Effective and Efficient Memory for Generally Intelligent Agents

by

Nathaniel Leonard Derbinsky

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
2012

Doctoral Committee:

Professor John E. Laird, Chair
Professor Benjamin J. Kuipers
Professor Richard L. Lewis
Assistant Professor Michael J. Cafarella
Michael C. van Lent, Soar Technology

"Compassion is what you're good at. I'm better at complex searches through organized data structures."

–Jane (Orson Scott Card, *Speaker for the Dead*)

© Nate Derbinsky 2012
All Rights Reserved

for my family

ACKNOWLEDGMENTS

I would like to thank Art Padilla and Laura Lunsford, who encouraged me to pursue a PhD. With their personal support and mentoring, as well as the innumerable and invaluable experiences I had while a Park Scholar at NC State (a program they created and nurtured), I have come to learn a great deal about myself, service to others, and this ever-exciting world of research. Without Art and Laura, I would never have come to the University of Michigan.

I would also like to thank John Laird, my advisor, who made the PhD an enlightening and enjoyable journey. Since our first meeting, John has shared his vast knowledge of AI and cognitive science; his curiosity for the possibilities of human-level intelligence; and his advice to navigate the challenges of life and research. Over the years, John has masterfully navigated a fine balance of focussing my attention while also supporting me in my many and varied interests and diversions. I will be forever grateful for his honesty, his generosity, his understanding, and his encouragement. Without John, I would never have become a researcher.

Thanks to the Soar group for friendship, discussions, and lots of help; appreciation in particular to Scott Lathrop, Bob Marinier, Sam Wintermute, Joseph Xu, Jon Voigt, Shiwali Mohan, and Justin Li. Finally, thanks to some really great folks that made grad school memorable and fun: Melanie Harries, Andrea Angott, Kyla McMullen, Jin Hu, Gargi Datta, Laura Fink, Mark Hodges, Jim Boerkoel, Olga Kornievskaia, Kaushik Veeraghavan, Kelly Cormier, Karen Alexa, Lucie Howell, David Chesney, Susan Montgomery, Georg Essl, Rada Chirkova, Francine Dolins, and Julie Weber.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF FIGURES vii

LIST OF TABLES x

LIST OF APPENDICES xi

ABSTRACT xii

CHAPTER

I. Introduction 1

1.1 Research Approach 3
1.1.1 Environment, Task, and Agent Analysis 3
1.1.2 Architectural Integration 4
1.1.3 Broad Empirical Evaluation 5

II. Memory Requirements 7

2.1 Environment, Task, and Agent Characteristics 7
2.2 Requirements for Memory Systems 10
2.3 Work in Related Research Fields 14
2.3.1 Cognitive Modeling/Architecture 14
2.3.2 Case-Based Reasoning 17
2.3.3 Information Retrieval/Database Management Systems 17
2.3.4 Knowledge Representation and Reasoning 18
2.3.5 Summary 19

III. Research Approach: Cognitive Architecture 21

3.1 Cognitive Architecture 22

3.2 Architecture Selection 23
3.2.1 Task-Independent, Effective and Efficient Access to Diverse Knowledge (R3-R6) 24
3.2.2 Support for Complex Tasks, Environments, and Interactions (C1-C5) 25
3.3 The Soar Cognitive Architecture 26
3.3.1 Architecture Overview 26
3.3.2 Analysis 30

IV. Episodic Memory 34

4.1 Motivation 34
4.2 Related Work 36
4.2.1 Episodic Memory in Soar 36
4.2.2 EM: A Generic Memory Module for Events 37
4.2.3 Case-Based Reasoning 38
4.3 Functional Specification 39
4.3.1 Integration with Soar 39
4.3.2 Analysis 41
4.4 Efficient Implementation 41
4.4.1 Regularities of State Representation and Dynamics . 42
4.4.2 Episode Storage 43
4.4.3 Cue Matching 46
4.4.4 Reconstruction 51
4.5 Evaluation 53
4.5.1 Metrics 53
4.5.2 Agent Capabilities 54
4.5.3 Word Sense Disambiguation 55
4.5.4 Generalized Planning 59
4.5.5 Video Games and Mobile Robotics 62
4.6 Discussion 69
4.6.1 Future Work 70

V. Semantic Memory 71

5.1 Motivation 71
5.2 Related Work 72
5.3 Functional Specification 74
5.3.1 Integration with Soar 75
5.3.2 Comparison to ACT-R 76
5.4 Efficient Implementation 77
5.4.1 Assumption: Small Element Cardinality 78
5.4.2 Contextual Meaning of Efficient Support 78
5.4.3 Cue Matching 79
5.5 Evaluation 85

5.5.1 Lexical Queries 85
5.5.2 Synthetic Data 87
5.5.3 Mobile Robotics 89
5.5.4 Word Sense Disambiguation 93
5.6 Discussion 106
5.6.1 Future Work 107

VI. Competence-Preserving Retention of Learned Knowledge . . 109

6.1 Motivation 109
6.2 Related Work 110
6.3 Functional Specification 111
6.3.1 The Forgetting Problem 112
6.4 Efficient Implementation 113
6.4.1 Analysis 115
6.5 Evaluation 116
6.5.1 Synthetic Data 116
6.5.2 Mobile Robotics 118
6.5.3 Multiplayer Dice Game 124
6.6 Discussion 129
6.6.1 Future Work 129

VII. Summary and Conclusion 131

7.1 Requirements Revisited 131
7.2 Future Work 134
7.2.1 Memory Space 134
7.2.2 Mechanism Integration 134
7.2.3 Agent Development 135
7.3 Research Contributions 135
7.4 Conclusion 136

APPENDICES 138

BIBLIOGRAPHY 185

LIST OF FIGURES

Figure

3.1 The structure of the Soar cognitive architecture 27

3.2 Search spaces in problem solving 31

4.1 Integration of episodic memory in Soar. 40

4.2 Episodic memory dynamic-graph index. 44

4.3 Regression of the *Storage* operation across several domains. 46

4.4 Regression of the *DNFGraph* component of the *CueMatching* operation in the TankSoar domain. 50

4.5 Regression of the Interval-Search component of the *CueMatching* operation in the TankSoar domain. 51

4.6 WSD: retrieval-time data versus encoded episodes, controlling for feature co-occurrence. 58

4.7 WSD: retrieval-time data versus encoded episodes, controlling for temporal selectivity. 59

4.8 Planning: retrieval-time data versus encoded episodes for the detecting-repeated-states experiment (reactive domains). 61

4.9 TankSoar: domain screenshot. 63

4.10 Eaters: domain screenshot. 64

4.11 Infinite Mario: domain screenshot. 65

4.12 Mobile Robotics: domain map. 67

4.13 Mobile Robotics: timing data for goal-management cue. 69

5.1 Example output of synthetic generator ($k = 3$). 86

5.2 Synthetic: augmentation-selectivity sweep. 88

5.3 Synthetic: cue-size sweep with linear regressions. 89

5.4 Mobile Robotics: domain map, with robotic agent located in the Soar lab (3844 BBB). 90

5.5 Mobile Robotics: working-memory size versus elapsed time. 92

5.6 Mobile Robotics: max decision-cycle time versus elapsed time. . . . 93

5.7 WSD: SemCor cumulative word proportion vs. certainty. 97

6.1 Synthetic: evaluation of decay-approximation quality. 117

6.2 Synthetic: evaluation of decay-approximation complexity ("approx") as compared to binary parameter search ("bsearch"). 118

6.3 Mobile Robotics: working-memory size comparison. 122

6.4 Mobile Robotics: maximum decision-time comparison. 123

6.5 Dice: Average memory usage. 127

6.6 Dice: Average task performance (% wins) ± 1 standard deviation. . 128

6.7 Dice: Average decisions/task action ± 1 standard deviation. 128

D.1 TankSoar: timing data for all cues. 160

D.2 TankSoar: timing data for all cues (sans map squares, #11-12; y-axis reduced to 5 msec.). 160

D.3 Eaters: timing data for all cues. 162

D.4 Eaters: timing data for all cues (y-axis reduced to 1 msec.). 162

D.5 Infinite Mario: timing data for all cues. 164

D.6 Infinite Mario: timing data for all cues
(sans visual scene, #13; y-axis reduced to 2 msec.). 164

D.7 Mobile Robotics: timing data for all cues. 166

D.8 Mobile Robotics: timing data for all cues
(sans goal management, #6; y-axis reduced to 1 msec.). 166

H.1 CDF of augmentation cardinality in SUMO. 179

H.2 CDF of augmentation cardinality in OpenCyc. 180

H.3 CDF of augmentation cardinality in WordNet. 180

LIST OF TABLES

Table

1.1 Overview of dissertation contributions. 6

2.1 The requirements on memory systems for generally intelligent agents imposed by characteristics of environment, task, and agent 14

2.2 The degree of requirements coverage in related fields of research . . 19

4.1 Evidence supporting temporal contiguity from several domains. . . . 42

4.2 WSD: occurrence and cue endpoints for SemCor. 57

4.3 Summary of empirical results for episodic memory in video-games and mobile-robotics domains. 68

5.1 Statistics of synthetic data sets used for semantic-memory evaluation. 87

5.2 WSD: Semantic concordance task-analysis summary. 98

5.3 WSD: Baseline task-performance results. 99

5.4 WSD: Memory bias task performance results for SemCor. 104

5.5 WSD: Memory bias task performance results for Senseval-2. 104

5.6 WSD: Memory bias task performance results for Senseval-3. 104

5.7 WSD: Individual bias evaluation: maximum query time. 105

5.8 WSD: Base-level activation approximation results. 106

LIST OF APPENDICES

Appendix

A. The Space of Memory Models 139

B. Episodic Memory: Relational Schemas 142

C. Episodic Memory: Algorithms 147

D. Episodic Memory: Evaluation Cues 158

E. Semantic Memory: Negative Cues 167

F. Semantic Memory: Relational Schemas 170

G. Semantic Memory: Algorithms 174

H. Semantic Memory: Augmentation Cardinality CDFs for SUMO, OpenCyc, and WordNet 179

I. Forgetting: Algorithms 181

ABSTRACT

Effective and Efficient Memory for Generally Intelligent Agents

by

Nathaniel Leonard Derbinsky

Chair: John E. Laird

Intelligent systems with access to large stores of experience, or *memory*, can draw upon and reason about this knowledge in a variety of situations, such as to improve the efficacy of their learning, decision-making, and actions in the world. However, little research has examined the computational challenges that arise when real-time agents require access to large stores of knowledge over long periods of time.

This dissertation explores the computational trade-offs involved in enhancing intelligent agents with effective and efficient memory. We exploit general properties of environments, tasks, and agent cues in order to develop scalable algorithms for *episodic* learning (autobiographical memory); *semantic* learning (context-independent store of facts and relations); and competence-preserving retention of learned knowledge (policies to *forget* memories while maintaining task performance). We evaluate these algorithms in Soar, a general cognitive architecture, for hours-to-days of real-time execution and demonstrate that agents with effective and efficient memory benefit along numerous dimensions when tasked within a variety of problem domains, including linguistics, planning, games, and mobile robotics.

CHAPTER I

Introduction

The overarching goal of this dissertation is to facilitate the development of *generally intelligent* agents: artificial-intelligence (AI) systems that persist for long periods of time in complex environments, autonomously contending with, and continually improving performance on, multiple, complex tasks. Human beings are the only known exemplars of general intelligence and one advantage that humans have over current AI systems is effective and efficient access to large stores of experience, or *memory*.

Computationally, a memory mechanism captures, or *encodes*, some aspect of agent experience; *stores* this information as internal knowledge, potentially changing it over time; and provides access to *retrieve* portions of this experience at a later time (see Appendix A for a more detailed breakdown of the dimensions along which memory mechanisms differ). Agents with memory can draw upon and reason about their experience in a variety of situations, such as to improve the efficacy of their learning, decision-making, and actions in the world.

Two memory mechanisms that are commonly used in AI systems are *episodic* and *semantic*. As first described in depth by *Tulving* (1972), episodic learning captures events and history that are embedded in individual experience, while semantic learning extracts facts from their experiential context. Intuitively, semantic memories are general facts about the world that an individual "knows" (e.g. *the main*

campus of the University of Michigan is located in Ann Arbor), whereas episodic memories allow an individual to "remember" a personal history (e.g. *the first time I visited Ann Arbor I had lunch at Zingerman's Deli - the pickles were so tasty!*). Prior work suggests that incorporating these forms of memory can lead to agents that are more capable in problem-solving, individually (*Kuppuswamy et al.*, 2006; *Nuxoll and Laird*, 2012) and collaboratively (*Deutsch et al.*, 2008; *Macedo and Cardoso*, 2004); are better able to account for human psychological phenomena, such as those relating to language learning (*Ball et al.*, 2010), memory blending (*Brom et al.*, 2010), and emotional appraisal (*Gomes et al.*, 2011); and more believable as virtual characters (*Gomes et al.*, 2011), instructors (*Taatgen et al.*, 2006), and long-term companions (*Lim et al.*, 2011). However, relatively little work has examined the computational challenges associated with maintaining effective and efficient access to large amounts of episodic and semantic experience over long periods of time.

The crux of the computational problem of memory for generally intelligent agents arises in the trade-offs among the potential benefit of effective access to large stores of prior experience; the time constraints imposed by dynamic environments; and the computational boundedness of agents. Consider an agent that has incomplete and uncertain perception of the world, but is tasked with a complex problem. For this agent, memories of prior experience comprise one source of knowledge with which to make decisions and act in the world; therefore, the more experience the agent accumulates, and the more flexibly this knowledge can be mined for task-relevant information, the greater the likelihood that the agent will find it beneficial in achieving its goal(s). However, if this agent is computationally bounded, then it does not have unlimited storage capacity, and thus cannot accrue unlimited stockpiles of experience. Furthermore, if it is to remain reactive to environmental dynamics, it cannot expend unlimited time encoding, organizing, and/or searching past experience for useful knowledge. We provide greater detail for this argument in Chapter II, but it

is this interplay between the characteristics of environment, task, and agent that motivates and constrains the exploration of memory mechanisms that are both effective and efficient.

This dissertation studies effective and efficient memory functionality that is useful to agents in a variety of tasks and scales computationally to large stores of knowledge over long agent lifetimes. We describe and evaluate efficient algorithms to support effective episodic (Chapters IV) and semantic (Chapter V) memory mechanisms. We also investigate a general framework to selectively retain, or forget, learned knowledge (Chapter VI): policies that work well across a variety of problem domains, effectively balancing the task performance of AI systems in complex tasks with reductions in the time to retrieve, and memory to store, learned knowledge.

1.1 Research Approach

To make progress towards effective and efficient memory for generally intelligent agents, we approach this research with three strategies: analysis of regularities in environments, tasks, and agents; integration within a general cognitive architecture; and empirical evaluation across a variety of problem domains. This section describes each component of our approach, laying the groundwork for the remainder of the dissertation. Throughout this discussion, we make reference to Table 1.1 (page 6), which summarizes the contributions of this dissertation. The "Publications" row of this table references the venues in which this work was originally published (where the author label "D" indicates "Derbinsky").

1.1.1 Environment, Task, and Agent Analysis

Discussed in detail in later chapters, our episodic and semantic memory models commit to computational problems that are, in the worst case, intractable for generally intelligent agents in arbitrary domains. A major component of this dissertation is

the analysis of general properties of environments, tasks, and agents that can inform algorithm design, and in practice lead to acceptable performance across a variety of environments and tasks. The "Analysis" row of Table 1.1 summarizes these properties for each of the memory models, and the "Algorithms" row lists the computational elements that exploit these properties (where novel components are starred*). We apply database and information-retrieval techniques to effectively and efficiently support task-independent memory functionality, as well as a forgetting framework that scales to large memory stores while maintaining task performance.

1.1.2 Architectural Integration

The development of generally intelligent agents is beyond the scope of this dissertation. However, to make claims about the generality of our work, we integrate the memory models and forgetting policies within Soar (*Laird*, 2012), a general cognitive architecture. Chapter III provides further detail about cognitive architecture as an appropriate methodology for our research, as well as justifies the selection, and describes the details, of Soar. All of the algorithms described in this dissertation have been implemented within Soar v9.3.2, which is open source and available as a free download[1]. This work extends the set of learning mechanisms available to Soar agents, including incremental episodic and semantic learning that scale to large stores of knowledge over long periods of time. These algorithms commit to very general knowledge representations and operations, and thus, as we discuss in later chapters, the analyses and algorithms extend to cognitive architectures other than Soar, as well as other agent-based systems that need to incorporate effective and efficient memory functionality.

[1] http://sitemaker.umich.edu/soar

1.1.3 Broad Empirical Evaluation

The "Evaluation" row of Table 1.1 summarizes the problem domains in which we evaluate our work. In each of these domains, we focus on three primary metrics: (1) *reactivity*, (2) *scalability*, and (3) *task performance*. First, for memory mechanisms to be useful, it is important that they keep pace with environmental dynamics. Thus, as described in more detail in Chapter III, we frequently reference 50 milliseconds as an upper bound for computation time in order for a memory mechanism to remain reactive to an agent's environment. Our evaluation shows our memory-mechanism algorithms are sufficiently efficient for a broad class of environments and tasks. We also characterize the degree to which general properties of environments, tasks, and agents affect memory mechanism performance. Second, for agents that persist for long periods of time, memory mechanisms need to maintain reactivity and usefulness even as stored knowledge grows large. The "Evaluation" row of Table 1.1 summarizes the orders of magnitude to which we scale both stored knowledge, as well as agent lifetime. This degree of scaling is on par with or much greater than prior work in this space. Third, memory mechanisms are important to the degree that they are useful in a variety of tasks. Therefore, in each of these domains, we focus on how our memory mechanisms contribute to improving task performance. To the degree that these improvements are general, we summarize them in the "Agent Benefits" row of Table 1.1.

