computer Science, 1993. Forthcoming.

[22] Ramesh Jain, Rangachar Kasturi, and Brian Schunck. *Machine Vision.* McGraw-Hill, New York, NY, 1995.

[23] Todd Jochem and Dean Pomerleau. Life in the fast lane: the evolution of an adaptive vehicle control system. *AI Magazine*, 17(2):11–50, Summer 1996.

[24] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Jounal of Artificial Intelligence Research*, 4:237–285, 1996. `http://www.cs.washington.edu/research/jair/abstracts/kaelbling96a.html`.

[25] David Kortenkamp, Marcus Huber, Frank Koss, William Belding, Jaeho Lee, Annie Wu, Clint Bidlack, and Seth Rogers. Mobile robot exploration and navigation of indoor spaces using sonar and vision. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS 94)*, pages 509–519, March 1994.

[26] C. M. Krishna. *On the design and analysis of real-time computers.* PhD thesis, University of Michigan, Ann Arbor, MI, 1984.

[27] John E. Laird. *Universal Subgoaling.* PhD thesis, Computer Science Department, Carnegie-Mellon University, 1983.

[28] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, January 1986.

[29] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[30] P. Langley, G. L. Bradshaw, and H. A. Simon. BACON.5: The discovery of conservation laws. In *Proceedings IJCAI-81*, 1981.

[31] P. Langley, H. A. Simon, G. L. Bradshaw, and J. M. Zytkow. *Scientific Discovery.* The MIT Press, Cambridge, MA, 1987.

[32] Pat Langley. Relevance and insight in experimental studies. *IEEE Expert*, pages 11–12, October 1996.

[33] Pat Langley. Learning to sense selectively in physical domains. In *Proceedings of the First International Conference on Autonomous Agents*, 1997. `ftp://robotics.stanford.edu/pub/langley/sense.aa97.ps.gz`.

[34] Jill Fain Lehman, Richard L. Lewis, and Allen Newell. Natural language comprehension in Soar: Spring 1991. Technical Report CMU-CS-91-117, School of Computer Science, Carnegie Mellon University, March 1991.

[35] Donald Michie and R. A. Chambers. *Machine Intelligence 2*, chapter BOXES: An Experiment in Adaptive Control, pages 125–133. Elsevier/North-Holland, Amsterdam, London, New York, 1968.

[36] Craig M. Miller. *A model of concept acquisition in the context of a unified theory of cognition*. PhD thesis, The University of Michigan, Dept. of Computer Science and Electrical Engineering, May 1993.

[37] Allen Newell. Physical symbol systems. *Cognitive Science*, 4:135–183, 1980.

[38] Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.

[39] Allen Newell and Herbert A. Simon. GPS: A program that simulates human thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. R. Oldenbourg KG., 1963.

[40] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, Reading, MA, 1994.

[41] Douglas J. Pearson. Tcl variable ideas. soar-group@cs.cmu.edu mailing list, June 1995.

[42] Douglas J. Pearson. [new] data-chunking example. `http://ai.eecs.umich.edu/-people/douglasp/soar/data-chunk-read-me.html`, February 1996.

[43] Douglas J. Pearson, Scott B. Huffman, Mark B. Willis, John E. Laird, and Randolph M. Jones. A symbolic solution to intelligent real-time control. *Robotics and Autonomous Systems*, 11:279–291, 1993. `http://ai.eecs.umich.edu/people/douglasp/pubs/robojournal.html`.

[44] Douglas J. Pearson and John E. Laird. Toward incremental knowledge correction for agents in complex environments. In *Machine Intelligence*, volume 15. Oxford University Press, 1996. `http://ai.eecs.umich.edu/people/douglasp/pubs/machine-int.html`.

[45] Garret Pelton. Generating constants. `ftp://centro.soar.cs.cmu.edu/afs/cs/-project/soar/member/gap/doc/generating-constants.Z`.

[46] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[47] Alfred A. Rizzi. *Dexterous Robot Manipulation*. PhD thesis, Yale University, 1994.

[48] Seth O. Rogers. Air-Soar documentation. `http://ai.eecs.umich.edu/air-soar`, 1993.

[49] Paul S. Rosenbloom, SooWoon Lee, and Amy Unruh. Responding to impasses in memory-driven behavior: A framework for planning. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[50] D. E. Rumelhart and J. L. McClelland, editors. *Parallel Distributed Processing*. MIT Press, Cambridge, MA, 1986. In two volumes.

[51] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[52] Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. Learning to fly. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 385–393, 1992.

[53] Stefan Schaal and Christopher G. Atkeson. Robot juggling: Implementation of memory-based learning. *IEEE Control Systems Magazine*, 14(1):57–71, February 1994. `ftp://ftp.cc.gatech.edu/pub/people/sschaal/schaal-CMS94.html`.

[54] Stefan Schaal and Christopher G. Atkeson. Robot learning by nonparametric regression. In *Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, volume 1, pages 478–485, 1994. `ftp://ftp.cc.gatech.edu/pub/people/sschaal/schaal-IROS94.html`.

[55] Stefan Schaal and Christopher G. Atkeson. From isolation to cooperation: An alternative view of a system of experts. In D. S. Touretzky, M. C. Moser, and M. E. Hanselmo, editors, *Advanced in Neural Information Processing Systems 8*, pages 605–611, Cambridge, MA, 1996. MIT Press. `ftp://ftp.cc.gatech.edu/pub/people/sschaal/schaal-NIPS95.html`.

[56] On-line shortcuts. Sixteenth Soar Workshop, June 1996.

[57] Paul E. Utgoff. ID5: An incremental ID3. In John E. Laird, editor, *Proceedings of the Fifth International Conference on Machine Learning*, pages 107–120, San Mateo, CA, June 1988. Morgan Kaufman.

[58] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, pages 435–461, New York, 1960.

[59] Michael D. Williams, James D. Hollan, and Albert L. Stevens. *Mental Models*, chapter Human Reasoning About a Simple Physical System, pages 131–154. L. Erlbaum Associates, Hillsdale, NJ, 1983.

[60] Gregg R. Yost. Acquiring knowledge in Soar. *IEEE Expert*, 8(3):26–34, 1993.