Finally, in Chapter VII, we revisit our research goals, evaluate the degree to which this dissertation makes progress towards the requirements for generally intelligent agents (Chapter II), and discuss fruitful directions for future work. We contend that the breadth and depth of this evaluation forms a major dissertation contribution, setting forth a benchmark suite that will be invaluable for future research of memory-endowed agents.

Table 1.1: Overview of dissertation contributions.

	Episodic Memory Chapter IV	Semantic Memory Chapter V	Selective Retention Chapter VI
Analysis Env. & Task Properties	• Temporal contiguity • Structural regularity • Cue-feature co-occurrence • Cue-feature selectivity (temporal/structural)	• Object cardinality • Bias efficiency/locality • Cue-feature co-occurrence • Cue-feature selectivity	• Temporal contiguity
Algorithms Novel*	• Dynamic-graph index* • Discrimination network* • Relational interval tree (*Kriegel et al.*, 2000)	• Inverted index (*Zobel and Moffat*, 2006) • Query optimization (*Chaudhuri*, 1998) • *Locally efficient* bias* • Base-level activation approximation*	• Decay map* • Base-level decay approximation* • Binary parameter search
Evaluation Domains & Scaling	• WSD[1], planning, video games, mobile robotics • $O(10^8)$ episodes, days of real-time	• Lexical queries, WSD[1], mobile robotics • $O(10^6)$ objects, hours of real-time	• Mobile robotics, multi-player dice game • $O(10^5)$ memories, days of real-time
Agent Benefits Demonstrations	• Cognitive capabilities: virtual sensing, detecting repetition, action modeling, environmental modeling, explaining behavior, managing long-term goals, predicting success/failure	• Adaptive heuristic reasoning bias • Improved reactivity • Access to large knowledge bases	• Reduced memory consumption • Improved reactivity
Publications Authors: Venue	• [DLL: AAAI 2012a] • [LDL: AAAI 2012] • [DLL: AAMAS 2012b] • [LDV: BRIMS 2011b] • [DL: ICCBR 2009]	• [DL: ICCM 2012a] • [LDL: AAAI 2012] • [DL: AAAI 2011] • [LDV: BRIMS 2011b] • [DLS: ICCM 2010]	• [DL: ICCM 2012a] • [DL: ICCM 2012b]

[1]Word Sense Disambiguation (*Navigli*, 2009)

CHAPTER II

Memory Requirements

In this chapter, we dissect memory systems in context of generally intelligent agents. We begin by characterizing this class of agents, enumerating properties of their structure, the types of task with which they contend, and the environment in which they are embedded, with a focus on how these characteristics relate to and constrain their memory mechanisms. Given this breakdown, we then set forth functional requirements on artificial memory mechanisms and discuss work in related research fields apropos the degree to which it applies to and satisfies these requirements.

2.1 Environment, Task, and Agent Characteristics

In this section we characterize generally intelligent agents by enumerating properties of their environments, tasks, and structure. We will use this characterization to develop requirements for memory mechanisms that support generally intelligent agents, requirements that will lend constraint and structure to our work in the remainder of this dissertation. This breakdown draws heavily on work by *Laird and Wray* (2010), in which they develop requirements for cognitive architecture, but specializes the discussion with respect to memory systems embedded within these architectures.

C1. **Environment is Diverse with Complex and Interacting Objects**

i. The agent can usefully interpret parts of the environment as if it consists of independent objects.

ii. There are many objects.

iii. Objects have numerous, diverse properties.

iv. Some objects share similarities with other objects.

C2. **Environment is Dynamic**

i. The environment changes independently of the agent.

ii. The environment may change rapidly, relative to agent decision-making.

iii. Environmental dynamics are complex: the agent cannot always accurately predict future states in detail.

iv. Some object properties (C1) change as a consequence of environment dynamics.

C3. **Task-Relevant Regularities Exist at Multiple Time Scales**

i. Environmental dynamics (C2) are not arbitrary: interactions are governed by physical laws that are constant, often predictable, and frequently lead to recurrence and regularity that impact the agent's ability to achieve goals.

ii. Regularities in environmental dynamics exist at multiple time scales.

iii. Regularities in environmental dynamics lead to regularities in intra- and inter-object property changes (C1, C2).

C4. **Tasks can be Complex, Diverse, and Novel**

i. Tasks properties and goals are complex.

ii. The agent will contend with numerous tasks during its existence.

iii. The agent will contend with tasks with novel properties and goals.

iv. Tasks vary in the time scales required to achieve them: some are close to the timescale of dynamics in the environment (C2) while others require extended behavior.

C5. **Agent/Environment/Task Interactions are Complex and Limited**

i. The environment is partially observable: it is impossible for the agents to perceive the entire state of the world.

ii. Agent sensors are noisy and may be occluded by objects (C1) and environmental dynamics (C2), making agent perception incomplete and uncertain.

C6. **Agent Computational Resources are Limited**

i. The agent has physical limits on its computational resources relative to dynamics of the environment (C2).

ii. Agent interactions (C5) within the complex (C1), dynamic (C2) environment and with complex tasks (C4), given bounded computational resources, make perfect rationality impossible.

C7. **Agent Existence is Long-Term and Continual**

i. Agent existence is long-term relative to primitive interactions with the environment (C2, C5).

ii. For the duration of its existence, the agent is always present in its environment.

2.2 Requirements for Memory Systems

Based upon the characteristics above, we derive the following requirements to constrain implementations of memory systems for generally intelligent agents. We refer to these requirements throughout the remainder of this dissertation when developing functional specifications, as well as evaluating our work. Once again, these borrow heavily from the cognitive-architecture requirements of *Laird and Wray* (2010), but are specialized for memory systems.

R1. **Support Incremental, Online Learning**

Given that the agent...

i. is continually (C7) embedded within an environment that changes quickly and in complex ways (C2); and

ii. must assimilate and exploit environmental regularities (C3), when and as they become apparent, to effectively contend with diverse ongoing and future tasks (C4);

the agent requires memory systems that...

i. support *incremental* encoding and storage of new information, such that the contents of the agent's internal knowledge cache keep pace with environmental dynamics; and

ii. support *online* knowledge retrievals, such that agent reasoning reflects and takes advantage of its latest observations of the state of the world.

R2. **Support Diverse, Comprehensive Learning**

Given that the agent...

i. is embedded within a complex environment (C1) for a long-term existence (C7); and

ii. must assimilate and exploit environmental regularities (C3), which occur at varying time scales, including those apparent from a single instance or spread across time, in order to effectively contend with diverse tasks (C4) that entail complex interactions (C5);

the agent requires memory systems that...

i. individually support *diverse* forms of learning, such that optimized mechanisms will efficiently and accurately detect specific types of environmental regularities; and

ii. conjointly support *comprehensive* coverage of learning, such that the agent is broadly sensitive to, as well as able to represent and apply, a wide variety of task-specific knowledge about the world.

R3. **Support Diverse Knowledge Representation**

Given that the agent...

i. is embedded within a complex environment (C1) for a long-term existence (C7); and

ii. must assimilate and exploit environmental regularities (C3) in order to effectively contend with diverse tasks (C4) that entail complex interactions (C5);

the agent requires memory systems that...

i. support representing *diverse* types of knowledge, including contextualized memories of experiences, as well as more generalized facts, beliefs, and relations about objects in the world.

R4. **Scale Efficiently to Large Bodies of Knowledge**

Given that the agent...

i. is embedded within a complex environment (C1) that changes quickly (C2) over a long-term, continual existence (C7); and

ii. is contending with diverse tasks (C4) entailing complex interactions (C5);

the agent requires memory systems that...

i. support *efficient* incorporation of new information and access to existing knowledge, such that agent retrievals, drawing from the wealth of available knowledge that arises from environmental and task experience over a long lifetime, are timely, given the rate of environmental dynamics.

R5. **Support Effective Access to Knowledge**

Given that the agent...

i. must assimilate and exploit environmental regularities (C3) in order to effectively contend with numerous tasks (C4) entailing complex interactions (C5); and

ii. is embedded within a dynamic environment (C2) and is limited with respect to its computational resources (C6);

the agent requires memory systems that...

i. support *effective* access to knowledge about environmental regularities and past task performance, such that retrievals improve the agent's ability to contend with the complexities of its current situation.

R6. **Support a Variety of Tasks**

Given that the agent...

i. is embedded within a complex environment (C1) that changes quickly (C2) over a long-term, continual existence (C7); and

ii. must assimilate and exploit environmental regularities (C3), given limited computational resources, in order to effectively contend with numerous tasks (C4) that entail complex interactions (C5);

the agent requires memory systems that...

i. encapsulate environmental regularities and interaction complexities that are *independent of task* and that occur at time scales greater than that of the agent, thereby reducing the complexity of learning task-dependent knowledge.

Table 2.1 illustrates how the characteristics (C1-C7) of environment, task, and agent, as described above, together impose these requirements (R1-R6) upon memory systems for generally intelligent agents. While all of these characteristics constrain memory systems, it is useful to note that task independence of the mechanism (R6) draws upon the breadth of the challenges with which generally intelligent agents contend, and all requirements are influenced by the constraint of dealing with numerous, complex, and novel tasks (C4), a property that typically does not apply to short-lived, task-dependent systems. As a result, most prior approaches don't provide solutions that meet all of these requirements, as we describe in the next section.

Table 2.1: The requirements on memory systems for generally intelligent agents imposed by characteristics of environment, task, and agent

Requirements	Characteristics: C1 Complex Environment	C2 Dynamic Environment	C3 Regularities in Environment	C4 Complex Tasks	C5 Complex Interactions	C6 Limited Agents	C7 Extended Agent
R1 Incremental Learning		✓	✓	✓			✓
R2 Comprehensive Learning	✓		✓	✓	✓		✓
R3 Diverse Representation	✓		✓	✓	✓		✓
R4 Scale Efficiently	✓	✓		✓	✓		✓
R5 Effective Access		✓	✓	✓	✓	✓	
R6 Task Independence	✓	✓	✓	✓	✓	✓	✓

2.3 Work in Related Research Fields

The characteristics of environment, task, and agent structure impose significant requirements upon memory systems that must be considered and satisfied concurrently. In this section we briefly and broadly discuss research in related fields and the degree to which their efforts relate to and satisfy these requirements in the context of memory systems.

2.3.1 Cognitive Modeling/Architecture

It is common for cognitive architectures and models to commit to task-independent approaches (R6) to the incremental and online encoding (R1) of arbitrary environmental perception (R3) as internal, declarative knowledge, as well as the later retrieval using diverse (R5), and often cognitively inspired, methods (*Langley et al.*, 2009). However, these mechanisms typically do not scale (R4) to large knowledge bases (e.g. *Douglass et al.*, 2009; *Douglass and Myers*, 2010). A prominent class of exception includes those architectures, like Soar (*Laird*, 2012), that utilize the Rete algorithm for efficient matching of procedural knowledge (*Forgy*, 1982; *Doorenbos*, 1995). Much work must still be done to explore the full breadth of learning mechanism implemen-

tation and integration (R2) that must be in place to effectively capture and apply the variety of task and environmental regularities encountered by long-living agents, including those of autobiographical agent experience (*Nuxoll and Laird*, 2012) and appraisals (*Marinier et al.*, 2009), as well as statistical regularities in environmental and task demands (*Anderson and Schooler*, 1991; *Schooler and Anderson*, 1997).

We now discuss some common cognitive architectures apropos of memory and selective-retention mechanisms.

2.3.1.1 ACT-R

ACT-R (*Anderson et al.*, 2004) has a declarative module that has been applied to model a large number of reasoning and memory phenomena in humans; its knowledge representation and functionality informs much of our work in Chapter V. ACT-R does not, however, have an episodic memory, though work has been done to model episodic-memory effects using the declarative memory (e.g. *Anderson and Ross*, 1980; *Altmann and Gray*, 1998). ACT-R does implement a selective-utilization policy, wherein declarative chunks below an activation threshold cannot be retrieved; however, decayed chunks are never removed, as activation spread and merging might overcome the effects of historical inactivity. Prior work has demonstrated that the ACT-R declarative module does not scale to large knowledge bases (e.g. *Douglass et al.*, 2009; *Douglass and Myers*, 2010) and so it is common to interface the memory with a declarative database or external processes for specific tasks.

2.3.1.2 EPIC

Research on EPIC (*Meyer and Kieras*, 1997) focuses on achieving quantitative fits to human behavior, especially on tasks that involve interacting with complex devices that involve visual, auditory, and tactile modalities. The EPIC conceptual organization calls for a long-term declarative memory, but makes no commitment as

to knowledge representation or retrieval processes/implementation. Models instead encode long-term knowledge as production rules and EPIC makes use of a rete (*Forgy*, 1982) to scale to large numbers of rules.

2.3.1.3 LIDA

The LIDA framework (*Franklin and Patterson*, 2006; *Snaider et al.*, 2011) has both a transient episodic memory for events and a long-term declarative memory for facts and autobiographical memories, both of which can be cued via the workspace. LIDA encodes "events" to episodic memory when attentional "codelets" (implemented as arbitrary code) form coalitions to select knowledge for "conscious broadcast." Long-term learning, or consolidation, occurs off-line: "perhaps largely during REM sleep, the entire, as yet undecided contents of transient episodic memory are written (encoded) into declarative memory." LIDA does not make strong commitments to knowledge representations or processes, only providing default implementations of certain modules (e.g. a sparse distributed memory for episodic (*Snaider and Franklin*, 2011), the knowledge for which is represented within the architecture as bit vectors). Work on software to implement the architecture began in 2009 and it is currently released in beta form. Some work has applied LIDA for cognitive modeling (*Madl et al.*, 2011), though no work has been published for real-time agents.

2.3.1.4 Icarus

The Icarus architecture (*Langley et al.*, 2004; *Langley and Choi*, 2006) has hierarchical, long-term knowledge of concepts and skills. While Icarus implements skill learning through analysis of successful problem-solving, it does not detail methods for acquisition of new conceptual knowledge. Icarus does not have an episodic memory, though preliminary work has been done to represent and reason about time and temporally organized beliefs (*Stracuzzi et al.*, 2009). In that work, *Stracuzzi et al.*

indicate that while all beliefs are retained in Icarus (i.e. there is no explicit forgetting policy), some details may be lost when reconstructing the details of event sequences from temporal beliefs. Icarus has been applied in complex environments, such as for human-robot interaction (*Trivedi et al.*, 2011) and as a non-player character in a first-person shooter game (*Choi et al.*, 2007), but details have not been published about the duration of execution, nor about agent reactivity. Of particular concern is that Icarus processes each perceptual state in its entirety, regardless of processing time available, an architectural commitment that may interfere with reactivity in time-sensitive domains.

2.3.2 Case-Based Reasoning

Case-based reasoning (*Kolodner*, 1993) research focuses on methods for effectively accessing (R5), adapting, and incrementally updating (R1) prior case information to solve specific problems. While work has been done in case-base maintenance methods (e.g. *Cummins and Bridge*, 2009) to combat issues of case utility in large case bases (*Smyth and Cunningham*, 1996), research typically does not focus on online problem solving at the pace of environmental dynamics (R1), nor are most systems evaluated over long lifetimes (R4). Most work is highly task-specific (R6), including static, problem-specific case formats (R3), as well as problem-optimized case retrieval, adaptation, revision, and retention algorithms (R2).

2.3.3 Information Retrieval/Database Management Systems

The Information Retrieval (*Singhal*, 2001) and Database Management System (*Ramakrishnan and Gehrke*, 2003) research communities have developed substantial literature over the last 50 years (e.g. *Codd*, 1970; *Agrawal and Srikant*, 1994; *Gray et al.*, 1997; *Chaudhuri*, 1998; *Zobel and Moffat*, 2006) on efficient data structures and techniques (R4) for supporting task-independent (R6), expressive queries (R5),

on large amounts of diverse data (R3), as well as batch analytical and statistical processing (R2). However, while it is common for problem specifications to detail properties of queries, users, and data services (e.g. *Agrawal et al.*, 2000), it is rare for research in these fields to focus on dynamic interactions with complex environments across numerous tasks. For example, recent work on guided interaction (e.g. *Nandi and Jagadish*, 2011) and query reformulations (e.g. *Rieh and Xie*, 2006) posit human-database collaboration to prune large, complex, and potentially dynamic query spaces, but these systems tend to be very specific to a constrained knowledge representation (e.g. keyword search) or task (e.g. information seeking). It is also rare to find work in these fields that contributes to issues of online (R1) or comprehensive (R2) learning of complex environmental and task regularities, nor effective forms of data access (R5) to support agent performance across a variety of problem domains.

2.3.4 Knowledge Representation and Reasoning

Knowledge Representation and Reasoning (KRR; *Davis et al.*, 1993) is an area of artificial-intelligence research that focuses on the epistemological and ontological issues of describing the diversity of the world (R3), paying particular attention to the effects on processes such as reuse (R6) and inference (R1, R5), typically with respect to properties of expressiveness, validity, and efficiency (R4). There is work applying KRR techniques to a variety of problems, such as planning (temporally and spatially), decision-making, and reasoning under uncertainty. However, the focus is typically not on diverse learning methods (R2) for numerous, novel tasks in complex domains.

It is also infrequent for knowledge bases or inference methods to make claims as to scaling (R4) to knowledge stores required for long-living, generally intelligent agents in dynamic environments. We now contrast our goals with three KRR systems that are exceptions to this trend: Algernon (*Crawford and Kuipers*, 1991), Cyc

Table 2.2: The degree of requirements coverage in related fields of research

Fields	Requirements: R1 Incremental Learning	R2 Comprehensive Learning	R3 Diverse Representation	R4 Scale Efficiently	R5 Effective Access	R6 Task Independence
Cognitive Modeling/ Architecture	☒	☐	☒		☒	☒
Case-Based Reasoning	☐	☐	☐	☐	☒	
Information Retrieval/ Databases		☐	☒	☒	☐	☒
Knowledge Representation	☐		☒	☐	☒	☒

(*Lenat*, 1995), and Scone (*Fahlman*, 2006). These KRR systems were developed as independent modules for use within AI systems, such as theorem proving (e.g. *Remolina*, 2001; *Siegel et al.*, 2005), qualitative reasoning (e.g. *Rajagopalan*, 1995; *Forbus and Hinrichs*, 2006; *Forbus et al.*, 2009), question answering (e.g. *Rickel*, 1995; *Lenat et al.*, 2010), and language understanding (e.g. *Tribble and Rosé*, 2006; *Curtis et al.*, 2006). However, there has been little work that has investigated the issues that arise when integrating knowledge-access mechanisms with goal-driven, real-time agents. For example, there has been little work that investigates how to design agents that effectively query these knowledge bases for task-relevant information (lisp functions in Algernon and Scone; n^{th}-order logic expressions in Cyc). Furthermore, while all three systems implement computational strategies for scalable query answering (Access-Limited Logic in Algernon; over 1000 competing heuristic modules in Cyc; and parallel marker-passing algorithms in Scone), there have not been thorough evaluations of these systems in dynamic environments, and so it is unclear as to whether they are suitably efficient for real-time agents.

2.3.5 Summary

The coverage of requirements in related fields of research is summarized in Table 2.2, where an empty cell signifies little-to-no contribution, an open box (☐) signifies partial contribution, and a crossed box (☒) signifies significant benefaction. From this

table, we gather three important points. First, the requirements that constrain the development of memory mechanisms, as imposed by the context of generally intelligent agents, are numerous and challenging in concurrence, such as scaling efficiently (R4), in a task independent fashion (R6), given a diversely represented (R3), comprehensive knowledge store (R2). Second, while it is unsurprising that no work in an individual field can lay claim to concurrently satisfying all of these requirements, there not work in any field that fully satisfies R2, the comprehensive spectrum of diverse learning mechanisms that memory systems must support to achieve human-level intelligence. Finally, there appear to be opportunities for cross-fertilization to more completely satisfy requirements within a single realm.

This dissertation applies database and information-retrieval techniques to effectively (R5) and efficiently (R4) support task-independent (R6) memory within a general cognitive architecture, extending the set of comprehensive learning mechanisms (R2) to include episodic (Chapter IV) and semantic learning (Chapter V). Additionally, Chapter VI evaluates a framework to support scalable, incremental learning (R1) by forgetting memories (R4) in a task-independent (R6) fashion, while still maintaining task performance (R5). This work will improve the coverage of comprehensive learning (R2) and efficient scaling (R4), in context of cognitive architecture, while maintaining existing requirement coverage.

CHAPTER III

Research Approach: Cognitive Architecture

One common approach to computational memory modeling is to develop mechanisms that are specialized for a specific problem or class of problems. For example, as discussed in Section 2.3.2, case-based reasoning systems commonly employ this tactic: the case format, as well as case retrieval, adaptation, revision, and retention algorithms, are specific to each task. By definition, this approach does not satisfy requirement R6 (task independence) and thus is not appropriate for research in context of generally intelligent agents.

A similar, though more general, approach to memory modeling is to investigate mechanisms that are specialized for a class of data types or representations. For example, many systems require that knowledge in memory adhere to a pre-specified "event" format (e.g. *Tecuci and Porter*, 2007a; *Deutsch et al.*, 2008; *Brom et al.*, 2010). These systems may afford agents added capability across a variety of problems, but this approach does not satisfy requirement R3 (diverse representations) and thus is also not appropriate for research in context of generally intelligent agents.

Yet another approach involves studying memory mechanisms, independent of agent architecture, goals, and behavior. As a prominent example, the rational analysis of memory (*Anderson*, 1991; *Anderson and Schooler*, 1991; *Schooler and Anderson*, 1997) posits that since the human brain optimizes behavior for task performance,

the best method for understanding human cognition lies in environmental and task analysis, rather than attempting to analyze specific human problem-solving methods. While useful in analyzing, developing, and evaluating individual memory mechanisms, this isolated approach considers only the structure, regularities, and interactions of environments and tasks, ignoring those of the agent (*Simon*, 1991). As a consequence, it is difficult to synthesize and apply these specifications within the context of a generally intelligent agent and leaves many integration questions unanswered.

To make progress towards memory for generally intelligent agents, our research approach is to study memory mechanisms in context of a general *cognitive architecture*. The remainder of this chapter discusses cognitive architecture as a research paradigm that is appropriate for this investigation; justifies the selection of *Soar* (*Laird*, 2012) as the most suitable architecture in context of our research goals; and both describes and analyzes details of Soar that will be pertinent throughout the remainder of this dissertation.

3.1 Cognitive Architecture

Research into cognitive architecture aims to develop and understand human-level intelligence across a diverse set of tasks and domains (*Newell*, 1990; *Langley et al.*, 2009). A cognitive architecture is a specification of those aspects of cognition that remain constant throughout the lifetime of an agent. These fixed components include short- and long-term memories of the agent's beliefs, goals, and experience; the representation of elements contained within these knowledge stores; functional processes that apply agent knowledge to produce behavior; and learning mechanisms that adapt agent knowledge over time. Cognitive architecture applies a systems-level approach to artificial-intelligence research, investigating how the integration of numerous computational mechanisms supports complex and adaptive behavior.

The last forty years witnessed the conception and development of many, diverse

cognitive architectures, but nearly all individual research efforts strive towards at least one of the following three goals: (1) biological plausibility, (2) psychological plausibility, and (3) agent functionality. For instance, systems such as Leabra (*O'Reilly*, 1996) attempt to computationally explore how intelligence arises from circuits of neurons and how architectural mechanisms and processes correspond to neurobiological data regarding brain regions and topological connectivity. By contrast, systems such as EPIC (*Meyer and Kieras*, 1997) and ACT-R (*Anderson et al.*, 2004) are typically applied at a layer above biological mechanisms and attempt to capture and model details of human performance, such as behavioral timing and memory recall errors, in a wide range of cognitive tasks. Finally, architectures like Soar (*Laird*, 2012) may look to humans for inspiration, but focus on developing artificial agents that demonstrate human-level intelligence, even if they function in ways, or make use of computational mechanisms, that very different from that of humans.

3.2 Architecture Selection

We consider the following two metrics when comparing architectures for evaluating memory mechanisms for generally intelligent agents. First, the architecture must *not*, a priori, invalidate a requirement for memory systems (see Section 2.2); if this were the case, candidate memory mechanisms would likely not satisfy the requirement either. As an example, consider an architecture in which agent state is represented as a fixed-length binary buffer: given such a constraint, it would be difficult to evaluate the degree to which a memory mechanism satisfies R3, that of supporting diverse knowledge representation. Second, we consider the degree to which the architecture can support experimentation across the dimensions (see Section 2.1) characterizing generally intelligent agents, the tasks with which they contend, and the environments in which they are embedded. Given these metrics, we discuss below the reasons for which we chose the Soar cognitive architecture (*Laird et al.*, 1987; *Newell*, 1990; *Laird*

and Rosenbloom, 1996; *Laird*, 2008, 2012) as a platform upon which to pursue our research goals.

3.2.1 Task-Independent, Effective and Efficient Access to Diverse Knowledge (R3-R6)

Especially relevant for this work is Soar's considerable history of efficiently representing and bringing to bear large bodies of knowledge to solve diverse problems using a variety of methods (*Doorenbos*, 1995; *Laird and Rosenbloom*, 1996). This simultaneous focus on efficiency and generality uniquely distinguishes Soar from other agent architectures.

At one extreme, some systems boast impressive generality and applicability, as exemplified by the impressive number and variety of psychological phenomena captured by ACT-R models (*Anderson et al.*, 2004). However, these systems rarely consider the computational implications of scaling their theoretical commitments to the large knowledge bases accumulated over long agent lifetimes. For instance, the ACT-R declarative module has been shown not to scale to large stores of declarative knowledge (*Douglass et al.*, 2009; *Douglass and Myers*, 2010). However, because ACT-R models primarily focus on explaining details of temporally constrained psychological experiments, research progress is not typically impeded and, until recently, little research has explored the degree to which the ACT-R theory can scale to the conditions with which generally intelligent agents grapple.

Diametrically opposed are systems that demonstrate competency on a constrained set of tasks and knowledge representation, while not contending with the overwhelming quantity and diversity of challenges with which generally intelligent agents contend in complex domains. For instance, Cyc (*Lenat*, 1995) demonstrates comprehensive data integration and hybrid inference capabilities over unparalleled size and scope of knowledge, but does not have search-control mechanisms necessary to contend with

simple real-time planning and problem-solving tasks.

By contrast, the current iteration of Soar (*Laird*, 2012) implements a fully relational, symbolic representation (R3) across a variety of task-independent (R6) memory systems that have been shown to scale (R4) to large stores of knowledge over long agent lifetimes. Consider, for instance, the TacAir-Soar system (*Jones et al.*, 1999), which was composed of thousands of symbolic production rules and managed 722 scheduled flights of fixed-wing aircraft that flew during an operational training exercise that ran for 48 continuous hours.

Building on Soar's existing implementation, it is tractable to study and evaluate memory mechanisms that are populated with large data sets, such as the WordNet lexicon (*Miller*, 1995), and experiences from agents, such as robots in real and simulated environments, that persist for days of cognitive real-time. At these time scales and degree of knowledge access, we can also begin to study how memory systems interact with other cognitive mechanisms (R5), such as procedural learning (*Laird et al.*, 1986; *Nason and Laird*, 2004), including their relative strengths and limitations in situations and tasks approaching that of human-level. Additionally, Soar's generality of knowledge representation and reasoning allows us to not only study a large spectrum of tasks, but to also extrapolate and apply our results to other systems and architectures.

3.2.2 Support for Complex Tasks, Environments, and Interactions (C1-C5)

The degree to which the agent architecture supports systems integration and deployment directly affects the ability to evaluate memory mechanisms on a variety of tasks in complex environments. While primarily an issue of engineering, this is a valid practical consideration that is crucial in the successful pursuit of our research goals.

Soar supports a variety of programming languages (e.g. C++, Java, and Python)

on all major operating systems (Windows, Mac OS, Linux, and iOS) and has been interfaced in diverse execution environments, including RL-Glue (*Tanner and White*, 2009; *Mohan and Laird*, 2011); game systems, such as ORTS (*Buro*, 2003; *Wintermute et al.*, 2007), ATARI (*Wintermute*, 2010) and Quake (*Laird*, 2001); mobile music generation (*Derbinsky and Essl*, 2011, 2012); and robotics simulation and hardware platforms (*Laird et al.*, 2011b).

This broad applicability sharply contrasts agent architectures that are limited by language/platform (e.g. Lisp requirement of ACT-R: *Anderson et al.*, 2004), as well as those that exist only as partial implementations (e.g. Icarus & Clarion: *Langley et al.*, 2004; *Langley and Choi*, 2006; *Sun*, 2006) or frameworks (e.g. LIDA: *Franklin and Patterson*, 2006; *Snaider et al.*, 2011).

3.3 The Soar Cognitive Architecture

We now discuss the technical and theoretical details of Soar relevant to our research of memory mechanisms. We begin with a description of Soar, and then analyze the degree to which the architecture lays foundations for satisfying the requirements (R1-R6) of memory systems for generally intelligent agents (Section 2.2).

3.3.1 Architecture Overview

Figure 3.1 shows the structure of Soar. At the center is a symbolic *working memory* that represents the agent's current state. Functionally, working memory serves as a common substrate upon which to represent arbitrary and novel combinations and compositions of symbols (R3): it is here that perception, goals, retrievals from long-term memory, external action directives, and structures from intermediate reasoning are jointly represented as a connected, directed graph. The primitive representational unit of knowledge in working memory is a symbolic triple (*identifier*, *attribute*, *value*), termed a working-memory element, or WME. The first symbol of a WME (identi-

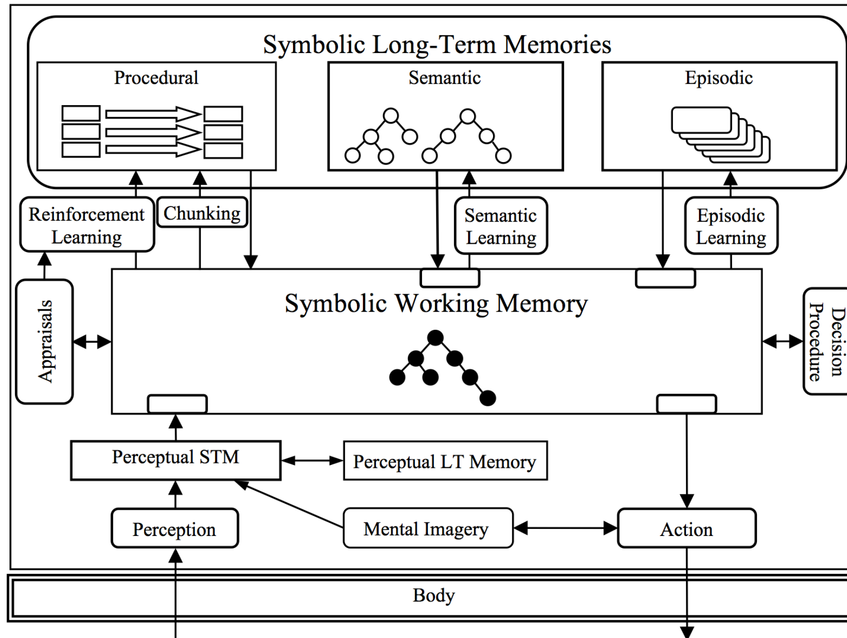

Figure 3.1: The structure of the Soar cognitive architecture (*Laird*, 2012).

fier) must be an existing node in the graph, whereas the second (attribute) and third (value) symbols may be either terminal constants or non-terminal graph nodes. Multiple WMEs that share the same identifier are termed an "object," and the set of individual WMEs sharing that identifier are termed "augmentations" of that object.

Procedural memory stores the agent's knowledge of when and how to perform actions, both internal, such as querying long-term declarative memories, and external, such as controlling robotic actuators. Knowledge in this memory is represented as if-then rules. The conditions of rules test patterns in working memory and the actions of rules add and/or remove working-memory elements. Soar makes use of the Rete algorithm for efficient rule matching (*Forgy*, 1982) and retrieval time scales to large stores of procedural knowledge (*Doorenbos*, 1995). However, the Rete algorithm is known to scale linearly with the number of elements in working memory, a computational issue that motivates maintaining a relatively small working memory (discussed further in Chapter VI).

Soar learns procedural knowledge via chunking (*Laird et al.*, 1986) and reinforce-

ment learning (RL; *Nason and Laird*, 2004) mechanisms. Chunking creates new productions: it converts deliberate subgoal processing into reactive rules by compiling over production-firing traces, a form of explanation-based learning (*Dejong and Mooney*, 1986). If subgoal processing does not interact with the environment, the chunked rule is redundant with existing knowledge and serves to improve performance by reducing deliberate processing. However, memory usage in Soar scales linearly with the number of rules, typically at a rate of 1-5 KB/rule, which motivates forgetting of under-utilized productions (discussed further in Chapter VI).

Reinforcement learning incrementally tunes existing production actions: it updates the expectation of action utility, with respect to a subset of state (represented in rule conditions) and an environmental or intrinsic reward signal. A production that can be updated by the RL mechanism (termed an *RL rule*) must satisfy a few simple criteria related to its actions, and is thus distinguishable from other rules (a distinction that is relevant to forgetting productions, discussed further in Chapter VI). When an RL rule that was learned via chunking is updated, that rule is no longer redundant with the knowledge that led to its creation, as it now incorporates information from environmental interaction that was not captured in the original subgoal processing.

Soar incorporates two long-term declarative memories, *episodic* (discussed in Chapter IV) and *semantic* (discussed in Chapter V). Broadly speaking, episodic memory incrementally encodes and temporally indexes snapshots of working memory, resulting in an autobiographical history of agent experience, while semantic memory stores working-memory objects, independent of overall working-memory connectivity. Agents retrieve knowledge from one of these memory systems by constructing a symbolic cue in working memory; the intended memory system (explicitly indicated by the agent) then interprets the cue, searches its store for the best matching memory, and if it finds a match, reconstructs the associated knowledge in working memory.

For episodic memory, the time to reconstruct knowledge depends, in part, on the size of working memory at the time of encoding, another motivation for a concise agent state (discussed further in Chapter VI).

Agent reasoning in Soar consists of a sequence of *decisions*, where the aim of each decision is to select and apply an operator in service of the agent's goal(s). The primitive decision cycle consists of the following phases: encode perceptual input; fire rules to elaborate agent state, as well as propose and evaluate operators; select an operator; fire rules that apply the operator; and then process output directives and retrievals from long-term memory. Unlike many other rule-based systems (e.g. ACT-R), multiple rules may fire in parallel during a single phase. The time to execute the decision cycle, which primarily depends on the speed with which the architecture can match rules and retrieve knowledge from episodic and semantic memories, determines agent reactivity. We have found that 50 milliseconds is an acceptable upper bound on this response time across numerous domains, including robotics, video games, and human-computer interaction (HCI) tasks; we will revisit this reactivity threshold in later chapters as a metric for evaluating mechanism performance.

There are two types of persistence for working-memory elements added as the result of rule firing. Rules that fire to apply a selected operator create operator-supported structures. These WMEs will persist in working memory until deliberately removed. In contrast, rules that do not test a selected operator create instantiation-supported structures, which persist only as long as the rules that created them match. This distinction is relevant to forgetting WMEs (discussed in Chapter VI).

As evident in Figure 3.1, Soar has additional memories and processing modules; however, they are not pertinent to this dissertation and are not discussed further.

3.3.2 Analysis

At an abstract level, Soar's symbolic representation of present state and topological integration of long-term declarative and procedural knowledge is not unique: this archetypal arrangement is similar to that of many other cognitive architectures, especially those that are either designed to model human behavior, such as ACT-R (*Anderson et al.*, 2004), LIDA (*Franklin and Patterson*, 2006), and Clarion (*Sun*, 2006), or are inspired by human behavior, such as Icarus (*Langley et al.*, 2004). Although there are many commonalities in these systems, there are also significant differences in design decisions of and theoretical commitments to knowledge representation, memory-system functionality, and learning. For instance, while Soar's short-term memory representation, as detailed below, is manifested as an arbitrarily complex symbolic graph, ACT-R maintains a fixed set of symbolic buffers and Clarion integrates symbolic and connectionist representations. These structural departures often reflect differences in research goals and phenomena of study.

To balance environmental reactivity (R1, R4) with rich access to large bodies of knowledge (R5) while contending with arbitrarily complex problems (R6), Soar adopts the *problem space hypothesis* (*Newell and Simon*, 1972) as a core theoretical commitment. According to this conjecture, problem solving in a task is defined as generalized search in a *problem space*. A problem space is composed of a set of *states* and a set of *operators*, which transform one state to the next. At each state in the problem space, knowledge is used to evaluate the operators that are available in the current state and determine the next-best operator. As the agent contends with multiple tasks and problems, it is possible that problem solving may extend over multiple problem spaces.

The process of extracting directly available knowledge from a knowledge base and making it available to the generative search process at the problem-space level is called *knowledge search* (*Newell*, 1990; *Strosnider and Paul*, 1994). The distinction

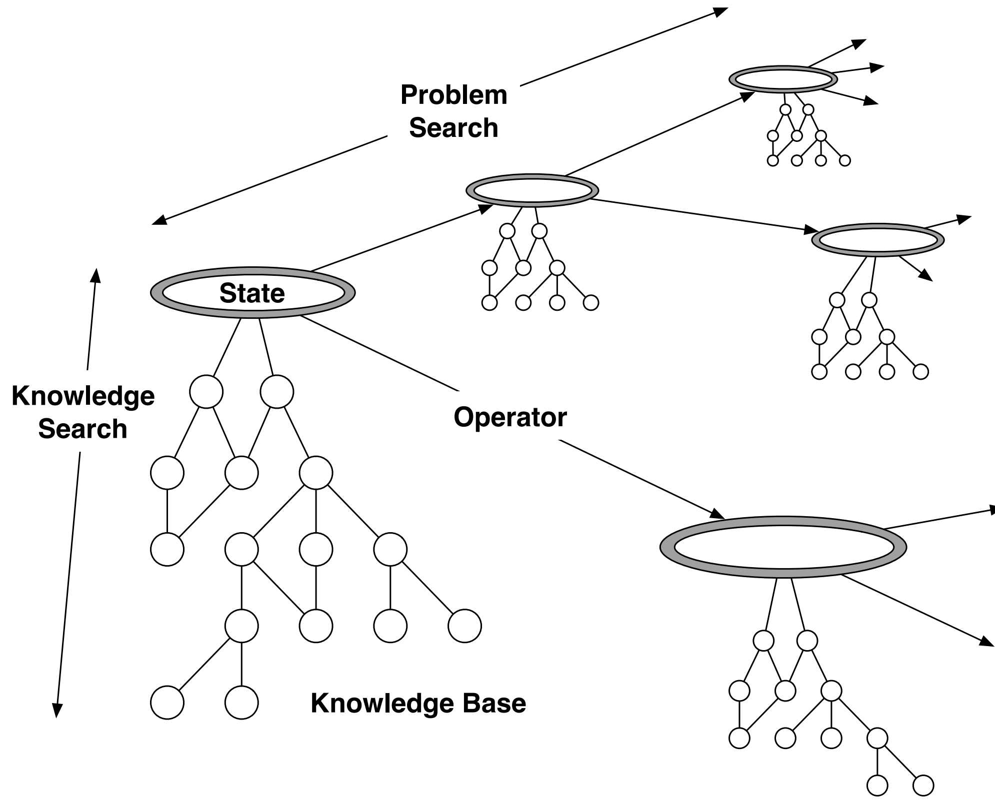

Figure 3.2: Search spaces in problem solving (*Strosnider and Paul*, 1994). States are represented by oval rings, operators as edges connecting states, and knowledge bases as the graphs stemming from each ring.

between problem search and knowledge search is depicted in Figure 3.2, where states are represented as shaded oval rings, operators as edges connecting these rings, and knowledge base(s) as the graphs stemming from each ring. Note that while the depth of the knowledge base below each state is limited, illustrating a finite quantity of immediate information, the plane upon which states are situated is potentially infinite in scope, demonstrating the possibility of an expansive problem space, resulting from the generative process of transforming states through operator application. Note also that search through problem space is not guaranteed to exhibit cyclicity: search progress is made as operators transform state and, given environmental dynamics and the resulting changing availability of operators within problem space(s), there is no guarantee of the ability to reach a prior state directly or indirectly via application of

one or more operators.

There are many types of knowledge search that are computationally unbounded, such as generalized logical inference and structural graph matching. Soar, however, firmly commits to the concept of bounded rationality (*Simon*, 1991), which contends that rationality of agent decision making is limited by information availability, mechanisms in the cognitive architecture, and finite time, relating to dynamic pressures of the environment. Consequently, Soar distinguishes generative search in problem space from bounded search of immediate knowledge in long-term memories. Thus any unbounded computation over knowledge is not incorporated within architectural retrieval mechanisms, but must be instead formulated as deliberate problem search, where task-dependent control knowledge can be brought to bear to prune the search space, while the agent maintains continual reactivity with its complex, dynamic environment. This deliberate search may involve multiple-knowledge accesses that are spread out over multiple decisions, as directed by task-relevant control knowledge.

A related commitment of the Soar cognitive architecture is the strict division between task-independent architectural mechanisms and task-dependent agent knowledge. This separation contains a strong analogy within the computer architecture world, in which designers strive to optimize system utility, as measured by such factors as production cost, potential for broad utilization, energy consumption, and application speed, by shifting the balance between those mechanisms fixed in highly efficient and potentially parallel hardware and the range of functionality supported by the less efficient, user-accessible software instruction set. When applied to knowledge search, this optimization process appeals to the difficulties, well-studied in the database and information retrieval communities, involved in maintaining a computationally efficient search over run-time domain knowledge (*Strosnider and Paul*, 1994). The resulting intuition, well-studied in computational theory, is that the greater the degree to which the complexity of a generalized search process can be constrained, the

greater the potential for efficient implementation via highly optimized data structures and algorithms. Soar applies this tenet by efficiently encoding in fixed mechanisms, such as memory systems, task-independent domain knowledge. For example, environmental regularities may exist at time scales that approach or exceed the life of the agent and exploiting this knowledge within architectural mechanisms may improve the quality and performance of knowledge retrieval processes (R6), while maintaining universal computation at the level of problem search, where experience and task-dependent agent knowledge may be brought to bear incrementally to prune complex task solving (*Laird and Wray*, 2010).

CHAPTER IV

Episodic Memory

This chapter documents our progress in understanding the computational challenges involved in extending generally intelligent agents with a task-independent episodic memory. We begin with a motivational description of episodic-memory systems, including a small amount of psychological background (Section 4.1); then discuss related work (Section 4.2); continue to our functional specification of an episodic-memory model (Section 4.3); describe data structures and algorithms that efficiently implement the mechanism (Section 4.4); evaluate the mechanism, as implemented within the Soar cognitive architecture (Section 4.5); and conclude with a summary and discussion of future work (Section 4.6).

4.1 Motivation

Tulving was the first to describe episodic memory in depth, characterizing it as a mechanism that captures historical knowledge contextualized in personal experience (*Tulving*, 1972, 1983). He distinguished it from semantic memory: intuitively, semantic knowledge (Chapter V) encodes what an individual "knows," whereas episodic knowledge represents an autobiographical stream of what an individual "remembers."

Nuxoll (2007) characterized episodic memory as having the following distinguishing functional characteristics that define it and may impact its implementation:

E1. **Architectural**: episodic retrievals are available for all tasks (related to R6, the task-independent requirement of memory systems for generally intelligent agents).

E2. **Automatic**: episodic memories are stored without deliberation and the process does not compete with knowledge-based reasoning. Reasoning can only indirectly influence episodic storage, such as through deliberate rehearsal (related to R1, the requirement for incremental, online learning).

E3. **Autonoetic**: retrieved episodic memories are distinguished from current sensing.

E4. **Autobiographical**: retrieved episodes are represented in the context in which they were originally experienced.

E5. **Temporally Indexed**: retrieved episodes include meta-data providing temporal context with respect to other episodes.

In context of the memory requirements for generally intelligent agents, incorporating episodic memory in a cognitive architecture contributes to the support of diverse, comprehensive learning (R2), by incrementally (R1) providing an agent rich (R3) access to a contextualized, temporally indexed, internal store of its prior experience. However, scaling (R4) effective access (R5) to this knowledge in a task-independent fashion (R6) over long agent lifetimes poses a significant challenge.

Prior research has shown that autonomous agents enhanced with episodic memory are more capable in problem solving, both individually (e.g. *Kuppuswamy et al.*, 2006; *Nuxoll and Laird*, 2012) and in collaboration with other agents (e.g. *Deutsch et al.*, 2008; *Macedo and Cardoso*, 2004); are better able to account for human psychological phenomena, such as those relating to memory blending (e.g. *Brom et al.*, 2010) and emotional appraisal (e.g. *Gomes et al.*, 2011); and are more believable as virtual characters (e.g. *Gomes et al.*, 2011) and long-term companions (e.g. *Lim et al.*, 2011).

Nuxoll (2007) postulated some of the functional roles episodic memory may serve in context of a general cognitive architecture, including *virtual sensing* (retrieving past sensing of features outside current perception), *action modeling* (predicting the outcome of actions), and *retroactive learning* (reviewing experiences and learning from them when sufficient time and/or other resources become available), and with Laird (2012), he demonstrated a subset of these capabilities. We return to these cognitive capabilities in Section 4.5, where we evaluate our episodic-memory implementation.

4.2 Related Work

Relatively little work examines the computational challenges associated with maintaining effective and efficient access to experience over long periods of time. Most approaches to storing and retrieving episodic knowledge are task-specific (e.g. *Macedo and Cardoso*, 2004) and/or apply to temporally limited problems, such as Ubibot (*Kuppuswamy et al.*, 2006), the ubiquitous robot, which was evaluated in a single 2D simulation that lasted 7 minutes and had fewer than 50 episodic memories.

4.2.1 Episodic Memory in Soar

The work of *Nuxoll* (2007) was the first to explore episodic memory within Soar. That work focused on issues of architectural integration and demonstrations of functional benefits for agents in two evaluation domains. This dissertation builds upon and extends this work along numerous dimensions, with a particular focus on computational efficiency and scaling. We present and evaluate algorithms and data structures that perform efficiently for days of real-time use; we evaluate these techniques in a much broader set of evaluation domains; we characterize performance in terms of general properties of domains and cues; and we exemplify a broader range of cognitive capabilities that agents can apply across domains by virtue of having an episodic memory.

4.2.2 EM: A Generic Memory Module for Events

EM (*Tecuci and Porter*, 2007a) is a generic store to support episodic-memory functionality in a variety of systems, including planning, classification, and goal recognition. EM is an external component with an API, wherein host systems must implement a thin interface layer. The term "episode" in EM defines a sequence of actions with a common goal and is represented as a triple: context ("general setting" of the episode), content (ordered set of the events that make up the episode), and outcome (a domain/task-specific evaluation of the result of the episode). Though meaningful in systems like planners, this representational constraint is inappropriate for generally intelligent agents (R3), as it may be difficult to pre-define action sequences and outcome evaluation functions for long-living agents (C7) that must contend with multiple, possibly novel, tasks (C4).

EM queries are composed of a partially defined episode and a single evaluation dimension. EM utilizes a two-stage evaluation scheme, whereby a constant number of candidate matches are identified (5 in published work) and then compared using a relatively expensive semantic matcher. Tecuci and Porter (2007a; 2007b) applied EM to planning, plan recognition, classification, and goal-schema recognition tasks in several domains. They presented evidence that in practice, their retrieval mechanism inspects far fewer events than are stored. However, they have not published retrieval-timing data, and thus it is unclear whether EM is applicable to real-time agents. Furthermore, their results come from learning over short periods of time (250-5000 episodes) in single-task domains, so it is unclear as to whether the underlying algorithms and data structures will be effective and efficient for agents with many orders of magnitude more episodes across a variety of problems.

4.2.3 Case-Based Reasoning

Episodic-memory research is closely related to studies in case-based reasoning (CBR). The goal of CBR is to optimize task performance given a case-base, where each case consists of a problem and its solution (*Kolodner*, 1993). In CBR systems, however, case structure is typically pre-specified, case-base size is either fixed or grows at a limited rate, and the cases usually do not have any inherent temporal structure. In contrast, an episodic store grows with experience, accumulating snapshots of an agent's experiences over time. An agent endowed with this memory can retrieve relevant episodes to facilitate reasoning and learning based upon prior events.

Efficient algorithms have been studied in CBR for qualitative and quantitative retrieval (e.g. *Stottler et al.*, 1989; *Wess et al.*, 1994; *Lenz and Burkhard*, 1996). The underlying algorithms and data structures supporting these algorithms, however, typically depend upon a relatively small and/or static number of case/cue dimensions, and do not take advantage of the temporal structure inherent to episodic memories.

Considerable work has been expended to explore heuristic methods that exchange reduced competency for increased retrieval efficiency (*Smyth and Cunningham*, 1996), including refined indexing (e.g. *Fox and Leake*, 1995; *Daengdej et al.*, 1996) and case-base maintenance (e.g. *Wilson and Martinez*, 2000; *Patterson et al.*, 2003; *Cummins and Bridge*, 2009). Many researchers achieve gains through a two-stage cue matching process that initially considers surface similarity, followed by structural evaluation (e.g. *Forbus et al.*, 1995).

The requirement of dealing with time-oriented problems has been acknowledged as a significant challenge within the CBR community (e.g. *Combi and Shahar*, 1997), motivating work on temporal CBR (T-CBR) systems (e.g. *Patterson et al.*, 2004), and research on the representation of and reasoning about time-dependent case attributes (e.g. *Jære et al.*, 2002), as well as preliminary approaches to temporal case sequences (e.g. *Ma and Knight*, 2003; *Sánchez-Marré et al.*, 2003). However, existing T-CBR

work does not deal with accumulating an episodic store, nor does it take advantage of temporal structure for efficient implementations.

4.3 Functional Specification

Our functional specification adopts and extends many of the high-level design decisions described by *Nuxoll* (2007). We describe those generally, followed by a mapping onto Soar, as depicted in Figure 4.1.

At a high level, our episodic-memory model comprises three phases: (1) *encoding* agent state; (2) *storing* this information as episodic knowledge; and (3) *supporting* retrieval at a later time. The episodic storage process automatically encodes a subset of agent state (E1, E2) at regular intervals (E2) and temporally indexes this knowledge within the episodic store (E5). This process does not modify or generalize stored episodic knowledge, and thus episodic knowledge grows strictly monotonically, faithfully capturing the full extent of agent experience. To retrieve episodic memories, the agent deliberately constructs an acyclic, graphical cue (E1), partially specifying relevant contextual features within the episode. The cue matching process selects a single match from the episodic store, defined as the most recent episode that has the greatest number of structures in common with cue leaf nodes, and the retrieved episode is fully reconstructed (E4) in a special buffer, such that the agent can reason about this knowledge without confusing current sensing and past experience (E3).

4.3.1 Integration with Soar

In Soar, agent state is represented in its working memory as a connected digraph. As depicted in Figure 4.1, this includes graph regions dedicated to reflecting environmental input, action directives to the agent's "body," and other knowledge for agent reasoning. During each decision cycle, Soar's episodic memory automatically encodes a subset of the contents of working memory. This information, as well as the

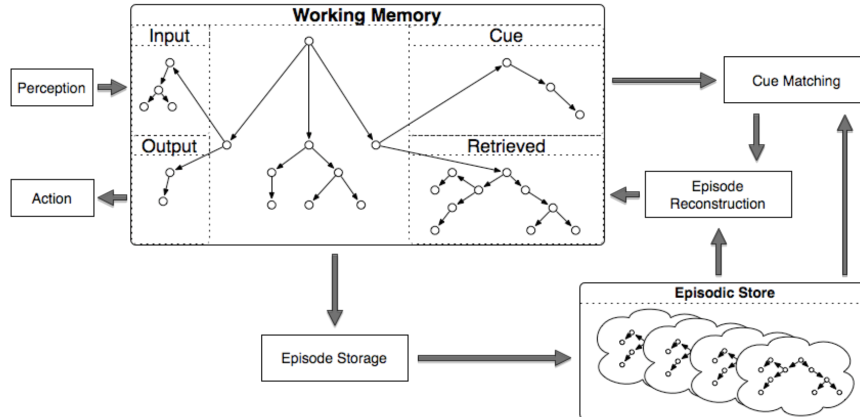

Figure 4.1: Integration of episodic memory in Soar.

time of encoding, is stored in episodic memory, where it remains without modification for the lifetime of the agent. Episodic memory does not encode the following portions of working memory: subgoal reasoning and retrieval buffers from both episodic and semantic memories. These automatic exclusions have analogous justifications. First, for many agents these regions include many working-memory elements that can change rapidly. For example, subgoal reasoning can build up large goal hierarchies that may retract when the agent perceives environmental features that invalidate the consistency of the stack. Second, rather than using episodic memories of prior retrievals/subgoal reasoning, the agent may be able to recreate these structures, or current task knowledge may make these memories unnecessary. Finally, in the case of subgoals, addressing dynamically sized stacks can be complex for agents and agent designers. Episodic memory does not encode structures that are not represented in working memory, such as procedural knowledge.

To retrieve an episode, agent task knowledge (represented as rules) constructs an episodic cue: a directed, connected, acyclic graph that specifies task-relevant relations and features. The cue-matching process identifies the "best" matching episode, defined as the most recent episode that has the greatest number of structures in common with cue leaf nodes. Episodic memory then reconstructs this episode within a pre-specified region of working memory.

4.3.2 Analysis

Soar's representation of working memory as an arbitrary graph structure has significant implications for the underlying implementation of episodic memory. A simpler representation, such as a vector or propositional representation, would make it possible to develop a simpler and faster implementation of episodic memory (R4), but at significant cost in expressiveness (R3) and generality (R6). Given this general representation, however, the underlying implementation of episodic memory (discussed in the next section) is independent of other details of Soar and should generalize to other architectures with graph-based representations of dynamic agent state.

The specification of cue matching commits to two algorithmic properties that affect scaling (R4). The process returns an episode if one exists that contains at least one feature in common with a cue leaf node. The mechanism also returns the "best" episode with respect to cue structure, leaf nodes, and temporal recency. Given these commitments, in the worst case, the encoding, storage, and retrieval operations scale at least linearly with the number of state changes.

4.4 Efficient Implementation

The goal of our efficient implementation is to exploit regularities of state representation and dynamics in order to improve expected performance of episodic memory via specialized data structures and algorithms. This section first describes and justifies the regularities we employ, and then proceeds to a detailed algorithmic discussion of for each of the three episodic operations: storage, cue matching, and reconstruction.[1] The design of this implementation was motivated by the need to minimize the growth in processing time for all episodic operations as the number of episodes increases. However, over long agent lifetimes, cue matching has the greatest growth

[1] This algorithmic work was previously published in (*Derbinsky and Laird*, 2009; *Derbinsky et al.*, 2012b).

Table 4.1: Evidence supporting temporal contiguity from several domains.

Domain	Avg. WM Size	Avg. WM Deltas	Avg. % Deltas/Size
Mobile Robotics	1,234.51	1.71	0.14%
TankSoar	2,733.13	22.31	0.82%
Infinite Mario	1,993.29	72.31	3.63%
Eaters	229.83	18.19	7.91%
WSD	78.77	6.38	8.10%

potential and the overall design is meant to minimize the time required for that operation without significantly impacting the memory or time required for the other operations. Appendix B details the knowledge representation as a set of relational schemas, while appendix C provides a concise description of the algorithms.

4.4.1 Regularities of State Representation and Dynamics

Our episodic-memory implementation makes two assumptions regarding regularities of agent state, both of which have been applied in the rule-matching literature. The first regularity is *temporal contiguity*: the world changes slowly, and thus changes to agent state, from episode to episode, will be few relative to the overall size of state. This idea is key to the space-time tradeoffs employed by the Rete algorithm for rule-matching systems (*Forgy*, 1982): since there are few working-memory changes at each time step, it is computationally beneficial to process each working-memory change, as opposed to a search across all rule conditions and/or working-memory elements. Table 4.1 summarizes evidence for this assumption in several experimental domains, including video games, mobile robotics, and a word sense disambiguation task (all discussed in more detail in Section 4.5). This data shows that, on average, few working-memory elements change per episode (2-73 WMEs) and that this is a relatively small proportion as compared to the agent's overall representation of state.

The second assumption is *structural regularity*: agent knowledge will reuse representational structure, and so over time, the number of distinct structures will be

much smaller than the total number of experienced structures. Optimizations to the Rete algorithm (e.g. *Doorenbos*, 1995) exploit this regularity through alpha/beta-node sharing. In preliminary domain analysis, we found that the only major exclusion to this assumption took the form of numerical representations with large, or infinite, ranges, such as a monotonically increasing representation of time or continuous values used to represent the position of the agent or objects.

4.4.2 Episode Storage

Given an agent whose present state is represented as a connected, directed graph, we define episodic storage as the process of encoding and storing, at a given point in time, all information necessary to recreate that state at a later time. By automatically associating a unique temporal identifier with this captured data (E5), a storage process supports an architectural (E1), automatic (E2), and autobiographical (E4) episodic-memory system.

To satisfy this definition, a naïve storage mechanism simply records all nodes and edges that comprise agent state during each episode. While sufficient, this approach has unsatisfying computational resource requirements: encoding time and storage space, per episode, is linear in the size of the state graph (*Nuxoll*, 2007).

Prior work by Nuxoll (2007) improved this approach by directly exploiting structural regularity and temporal contiguity. To exploit structural regularity, he developed a temporally global data structure, termed the *Working-Memory Tree*, that indexed all distinct structures that had ever been encountered during episodic encoding. This approach served to compress the representation of repeated structures, as well as support more efficient indexing for later cue matching. However, as the name implies, Nuxoll's data structure only supported a tree representation of episodes. We extended this work to support a fully relational representation, which we termed the *Working-Memory Graph* (WMG).

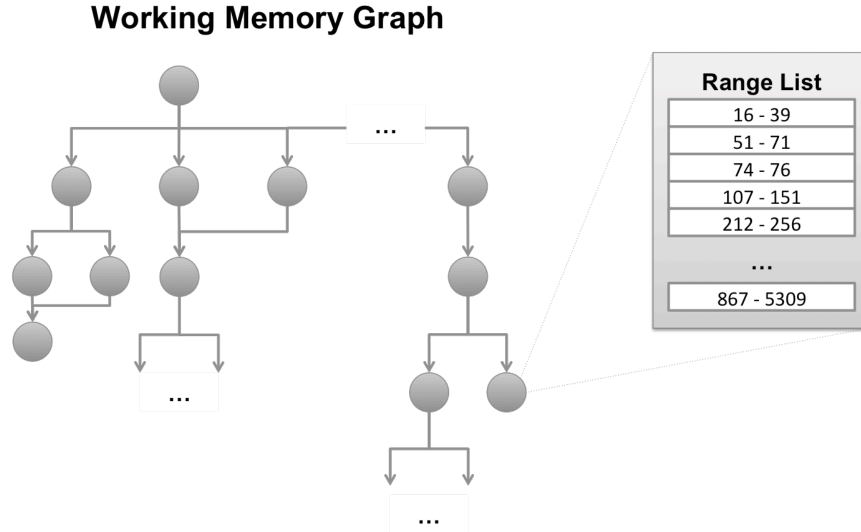

Figure 4.2: Episodic memory dynamic-graph index: the working-memory graph encodes all distinct graph structures and the temporal intervals capture the episodes when these structures were added to/removed from agent state.

To exploit temporal contiguity, Nuxoll represented episodes implicitly via temporal intervals over "pointers" to working-memory tree structures. The result was that encoding time and storage memory requirements scaled in the number of working-memory changes, as opposed to the size of each episode. As illustrated in Figure 4.2, we extended this approach to support the WMG. The combination of these two data structures is a novel dynamic-graph index that efficiently encodes all information necessary to reproduce all states of working memory at the granularity of an episode.

We formalized this representation as a set of relational schemas, which are detailed in Appendix B. We employed one optimization in the representation for which we had preliminary evidence of computational time and space savings. The optimization was the separation of interval representation into three types: the "now" intervals represent those structures that are currently in working memory; the "point" intervals represent those structures that were in working memory for only a single episode; and the "range" intervals are all other temporal ranges. The reason for distinguishing "now" from "range" is one of computational efficiency: all temporal ranges are

indexed, and thus there is a savings in not adding to/removing from indexes if not necessary. Therefore, new intervals are represented in the "now" relation; when the associated WME is removed from working memory, the starting point is removed from the "now" relation and added to either "point" or "range," as appropriate. The reason for distinguishing "point" from "range" has a small space-savings cost, but also prevents unnecessary indexing and interval-based querying and associated processing.

We populate these data structures with new intervals by executing the following algorithm for every element added to working memory:

1. if a corresponding edge does not exist in the WMG, add it
2. point the WME to the corresponding WMG edge
3. start a new interval at the pointed WMG edge

When a WME is removed from working memory, we record the end of the corresponding WMG-edge interval (opposite of step 3). Thus, our implementation only stores element *changes*.

4.4.2.1 Analysis

The description of our storage mechanism predicts that memory requirements will scale linearly with changes to agent state (R): $Storage = (X_0)(R)$. To fit parameters to this model, we ran 49 agents across 12 distinct domains (discussed in greater detail in Section 4.5), including video games, mobile robotics, planning, and word sense disambiguation. We collected average working-memory changes and storage requirements and performed a linear regression, as summarized in Figure 4.3. Our analysis found this model to be highly predictive ($R^2 = 0.98$). Consequently, the memory requirements for this algorithm will grow more quickly for those agents embedded within more dynamic environments. We also see that even with the cost of indexing, this algorithm requires only about 1-5 KB of storage per episode on

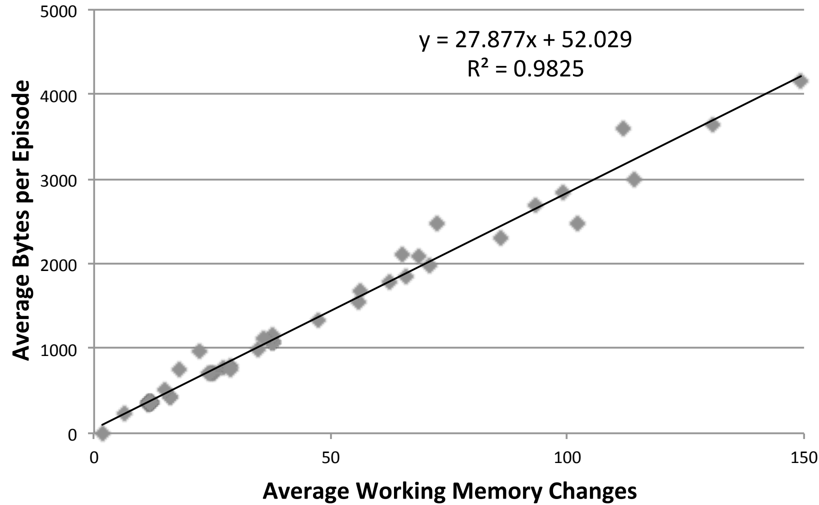

Figure 4.3: Regression of the *Storage* operation across several domains.

average; however, as the storage cost is variable with the changes in each episode, more turbulent episodes will consume greater amounts of memory.

4.4.3 Cue Matching

Given an episodic store, as illustrated in Figure 4.2, and a cue, represented as an acyclic graph, we define cue matching as identifying the most temporally recent episode that shares the greatest number of symbolic features structurally in common with cue leaf nodes. This operation functionally supports task-independent (E1) access to episodic knowledge.

To satisfy this definition, a naïve cue-matching mechanism performs a graph-match comparison between the cue and each episode in the store, beginning with the most recent and concluding once a perfect match is found or once all episodes are considered and ranked. While sufficient, this approach has unsatisfying computational-resource requirements: potentially exponential time, with respect to average episode size, for each graph-match execution, and a linear growth, with respect to the number of episodes in the store. With the following three strategies, we have built on prior work to improve the tractability of both of the aforementioned issues.

The first optimization is to only consider *candidate* episodes for evaluation, that is those episodes that contain at least one feature in common with the cue. It is computationally efficient to incrementally examine this list of candidates by cross-referencing the cue with the working-memory graph and merging pertinent lists of temporal ranges.

The second optimization is to implement *Interval Search* (*Nuxoll*, 2007). The key insight of this algorithm is that a candidate match score only changes at the endpoints of episode-element intervals. For example, consider the feature expanded in Figure 4.2. Were this node a cue feature, we could potentially avoid evaluating episodes 868 through 5308 (a savings of over 4000 evaluations!), as they are known, without exception, to all contain this feature. By just performing episode evaluation at interval endpoints (i.e. episodes where there is the potential for a *change* of evaluation score), we can achieve significant computational savings. We efficiently implement this algorithm by maintaining B+-tree indexes of all temporal-interval endpoints (one tree for interval start, another for interval end), keyed on WMG pointers. Walking a pointer's endpoints in descending order of time entails finding the most recent endpoint (log time with respect to the number of endpoints), and walking the leaf nodes in order (constant time per endpoint, since B+-tree leaf nodes form a doubly linked list). To process a multi-node cue, we maintain parallel B+-tree pointers for each cue node and all pointers within a B+-tree are stored in a priority queues (keyed on endpoint value). As pointers are popped from these priority queues, we perform episode evaluation, increment the B+-tree pointer, and then re-insert into the priority queue.

The final optimization is to minimize the frequency of potentially combinatorial graph-match evaluations by implementing a two-stage matching policy. The key observation here is that a candidate episode only has the potential to match a cue if it contains all *surface* cue features independently, where a surface feature is defined as a distinct, directed path from root to leaf. Given the WMG, surface-feature matching

is equivalent to satisfaction of a set of monotonic, disjunctive-normal-form (DNF) boolean clauses, where each literal is an edge in the cue and each clause is a path from root to leaf. To efficiently track satisfaction of these clauses, we developed and implemented a novel discrimination network, termed the *DNF Graph*, which efficiently and incrementally maintains satisfaction of a set of DNF formulas during each endpoint of Interval Search. At each endpoint in the Interval-Search algorithm, exactly one literal is activated or de-activated (depending on whether we encounter an end/start). If the associated cue edge is non-terminal, this change may entail recursive propagation to child literals. If, during propagation, we alter clause satisfaction, we modify the global match score. Thus, we extend endpoint iteration to track *only changes* in boolean satisfaction of the DNF Graph and, by extension, modifications of candidate match score.

If a perfect surface match is found, we perform a full graph-match comparison. This comparison utilizes standard heuristics to guide search, such as most-constrained-variable (MCV). If the cue and candidate episode unify structurally, the cue-matching process is complete. If not, we maintain a reference to the most recent match with the greatest surface-match score, which will be returned when Interval Search exhausts all candidate episodes.

The result of these optimizations is a cue-matching algorithm that is guaranteed to satisfy our functional specification, but which attempts to minimize temporal linear scanning and combinatorial graph-match evaluation by processing only *changes* between candidate episodes.

4.4.3.1 Analysis

To analyze how characteristics of environments, tasks, and agent cues affect the operation of our algorithms and data structures in practice, we developed predictive

performance models:

$$CueMatching = DNFGraph + IntervalSearch + GraphMatch$$
$$DNFGraph = (X_1)(log_2[U \cdot R])(L)$$
$$IntervalSearch = (X_2)(\frac{1}{T})(Distance)(\Delta)$$

To fit parameters to these models, we performed 100 isolated executions of primitive operations (*DNFGraph* and *IntervalSearch*) on data collected from 10 trials of the *mapping-bot* agent in the *TankSoar* domain (discussed in greater detail in Section 4.5) at 10 time points (100K, 200K, ... 1M). We collected the necessary episode statistics (described below) and performed linear regressions for 15 different queries

The constants in the equations (X_1, X_2) reflect linear scaling factors for a given computer. The *CueMatching* operation comprises *DNFGraph* and *IntervalSearch* operations. The *DNFGraph* operation is linearly dependent upon the logarithmic growth of the average number, U, of historically unique internal and leaf nodes multiplied by R, the total number of stored intervals, as well as linearly dependent upon L, the number of literals associated with the cue nodes. The slower-growing component (comprising U and R) predicts that time for this operation will scale with the number of distinct structures and the number of changes. The more dominant cost (L) refers to the *structural selectivity* of the cue, or the number of ways in which the cue could match a candidate episode. Our analysis in TankSoar (see Figure 4.4) found this model to be highly predictive of *DNFGraph* performance ($R^2 = 0.996$), suggesting that this algorithm will perform poorly if supplied with a cue that has features that can match candidate episodes in many ways. This algorithmic property is analogous to the problem of matching multi-valued attributes in rule-based systems (*Tambe et al.*, 1990).

The *IntervalSearch* operation is expressed as a proportion of relevant cue node

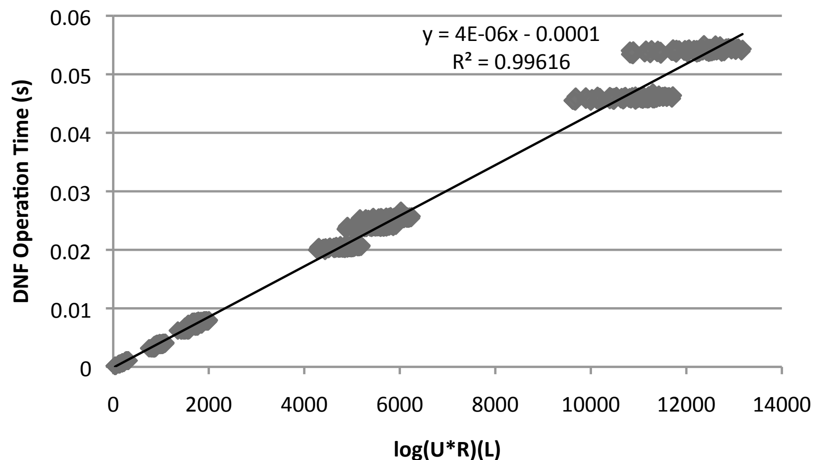

Figure 4.4: Regression of the $DNFGraph$ component of the $CueMatching$ operation in the TankSoar domain.

intervals. T represents the total number of episodes recorded. $Distance$ represents the temporal difference between the current episode and the best match. Δ represents the total number of intervals relevant to the cue. Intuitively, the farther back in time the algorithm must search for an episode, the more intervals it must examine. Given the nature of this algorithm, search distance will be affected by two properties of cue features: *temporal selectivity* (the number of episode endpoints indexed by a cue feature) and *co-occurrence frequency* (the likelihood of multiple features to co-occur within a single episode). Our analysis in TankSoar (see Figure 4.5) found this model to be highly predictive of $IntervalSearch$ performance ($R^2 = 0.989$), suggesting that this algorithm will perform poorly if supplied with a cue that has multiple features that occur in many episodes (i.e. low temporal selectivity) and/or have low probability of cue features co-existing within candidate episodes (i.e. low feature co-occurrence).

The $GraphMatch$ operation is much more difficult to characterize. CSP backtracking depends upon cue breadth, depth, structure (such as shared internal cue nodes), and corresponding candidate episodes, but can be combinatorial in the worst case (though our two-phase matching policy attempts to minimize this cost). We have not extensively studied this component.

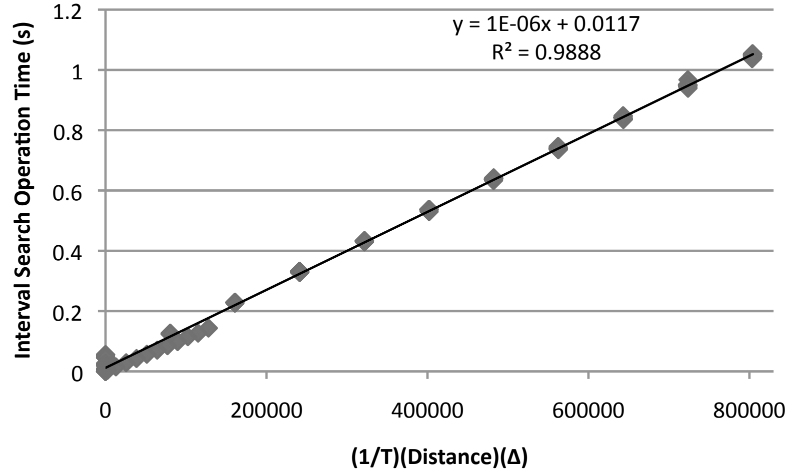

Figure 4.5: Regression of the Interval-Search component of the *CueMatching* operation in the TankSoar domain.

4.4.4 Reconstruction

Given an episodic store and a valid temporal id of an episode, we define reconstruction as the process of faithfully reproducing all working-memory elements that originally comprised that episode. This process is tantamount to an interval intersection query: collect all working-memory graph structures that started before and ended after a particular point in time.

To efficiently support this operation, we implemented a Relational-Interval Tree (*Kriegel et al.*, 2000), which is a mapping of the interval-tree data structure onto Relational-Database-Management-System (RDBMS) B+-tree indexes and SQL queries. As with the standard interval tree, intersection queries execute in time logarithmic with the number of stored intervals. Because intervals represent working-memory element changes, this growth characteristic is sub-linear with respect to the number of episodic memories and so episodic reconstruction should remain efficient even as agents persist for long periods of time.

4.4.4.1 Analysis

We developed the following predictive model for *episode-reconstruction* performance:

$$Reconstruction = RITree + Collect$$

$$RITree = (X_3)(log_2 R)$$

$$Collect = (X_4)(M)(1 + log_2 U)$$

As with the $CueMatching$ operation, we fit our models within the *TankSoar* domain with the *mapping-bot* agent. We performed 100 isolated executions of primitive operations ($RITree$ and $Collect$) on the same data collected for $CueMatching$ (10 trials, 10 time points from 100K to 1M episodes). We collected the necessary statistics (described below) for 50 episodes selected randomly (5 per 10,000 episodes through the first 100,000 episodes of execution) and performed linear regressions to fit data points.

Total time for $Reconstruction$ is the sum of two operations: $RITree$ and $Collect$. $RITree$ refers to the process of extracting pertinent intervals from the interval tree and the logarithmic dependent variable, R, refers to the total number of ranges in the Relational-Interval Tree. Our analysis in TankSoar found this model to be moderately predictive of $RITree$ performance ($R^2 = 0.7$), suggesting that performance will scale well with agent-state changes.

The $Collect$ operation refers to cross-referencing pertinent episode intervals with structural information in the Working-Memory Graph. This process depends upon the average number, U, of historically unique internal and leaf nodes, as well as the number of elements, M, comprising the episode to be reconstructed. Because episode size did not vary greatly for our evaluation agent, the dominant linear factor, M,

highlighted noise in the experimental data and thus R^2 was only 0.73. This suggests our algorithm will perform poorly if the size of working memory is large (a problem we investigate in Chapter VI).

4.5 Evaluation

We instantiated the efficient implementation as the episodic memory of Soar v9.3.2, using version 3 of the SQLite in-process relational database engine (*Hipp*, 2012) to implement our relational schemas. Our goal in this section is to understand the degree to which Soar's episodic memory supports useful operation across a variety of domains while scaling to long agent lifetimes.[2]

4.5.1 Metrics

In order to evaluate episodic-memory scaling, we measure two classes of computational-resource usage during agent runs: execution time and storage requirements.

The time it takes for Soar to complete a decision cycle dictates the rate at which it can respond to environmental change, and is thus a direct measure of agent reactivity. We instrumented Soar to directly measure time required for encoding/storing episodes, as well as performing cue matching (i.e. the time for retrieval, without reconstructing episodes in Soar's working memory). We report maximum time: whereas average time can mask momentary computation "surges," the maximum captures the agent's ability to respond under algorithmically stressful circumstances. We compare this metric to a reactivity threshold of 50 milliseconds, a response time that is sufficient for real-time control in games, robotics, and HCI tasks. We note that in practice, maximum time can be two or more orders of magnitude greater than average; however, in this evaluation, it was typically within one, and can thus also inform expectations of average performance.

[2]This evaluation was previously published in (*Derbinsky et al.*, 2012a).

Since memory becomes an important factor for long runs of agents, we measure the amount of memory used by episodic memory. We also relate this measure to the average size of and changes to working memory.

To reliably measure cue-matching timing data, we instrumented Soar to perform this operation 100 times for each cue at regular intervals across the lifetime of the agent. Storage timing data, however, only captures a single operation, and is thus noisier and we can only extract qualitative trends. All experiments were performed on a Xeon L5520 2.26GHz CPU with 48GB RAM running 64-bit Ubuntu v10.10.

4.5.2 Agent Capabilities

For each evaluation domain, we developed a specialized set of cues that implemented a set of cognitive capabilities, or high-level functionalities supported by episodic memory (*Nuxoll and Laird*, 2012). The following are the full set of capabilities that we include in this evaluation:

- **Virtual Sensing**. An agent retrieves past episodes that include sensory information beyond its current perceptual range that are relevant to the current task.
- **Detecting Repetition**. An agent retrieves past episodes that are identical (or close to identical), possibly indicating a lack of progress towards goal(s).
- **Action Modeling**. An agent retrieves an episode of performing an action, as well as one or more episodes that temporally followed. The agent reasons about task-relevant feature changes to develop a model of the consequences of its actions.
- **Environmental Modeling**. An agent retrieves an episode, as well as one or more episodes that temporally followed. The agent reasons about task-relevant

feature changes to develop a model of the dynamics of the world, independent of its own actions.

- **Explaining Behavior**. An agent retrieves an episode, as well as one or more episodes that temporally followed. The agent reasons about its prior actions, with respect to task-relevant features of state, in order to develop an explanation of its behavior.

- **Managing Long-Term Goals**. An agent retrieves goals that were initiated in the past but are not currently active, to determine if they should be active in the current context.

- **Predicting Success/Failure**. An agent retrieves an episode in which it took an action, as well as one or more episodes that temporally followed. The agent reasons about task-relevant feature changes, as well as how they relate to present task goals, in order to estimate the likelihood of success/failure if it takes the action in its present situation.

4.5.3 Word Sense Disambiguation

An important problem for any agent that uses natural language is word sense disambiguation (WSD) – the task of determining the meaning of words in context. In this domain, we extend prior work that explored the degree to which memory-retrieval bias was beneficial in this task (Chapter V; *Derbinsky and Laird*, 2011). In this formulation of WSD, the agent perceives a <lexical word, part-of-speech >pair, such as < “say”, verb >, and, after attempting to disambiguate the word, the agent receives feedback, which includes all word meanings that were appropriate in that context. To measure the benefit of long-term memory in this task, the agent perceives the corpus, in order, numerous times, and is evaluated on disambiguation-learning speed and accuracy.

For this task, we implemented an agent that represents the last n lexical-word inputs as an *n-gram*. The agent then uses a sequence of episodic cognitive capabilities to disambiguate the meaning: first, it cues episodic memory to detect a *repeated situation* (e.g. "when did I last perceive the 3-gram {Friday, say, group}?"); it then retrieves the next episode, forming an *environmental model* of feedback (e.g. "what happened when I replied 'express a supposition'?"); and then disambiguates using this prior information, *predicting future success* based upon prior experience.

We evaluated the agent using SemCor (*Miller et al.*, 1993), the largest and most widely used sense-tagged corpus (185,269 words). During its first exposure to the corpus, the agent can disambiguate 14.57% of words using a 2-gram representation of context, and 2.32% using 3-grams. During its next corpus exposure, these performance levels improve to 92.82% and 99.47%, respectively. These learning results demonstrate the benefit of flexible access to a high-fidelity store of experience.

This domain is quite small, on average requiring 234 bytes of memory to store the working-memory changes that occur during each episode. However, as with all natural-language texts, there are some words that appear more often than others in SemCor, and so this task exemplifies the effects of temporal selectivity and cue-feature co-occurrence on episodic-memory performance.

To evaluate scaling performance, we selected two 3-word phrases from the corpus and used a set of cues that represented all 1-, 2-, and 3-gram contexts for these phrases (11 cues total, as one word was shared; see Table 4.2). We ran the agent five times across SemCor, producing 4.6M episodes. We measured the performance of episodic storage and retrievals every 50K episodes.

All episodic operations met our reactivity criteria (i.e. < 50 milliseconds). Maximum storage time was essentially constant, with a maximum of 0.5 milliseconds. The maximum query time, across all 11 cues, was 22.05 milliseconds. We regressed a model that predicts cue-matching time in milliseconds, as a linear factor of the number of

Table 4.2: WSD: occurrence and cue endpoints for SemCor.

n-Gram	Occurrence	Endpoints	Proportion
{group}	1,333	1,333	$\sim 0\%$
{say}	1,005	1,005	$\sim 0\%$
{Friday}	18	18	$\sim 0\%$
{well}	150	150	$\sim 0\%$
{be}	8,400	8,400	$\sim 0\%$
{say, group}	6	2,338	0.21%
{Friday, say}	1	1,023	0.55%
{Friday, say, group}	1	2,356	1.27%
{be, say}	69	9,405	0.07%
{well, be}	27	8,550	0.17%
{well, be, say}	1	9,555	5.16%

interval endpoints walked ($R^2 > 0.999$): $0.0024x + 0.0647$. This model predicts that retrieval time, in this task, is dependent almost exclusively on interval walking. Thus, we can estimate scaling limits by computing the number of endpoints walked when the function value equals 50 milliseconds ($x = \lfloor [50 - 0.0647]/0.0024 \rfloor = 20,806$ endpoints). From this analysis, we conclude that if we assume one word per episode, this episodic-memory implementation can reactively perform cue matching that examines $(20,806/185,269) = 11.23\%$ of SemCor.

We now examine how this scaling capacity compares to the space of possible cues, and the evaluation cues we used in this task. In SemCor, only two lexical words occur more frequently than in 1% of inputs: "be" (4.53%) and "person" (3.61%). Since these frequencies are far below the threshold of 11.23%, we conclude that this episodic-memory mechanism can reactively respond to any individual feature as a cue. However, as cue size increases, the number of potential endpoints to walk increases additively with each word, while co-occurrence frequency, the number of times the n-gram occurs within the corpus, can only stay constant or decrease. For instance, consider the following two phrases used for our cue evaluation: {Friday, say, group} and {well, be, say}. The endpoint and co-occurrence frequency data of all 1-, 2, and 3-grams of these phrases is presented in Table 4.2, where the final column is

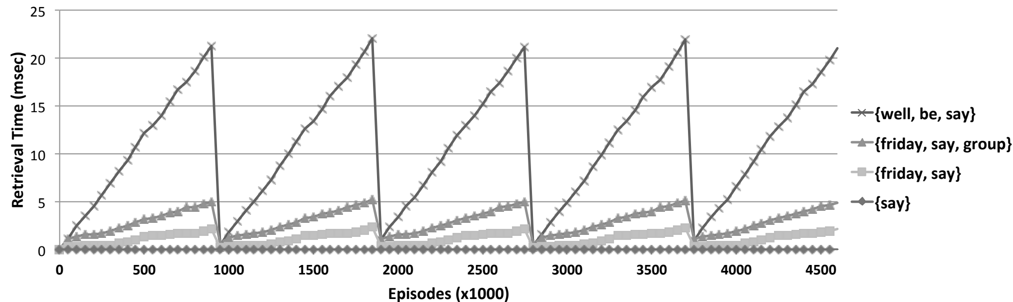

Figure 4.6: WSD: retrieval-time data versus encoded episodes, controlling for feature co-occurrence.

(endpoints/occurrence) divided by the size of SemCor, which estimates the proportion of SemCor likely to be examined for each cue (assuming uniform distribution of occurrence). For 1-grams, this mechanism achieves constant-time cue matching, independent of these data, since it concludes cue matching after the first match.

For those n-grams with a co-occurrence of 1, cue-matching time exhibits saw-tooth patterns, where peaks are once-per-corpus exposure, since the number of endpoints to examine increases until the n-gram is re-encountered. This data is presented in Figure 4.6, which plots retrieval time versus encoded episodes, with the 1-gram {say} is plotted as a baseline. This chart clearly indicates the correlation between search distance and retrieval time (see predictive model of Interval Search in Section 4.4.3.1).

For non-zero co-occurrence, we see more frequent, non-uniform heights/slopes and in the data, as the n-grams are encountered through the corpus, which relates to temporal selectivity. Figure 4.7 plots retrieval time versus encoded episodes of the n-grams that include the word "say" and have non-zero co-occurrence, with {say} provided again as a baseline. This evaluation shows that our episodic-memory mechanism can perform this task reactively for 1-grams, 2-grams, and 3-grams, as SemCor proportion for all cues of these lengths is below 11.23%. However, of the more than 184,000 distinct 4-grams in this corpus, there are 368 that require examining more than 11.23% of SemCor, and thus a 4-gram is the scaling limit.

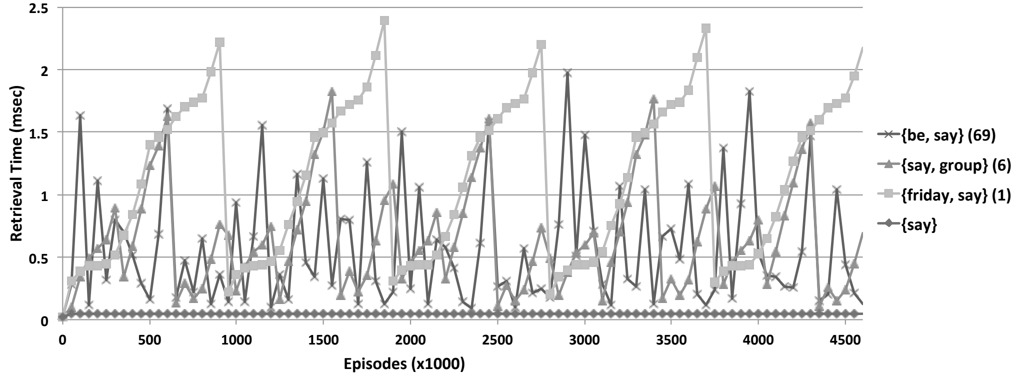

Figure 4.7: WSD: retrieval-time data versus encoded episodes, controlling for temporal selectivity.

4.5.4 Generalized Planning

In WSD, temporal selectivity and co-occurrence of cue features was the primary factor affecting performance, whereas cue size and structure had little effect. To evaluate cue complexity, we extended prior work that used episodic memory as a source of action-modeling knowledge for planning (*Xu and Laird*, 2010).

In this evaluation, we used 12 planning domains, common in competitions (*Logistics*, *Blocksworld*, *Eight-puzzle*, *Grid*, *Gripper*, *Hanoi*, *Maze*, *Mine-Eater*, *Miconic*, *Mystery*, *Rockets*, and *Taxi*) and made 44 problem instances by varying domain parameters (e.g. number of blocks in Blocksworld). These domains were originally expressed in the Planning Domain Definition Language (PDDL) and were automatically converted into Soar rules that encode the states in which the agent can select actions (*operator proposal*), as well as the consequences of those actions (*operator application*). During execution, the agent's working memory captures a set of objects and relations and, at each episode, the agent randomly selects an action, exploring the state space over time.

Our first experiment explored whether episodic memory could *detect repeated states*. For each problem instance, we extracted a random problem state as our evaluation cue. We then ran the agent for 50K episodes, and measured performance every

1K episodes as it explored the state space. This experiment evaluates the episodic memory while stressing the dimension of cue structural selectivity: in these domains, the cue is relatively large and the agent state is structurally homogenous, and thus cues match multiple structures in most other episodes.

The episodic memory reactively stored episodes in all problem instances (maximum < 12.04 milliseconds). Memory consumption in each domain was strongly correlated with the number of working-memory changes ($R^2 = 0.86$): storage ranged from 562 bytes per episode to 5454, averaging 1741. Using the full 48GB of RAM on our evaluation computer, we could thus store between 9 and 91 million episodes, with more than 29 million on average. In summary, storage was not a scaling concern in this set of domains.

Of the 44 problem instances, there were 12 in which cue matching remained reactive for the full 50K episodes, all of which were instances of the Miconic, Maze, Hanoi, and Gripper domains (see Figure 4.8). These problem instances did not exhibit growth in their cue-matching time, while the remaining problem instances grew rapidly and became unreactive in fewer than 10k episodes. When we explored the data for explanatory factors, we found that retrieval time within each domain strongly correlated with the number of episodes searched and working-memory size ($R^2 = 0.85$). The 12 problem instances that did scale had the smallest average working-memory sizes, as well as relatively small state search spaces (yielding small, bounded interval searches). For example, in Figure 4.8, the "hanoi__4" domain had one of the largest working memories (234 on average) and had, by far, the largest state space: on average, retrievals searched 784 episodes, and 3,505 at maximum, versus "maze6x6," the next largest, which required 228 on average and 1,215 maximum. The remaining instances were either too structurally unselective, too temporally unselective (due to a large state space), or both. These results characterize an upper bound in cue complexity for reactive retrievals.

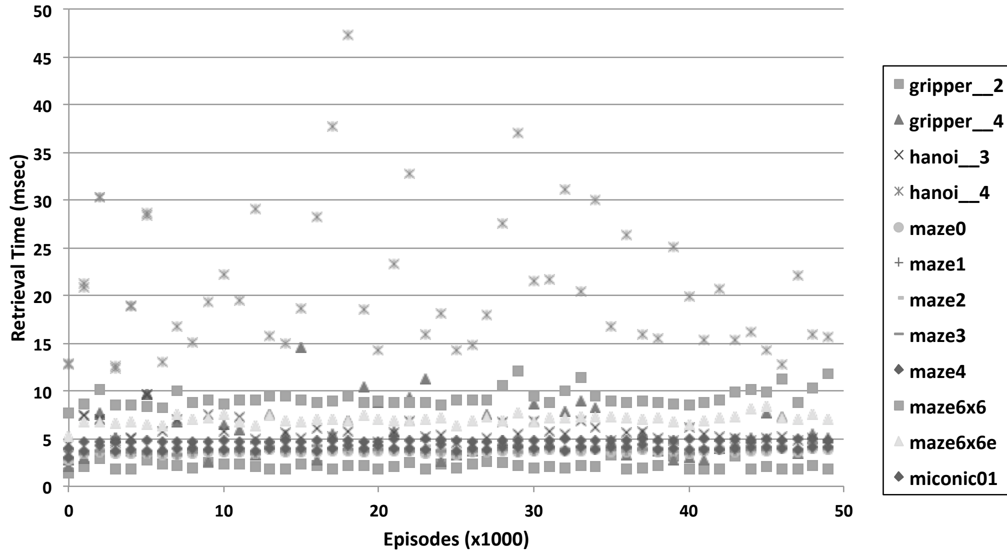

Figure 4.8: Planning: retrieval-time data versus encoded episodes for the detecting-repeated-states experiment (reactive domains).

Our second experiment explored whether episodic memory could be used to detect *analogous* states. We used the same setup as in the previous experiment, but removed all grounded features in the evaluation cues. However, this had the effect of making the cues less structurally selective (i.e. each cue feature could match more structures when compared to an episode). As a result, the episodic memory could not scale on any problem instance, primarily due to frequent and expensive structural matching. These findings suggest that the episodic-memory mechanism is not appropriate for direct analogical mapping.

Our final experiment explored whether episodic memory could be used to detect analogous states *if* the agent had knowledge of important schemas at the time of encoding. This experiment relates to prior work showing that experts are able to encode memories that can be relationally retrieved (*Gentner et al.*, 2009). We encoded the cues from our second experiment, those without grounded features, as rules that would place a flag in working memory whenever the pattern appeared, a feature episodic memory would automatically encode and could be queried for di-

rectly. The episodic-memory mechanism could scale to 50K episodes in all problem instances given this task formulation (maximum $<$ 0.08 milliseconds), suggesting agents can perform limited analogical reasoning over large stores of prior experience, while remaining reactive, by joining task-dependent recognition knowledge with a task-independent episodic memory.

4.5.5 Video Games and Mobile Robotics

The previous evaluations focused on specific aspects of episodic-retrieval cues: WSD stressed temporal selectivity and feature co-occurrence, while the planning domains stressed structural selectivity. We found that the episodic-memory mechanism has scaling limits that depend on domain structure and dynamics, knowledge representation, and cues. Here we examine the degree to which these limitations apply to domains in which agent actions impact its future perceptions of the world. We describe the domains, and then present combined results.

4.5.5.1 TankSoar

TankSoar (see Figure 4.9) is a video game that has been used in evaluating numerous aspects of Soar, including episodic memory (*Nuxoll and Laird*, 2012). In TankSoar, the agent controls a tank and moves in a discrete 15x15 maze. The agent has numerous sensors, including path blockage and radar feedback, and it can perform actions that include turning, moving, and controlling its radar. The agent we use, *mapping-bot*, explores the world and populates an internal map, stored in working memory.

This task is interesting for episodic-memory evaluation due to a large working memory with relatively few changes. However, most perceptual structures change frequently and many are highly selective, both temporally and structurally.

We used 15 cues in TankSoar (see Appendix D for the full set), which implemented

Figure 4.9: TankSoar: domain screenshot. The agent (the tank marked with a red dot) is using its radar to sense a "battery," which is a source of energy (a limited resource).

virtual-sensing, detecting-repetition, and action-modeling capabilities. For example:

1. "When did I last sense a missile pack on my radar?"
2. "When was I last at this (x, y) position on my map?"
3. "What happened last time I rotated left and turned on my radar while I was blocked in the forward direction?"

Cues that referred to a map cell (e.g. #2) were structurally unselective, as they could refer to any of the 225 entries. The temporal selectivity of cues relating to perceptual structures was typically reduced as the cue size increased, due to non-overlap in feature co-occurrence (e.g. in #1 there were episodes when the agent used the radar,

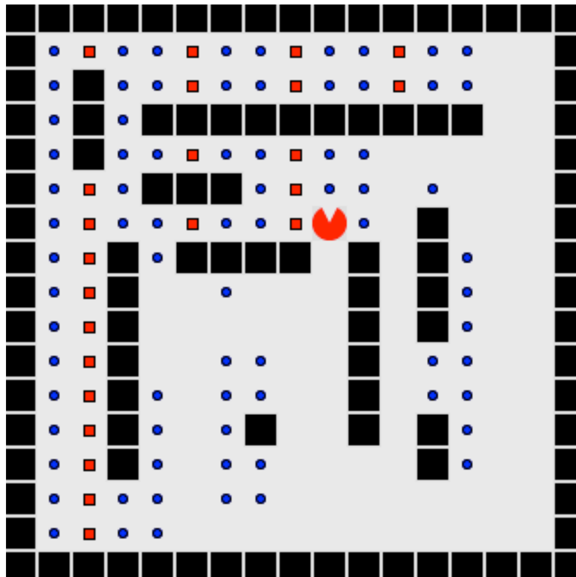

Figure 4.10: Eaters: domain screenshot. The agent (the red *Pac-Man*) is now in a position to move in one of four cardinal directions, where it will consume either "normal" food (north and east), "bonus" food (west), or no food (south). Black squares represent walls, which the agent cannot penetrate.

but did not sense a missile pack). We ran *mapping-bot* for 3.5M episodes, which is >48-hours of simulated real-time (SRT: 50 milliseconds/episode), and measured performance every 50K.

4.5.5.2 Eaters

Eaters (see Figure 4.10) has also been used in previous episodic-memory evaluations. Eaters is a video game, similar to PAC-MAN, where the agent controls an "eater," which moves through a 15x15 grid-world, eating different types of food. The agent senses a 2-cell radius, and can move in any of the four cardinal directions. The agent we use, *advanced-move*, prioritizes movement based on food types.

This task is interesting as a contrast to TankSoar. The agent's working-memory size is drastically smaller, but changes are comparable. We used 7 cues that exemplified virtual sensing, detecting repetition, action modeling, and explaining behavior (see Appendix D for the full set). For instance: "What happened the last time there

Figure 4.11: Infinite Mario: domain screenshot. The agent (Mario) has avoid the enemy "goomba" (left of Mario) and is pursuing coins (right of Mario) that will supply it reward.

was normal food to the east of me and bonus food to the west of me?" Examining the episode following this retrieval supports the agent explaining its own preferences regarding the relative desirability of these food types, informing predictions of its own future decisions. The agent state is sufficiently simple such that no evaluation cue was unselective, either structurally or temporally. We ran *advanced-move* for 3.5M episodes (>48 hours, SRT) and measured every 50K.

4.5.5.3 Infinite Mario

Infinite Mario (see Figure 4.11) is a video game used in the 2009 Reinforcement-Learning (RL) Competition and is based on Super Mario. The allocentric visual scene comprises a two-dimensional matrix (16x22) of tiles and the agent can take actions that include moving, jumping, and increasing speed. We use an agent that applies an object-oriented representation and hierarchical RL to quickly improve performance in the task (*Mohan and Laird*, 2011) and collect data on game level 0.

Several aspects of this game are interesting for evaluation. The working memory is large and contains a variety of representational patterns, including flat features that are both symbolic (e.g. Mario is "small," "big," or "fiery") and real-valued (e.g. distance to enemies); hyper-edges (e.g. rows in the visual scene); and relational structures (e.g. relating hierarchical state representations to perception). Also, due to side-scrolling, a relatively large percentage of the visual scene changes between episodes, stressing temporal contiguity.

We evaluated 14 virtual-sensing and action-modeling cues (see Appendix D for the full set). For example, the following cue combines perceptual features with those derived from task knowledge: "What did I do when last I encountered a winged, downward-flying 'Goomba' that was a threat?" Cues that virtually sensed visual-scene cells were structurally unselective. We ran the agent for 3.5M episodes (>48 hours, SRT) and measured performance every 50K.

4.5.5.4 Mobile Robotics

For this evaluation, we used an existing mobile-robotics platform that has been applied to simulation and physical hardware (*Laird et al.*, 2011b). The agent perceives both physical perception data, including real-valued abstractions of laser range-finder data, as well as symbolic representations of objects, rooms, and doorways. The task is to explore a building with 100 offices (see Figure 4.12), and then execute a fixed-patrol pattern. While performing these tasks, the agent builds an internal map, which it uses for path planning and navigation.

We evaluated 6 cues for virtual sensing and goal management (see Appendix D for the full set). Consider the following cue: "When was my desired destination doorway #5?" The agent could examine episodes that followed to recall progress made towards that goal. However, as the agent accumulated more distinct goals, this cue became less temporally selective.

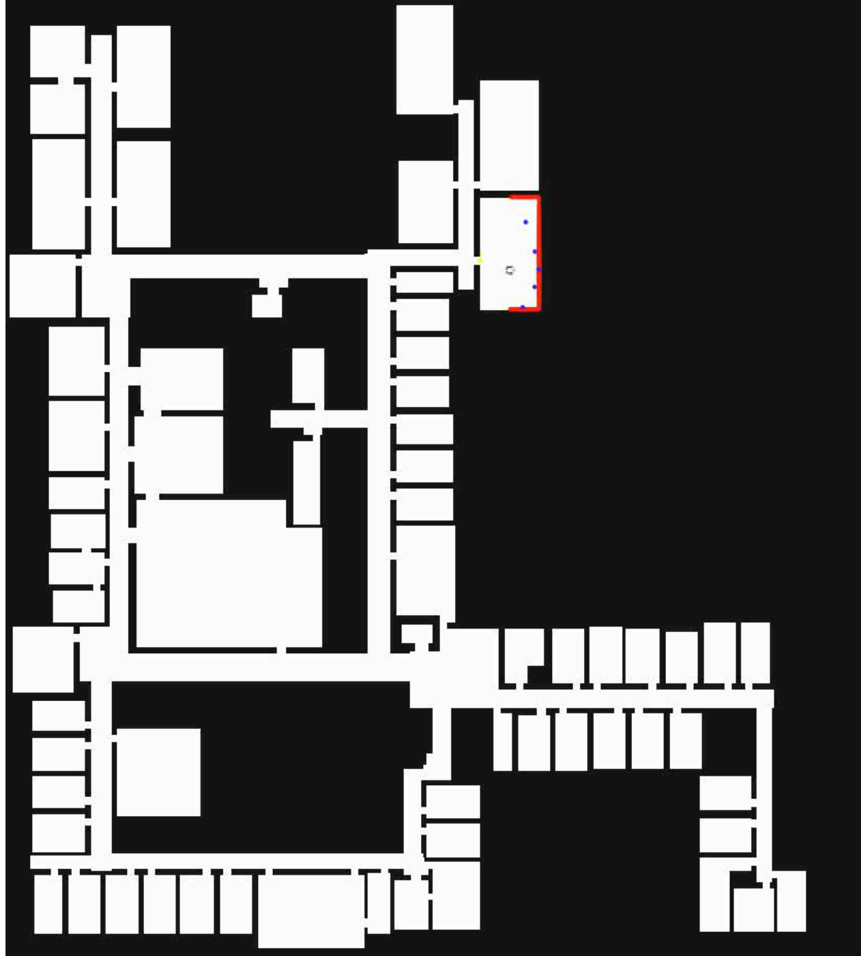

Figure 4.12: Mobile Robotics: domain map. The robotic agent is located in the Soar lab (3844 BBB) and is using LIDAR (red dots) to sense the room. The Soar agent receives maximal distance data at five points (blue dots) in its 180° sensory range.

We ran the agent in simulation for 12 hours of real time, which totaled 108.6 million episodes. Unlike the video games, which run in lock-step with discretized time, this agent runs asynchronously with a dynamic simulated environment, as would a real robot. The agent spends most of its time moving and very little changes in its perception of the world from episode to episode. We expect that this type of environment is representative of real-world agents, whereas the video games are, to a greater extent, anomalous. We measured performance every 300K episodes, which amounts to about once every 2 minutes.

Table 4.3: Summary of empirical results for episodic memory in video-games and mobile-robotics domains.

	Storage		Cue Matching (Max. Time in msec.)	
	Max. Time (msec.)	**Avg. Bytes/Episode**	**High Selectivity**	**Low Selectivity**
TankSoar	18.66	1,035	4.77	18.31
Eaters	1.39	813	0.71	–
Infinite Mario	55.01	2,646	1.66	40.43
Mobile Robotics	3.17	113	0.75	27.50

4.5.5.5 Results

The left half of Table 4.3 presents storage results, grouped by domain. The episodic-memory mechanism stored episodes in less than 50 milliseconds for all domains except Infinite Mario, where infrequent spikes in perceptual changes, caused by Mario dying and restarting the level, defied the temporal-contiguity assumption. The storage cost across domains correlated with working-memory changes ($R^2 > 0.93$).

The right half of Table 4.3 presents cue-matching results, grouped by domain and cue selectivity (temporal or structural). The episodic-memory mechanism maintained reactivity across all domains. With one exception, retrieval time did not meaningfully increase with time. The growth rate for goal management in mobile robotics (see Figure 4.13) depended upon the properties of the robot's mission: when behavior shifted from exploration to patrol ($\sim$10 million episodes), new goal locations were encoded less frequently, and thus temporal selectivity decreased at a smaller rate. If the agent had to explore a much larger building, and thus the original growth rate had continued, the cue-matching time would have grown beyond 50 milliseconds after 34 million episodes (fewer than 4 hours of real time). In the Beyster building, however, the agent can maintain reactivity for longer than it can store episodes in main memory: the cue-matching time will grow beyond 50 milliseconds after nearly 634 million episodes (longer than 3 days of continuous real-time operation), whereas

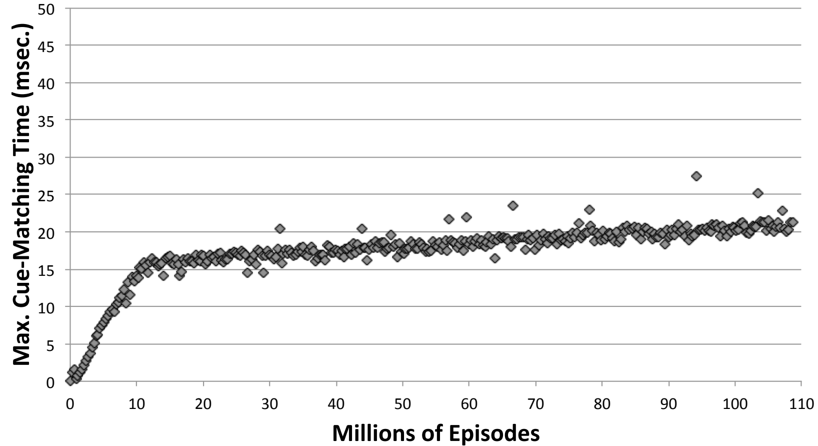

Figure 4.13: Mobile Robotics: timing data for goal-management cue.

memory growth will exceed 48GB of memory after about 437 million episodes (just over 2 days).

4.6 Discussion

In this chapter, we presented and evaluated techniques to enhance intelligent agents with episodic memory. Our memory model represents episodes as connected digraphs, a sufficiently general representation (R3) for effective use in a variety of tasks. It also incorporates task-independent (R6) processes to incrementally encode and store new agent experiences (R1), as well as support flexible retrievals (R5) that, in practice, scale to long agent lifetimes (R4). We implemented this mechanism in the Soar cognitive architecture and evaluated it in a variety of problem domains, including word sense disambiguation, generalized planning, video games, and mobile robotics. We showed that while the algorithms are not immune to properties of domains and cues that negatively affect performance (e.g. low temporal selectivity, low structural selectivity, and low feature co-occurrence), the mechanism does support many cues that agents can apply to support useful cognitive capabilities (R2).

4.6.1 Future Work

In order for an episodic-memory mechanism to fulfill the requirements imposed by generally intelligent agents (Chapter II), computational-resource utilization must be bounded. The current implementation has sources of unbounded memory and computation growth, scaling primarily with agent-state changes over long lifetimes. To bound memory usage, it will be necessary to investigate methods to effectively forget episodic knowledge, possibly maintaining higher-level abstractions over episodic "events." To bound computation time, some benefits may come from time-slicing or parallel algorithms, but ultimately it is likely that our memory model will require a relaxed specification of retrievals (i.e. heuristic search). However, as with all memory research, it will be important to balance scaling (R4) with agent task performance (R5), and thus much more work must be done to develop and evaluate agents that use episodic memory in a variety of problem domains. This will entail research into general methods for integrating episodic retrievals with agent reasoning in known tasks, as well as how strategies for agents to learn to utilize episodic knowledge in novel situations.

Another direction of research will be to investigate how an episodic-memory mechanism integrates with other components and processes in a general cognitive architecture. We have done some initial work to investigate how the Working-Memory Graph can serve as a frugal source of heuristic knowledge for recognition judgements (*Li et al.*, 2012). Another fruitful avenue will be to explore how historical information in episodic memory can inform *consolidation*, or automatic encoding, of semantic knowledge. Finally, our retrieval mechanism currently biases only towards recency of information, a crude proxy for the relevance/usefulness of information. It will be interesting to investigate the degree to which meta-data from other cognitive mechanisms and processes (e.g. emotional appraisals; *Marinier et al.*, 2009) can improve the robustness of this search across a variety of problem domains.

CHAPTER V

Semantic Memory

This chapter documents our progress in understanding the computational challenges involved in extending generally intelligent agents with a task-independent semantic memory. We begin with a motivational description of semantic-memory systems, including a small amount of psychological background (Section 5.1); then discuss related work (Section 5.2); continue to our functional specification of a semantic-memory model (Section 5.3); describe data structures and algorithms that efficiently implement the mechanism (Section 5.4); evaluate the mechanism, as implemented within the Soar cognitive architecture (Section 5.5); and conclude with a summary and discussion of future work (Section 5.6).

5.1 Motivation

Tulving characterized semantic memory as a mechanism that captures declarative facts about the world, independent of the context in which they were originally learned (*Tulving*, 1972, 1983). Intuitively, semantic knowledge encodes what an agent "knows," whereas episodic knowledge (Chapter IV) represents an autobiographical stream of experience that the agent "remembers."

In context of the memory requirements for generally intelligent agents, incorporating semantic memory in a cognitive architecture contributes to the support of diverse,

comprehensive learning (R2), by incrementally (R1) providing an agent (R3) rich access to general knowledge that is re-usable over time, independent of how situations differ temporally, spatially, or with respect to other contextual distinctions relative to the agent's state and goals. However, as with episodic memory, scaling (R4) effective access (R5) to this knowledge in a task-independent fashion (R6) over long agent lifetimes poses a significant challenge.

Agents that are enhanced with a scalable semantic memory can access large stores of information in a variety of situations. For instance, agents that communicate using natural language require flexible access to a lexicon, such as WordNet (*Miller*, 1995). Additionally, agents that work on a variety of problems in diverse domains benefit from conceptual and ontological information about the world, provided by such knowledge bases as the SUMO upper ontology (*Niles and Pease*, 2001) and Cyc (*Lenat*, 1995). If the agent has sufficient processes and task knowledge to make use of these large information stores, there is evidence that semantic memory may serve to improve the agent's ability to scale reasoning to complex problems wherein state and feature spaces are very large, such as via generalization (e.g. *Sanner et al.*, 2000), heuristic inference (e.g. *Anderson and Schooler*, 1991; *Schooler and Hertwig*, 2005), and partitioning knowledge into local/short-term and distal/long-term (e.g. *Newell*, 1990). We return to these cognitive capabilities in Section 5.5, where we evaluate our semantic-memory implementation.

5.2 Related Work

Unlike episodic memory, incorporation of semantic-memory mechanisms and knowledge bases has been an area of active research in the cognitive-architecture community (*Langley et al.*, 2009). For example, Icarus (*Langley et al.*, 2004; *Langley and Choi*, 2006) has a hierarchical organization of long-term concepts that is accessible to deductive reasoning; Companions (*Forbus and Hinrichs*, 2006; *Forbus et al.*, 2009)

informs analogical reasoning with a large knowledge base of ontological and domain knowledge, initialized with the contents of ResearchCyc (*Lenat*, 1995); and the LIDA framework (*Franklin and Patterson*, 2006) posits a declarative memory, populated from consolidated episodic knowledge, which can supplement the current situated awareness. The ACT architecture (*Anderson*, 1983) was the first to have a long-term declarative memory and ACT-R (*Anderson et al.*, 2004) implements properties, derived from the rational analysis of memory (*Anderson and Schooler*, 1991), of how to retrieve useful items from long-term memory. The ACT-R model has been extensively used to model a rich set of psychological phenomena, such as the Fan effect (*Anderson and Reder*, 1999), category learning (*Anderson*, 1991), and list memory (*Anderson et al.*, 1998).

There has been very little work, however, that explores how to support effective semantic-memory retrievals for real-time agents as knowledge bases grow large. As an illustrative example, the Air Force Research Laboratory applied ACT-R to develop a Synthetic Teammate capable of functioning as the Air Vehicle Operator (AVO), or pilot, in a 3-person team task simulation of an Uninhabited Aerial Vehicle (UAV) performing reconnaissance missions (*Ball et al.*, 2010). A crucial component of the system was language comprehension and so the system integrated a large subset of the WordNet lexicon (*Miller*, 1995) into the model's long-term declarative memory. By itself, the language comprehension component pushed the scale of the long-term memory beyond the capacity of the ACT-R data storage and access mechanisms, and thus the model either ran many times slower than real time, or had compromised stability, depending upon the proportion of WordNet that was made available to the model (*Douglass et al.*, 2009). To extend the capabilities of the ACT-R declarative memory, *Douglass et al.* developed a semantic-memory mechanism using the PostgreSQL relational database management system. While their semantic-memory module did support 40-milliseconds retrievals for a set of cues, the authors did not

characterize the computational profile of their retrieval mechanism, did not evaluate their approach beyond a single domain, and did not support any retrieval biases in the case of ambiguous cues (e.g. base-level activation).

5.3 Functional Specification

Abstractly, we define a semantic-memory store as a set of *elements*.[1] A semantic element is composed of a *bias* value and a set of symbolic *augmentations*. For example, consider the following example semantic store, in which upper-case English letters identify elements, lower-case Greek letters represent augmentations, and the numbers in square brackets represent bias values:

A [1.41]: { α, β, ϵ, ϕ }

B [1.73]: { α, ϵ }

C [3.14]: { γ }

D [2.72]: { γ, ϕ }

We define a symbolic retrieval *cue* as a set of symbols corresponding to the set of augmentations of a particular semantic store, such as { α, ϵ }. The set specifies the augmentations that an element *must* contain.

Given a semantic store and a cue, we define the *result* of a semantic retrieval to be a single element from the store, including all augmentations, that satisfies the constraints of the cue and has the maximal bias value. Thus, given the example cue { α, ϵ }, the result is B: while both A and B satisfy the cue, the bias value of B (1.73) is greater than that of A (1.41).

[1]This specification was previously published in (*Derbinsky et al.*, 2010).

5.3.1 Integration with Soar

In Soar, working memory is composed of a set of symbolic triples, or WMEs. The set of WMEs that share the same first symbol, or "identifier," are termed an "object," and the set of individual WMEs sharing that identifier are termed "augmentations" of that object. Thus, an identifier in working memory is a natural mapping to a semantic-memory element, where object augmentations map to semantic-memory augmentations.

To add an element to semantic memory, agent task knowledge (represented as rules) constructs a storage command: a reference to an identifier in working memory. Semantic memory copies the working-memory augmentations and associates them with a semantic-memory element. Furthermore, the working-memory identifier now refers directly to the semantic-memory element, such that future storage of the working-memory object overwrites the contents of semantic-memory augmentations. Note that while Soar's working memory forms a single connected graph, semantic memory is composed of multiple, possibly disconnected graphs.

To retrieve an element, agent task knowledge (represented as rules) constructs a semantic cue: an object in working memory. Semantic memory then interprets the augmentations of the object as the cue and attempts to find a result. If found, semantic memory reconstructs the semantic-memory element in working memory at a pre-specified region. Given a working-memory identifier that refers to a semantic-memory element, agent task knowledge can also request that the current semantic-memory augmentations overwrite working-memory augmentations (which may not be equivalent if changes have occurred to a working-memory object since the most recent storage).

5.3.2 Comparison to ACT-R

The ACT-R declarative memory is widely used within the cognitive-architecture and cognitive-modeling research communities. Thus, here we compare our abstract semantic-memory formulation to the ACT-R declarative module.

In ACT-R, declarative knowledge is encoded as a set of *chunks*, each of which is a collection of labeled *slots* that have values. Each chunk is given a *type* via an ISA designation, which dictates the set of slots it has. To retrieve declarative knowledge, a production rule issues a request to the declarative module by populating the declarative buffer with a set of constraints, including a test and a slot-value pair. One type of test is a *positive* test ("+"), which is interpreted as meaning the slot-value pair *must* exist in retrieved knowledge. Another is a *negative* test ("-"), which means the slot-value pair must *not* exist in retrieved knowledge. The DM module also supports non-symbolic comparison tests (e.g. $\geq$, $<$, etc). Given this request, the ACT-R DM module searches the store for matching chunks. If any are found, the module, indicates a successful retrieval, selects a result from amongst the candidates chunks, and reconstructs it in the appropriate buffer. The module also supports the use of non-symbolic activation to bias selection amongst candidate chunks, functionality that is used in many cognitive models. The most commonly utilized activation models are *base-level*, which incorporates the history of past retrievals, and *spreading*, which incorporates retrieval context.

We now map the ACT-R DM to our abstract formulation. First, without loss of generality, we interpret the chunk type as a slot-value pair (slot label "ISA" and value equivalent to the type). Next, since we are considering qualitative matching (equality is defined as equivalent symbols), each distinct slot-value pair can be equivalently represented as a single, composite symbol (by concatenating the slot label and value with a unique separating character, such as "ISA:typeName"). Since slot-value pair order is arbitrary, a chunk instance can be equivalently represented as a set of [com-

posite] symbols. In ACT-R, all chunks of a given type must contain values for the same set of slots and a chunk type can only have one slot of a given label; without loss of generality, we eliminate both of these constraints. Given the analysis above, a chunk maps to a semantic-memory element, and slot-value pairs to augmentations.

We apply a similar analysis to DM retrieval requests, with distinct slot-value pairs compressed to a single composite symbol. If we require that equivalent slot-value pairs in chunks and retrieval requests resolve to the same composite symbols, then the set of positive tests form the cue. In Appendix E, we discuss issues involving, and preliminary work on, negative tests; however, we have not implemented support for negative cues in Soar and do not discuss them further. Our model only supports qualitative matching, and thus does not support non-symbolic cue tests. However, outside of these differences, the symbolic ACT-R DM retrieval interface is an instance of our problem formulation and thus results from our work, though implemented in Soar, extend to ACT-R models, and any other system that can be similarly mapped.

5.4 Efficient Implementation

In this section, we discuss indexing structures and processes to efficiently support semantic-memory functionality, even as the number of elements grows to be very large.[2] We first make and justify an assumption regarding the contents of semantic memory: most elements have a small number of augmentations. Given this assumptions, the storage and reconstruction operations of semantic memory are computationally trivial, and thus will not be discussed further. Instead, the remainder of this section will address the problem of supporting effective and efficient retrievals. However, prior to getting lost in the weeds, we will first consider what is meant by *efficient support* with respect to our problem formulation. Appendix F details the knowledge representation as a set of relational schemas, while appendix G provides a

[2]This implementation was previously published in (*Derbinsky et al.*, 2010).

concise description of the algorithms.

5.4.1 Assumption: Small Element Cardinality

Core to our semantic-memory implementation is the assumption that for most elements, augmentation cardinality is small. To validate that this assumption is reasonable for real data sets, we studied three large, commonly used knowledge bases (KBs): SUMO (*Niles and Pease*, 2001); OpenCyc, a subset of Cyc (*Lenat*, 1995); and WordNet (*Miller*, 1995). For each KB, we extracted the number of features of each named entity (see Appendix H for feature CDFs). Each distribution was unimodal, with more than 90% of elements having fewer than 50 elements, and exhibited strong right skew, suggesting that while the there was a common range of feature size for most elements, there existed rare cases with exceptionally large cardinalities.

5.4.2 Contextual Meaning of Efficient Support

As a baseline, consider a naïve retrieval mechanism that iterates through the semantic store, comparing each element to the cue, and returning the first valid result, if one exists. To understand the costs, we define E as the set of elements in the store, and a as the average number of augmentations per element, and C as the cue. Sets surrounded with vertical bars, such a $|E|$, refer to the cardinality, or number of items contained in the set.

Assuming no specialized indexing, the memory cost of the baseline mechanism grows with the product of the number of elements and the average augmentation cardinality ($a \cdot |E|$). In the worst case, the baseline mechanism must traverse all of this memory for each cue element, and thus the time cost multiplies by the size of the cue ($a \cdot |E| \cdot |C|$). In context of large semantic-memory stores, it is likely that $|E|$ will dominate a and $|C|$, and thus memory and retrieval costs will scale linearly with the number of elements in the semantic memory.

Memory, though not unlimited, is generally considered cheap and plentiful, while time is expensive and limited, and thus our goal is to minimize retrieval time, possibly at the cost of memory. Thus we pose efficient support for semantic retrievals as sub-linear in the number of elements in the semantic memory, $|E|$, while remaining linear in memory.

5.4.3 Cue Matching

The cue for semantic-memory retrievals is a non-empty set of augmentations that a resulting element *must* contain. To assist in our analysis, we define R_c as the elements that contain an augmentation c and, accumulated over all c in C, R to be the bag of candidate elements (which may contain duplicates, if an element contains more than one augmentation, c, in C).

Before presenting our mechanism, we note that the retrieval problem is a constrained form of a subset query on set-values, which has been widely studied in database and information retrieval (IR) communities (*Terrovitis et al.*, 2006). In its general form, the worst-case time cost is known to be linear in the sum of the number of candidate elements for each cue augmentation, $|R|$, though clever indexing methods have shown massive average-case improvements in real-world data.

Our approach includes two components: indexing and query evaluation.

5.4.3.1 Indexing

We supplement the semantic memory with an inverted table of semantic-memory elements (*Zobel and Moffat*, 2006), where augmentations are the "terms" and elements are the "documents." As is common, we associate with each distinct augmentation the cardinality of its associated element list. We also impose an ordering on each element list, sorted in descending order based upon the element bias values; this sorting facilitates iteration of retrieval candidates during query evaluation.

To support efficient updates to this data structure, we introduce a threshold parameter, τ, which represents a "small" value of augmentation cardinality. If the augmentation cardinality of an element is less than τ, the bias value of the element is represented explicitly within the element list of each distinct augmentation. Otherwise, we associate with the element a special "infinite" bias value (∞), which forces it to the front of the sorted element list. As we describe shortly, this bias-representation dichotomy balances computation at the time of index updates (for small augmentation cardinality) and cue evaluation (large augmentation cardinality). For example, given the following semantic memory:

A [1.41]: { α, β, ϵ, ϕ }

B [1.73]: { α, ϵ }

C [3.14]: { γ }

D [2.72]: { γ, ϕ }

Our supplementary index would contain the following information, for a value of $\tau = 3$, where the number in parentheses is the element-list cardinality:

α (2): [A=∞, B=1.73]

β (1): [A=∞]

γ (2): [C=3.14, D=2.72]

ϵ (2): [A=∞, B=1.73]

ϕ (2): [A=∞, D=2.72]

We formalized this representation as a set of relational schemas, which are detailed in Appendix F. We now analyze the update operations with respect to a semantic-memory element, e, and the supplementary index:

- **Add an Augmentation**: $|e| \neq (\tau - 1)$

  If the cardinality change does not violate τ, then add to element list.

  1. Hash to the appropriate element list: $O(\log[\text{distinct augmentations}])$
  2. Insert in sorted element list: $O(\log[|\text{list}|])$
  3. Update element-list cardinality: $O(1)$

- **Add an Augmentation**: $|e| = (\tau - 1)$

  If the cardinality change violates τ, add to element list and convert existing augmentations to ∞.

  1. Hash to the appropriate element list: $O(\log[\text{distinct augmentations}])$
  2. Prepend to sorted element list: $O(1)$
  3. Update element-list cardinality: $O(1)$
  4. For each augmentation, $a \in e$: $O(|e|)$

     a) Remove from sorted element list (old bias value): $O(\log[|\text{list}|])$

     b) Prepend to sorted element list: $O(1)$

- **Remove an Augmentation**: $|e| \neq \tau$

  If the cardinality change does not violate τ, then remove from element list.

  1. Hash to the appropriate element list: $O(\log[\text{distinct augmentations}])$
  2. Remove from sorted element list: $O(\log[|\text{list}|])$
  3. Update element-list cardinality: $O(1)$

- **Remove an Augmentation**: $|e| = \tau$

  If the cardinality change violates τ, then remove from element list and convert existing augmentations to true bias value.

  1. Hash to the appropriate element list: $O(\log[\text{distinct augmentations}])$
  2. Remove from sorted element list: $O(\log[|\text{list}|])$
  3. Update element-list cardinality: $O(1)$
  4. For each augmentation, $a \in e$: $O(|e|)$
     a) Remove from sorted element list (∞): $O(\log[|\text{list}|])$
     b) Add to sorted element list (true bias value): $O(\log[|\text{list}|])$

- **Change Bias Value of Element**: $|e| < \tau$

  If the cardinality is less than τ, update the bias value of each augmentation and re-sort each element list.

  1. For each augmentation, $a \in e$: $O(|e|)$
     a) Hash to appropriate element list: $O(\log[\text{distinct augmentations}])$
     b) Remove from sorted element list (old bias value): $O(\log[|\text{list}|])$
     c) Insert in sorted element list (new bias value): $O(\log[|\text{list}|])$

- **Change Bias Value of Element**: $|e| \geq \tau$

  If the cardinality is greater than or equal to τ, all augmentations already reflect the ∞ bias value and thus do not require updates.

  1. No action necessary

An important outcome is that the only linear component in these operations is that which iterates over the augmentations of an element (e.g. when changing bias value of an element). Since this iteration is bounded in the element cardinality, and we only perform this iteration when element cardinality is a small value, all

index-update operations are efficient independently. Furthermore, if our assumption regarding small element cardinality holds, the number of augmentation changes at any point in time should be small. However, we require further constraint on the process that updates bias-value changes in order to guarantee efficiency.

For this approach, we assume that the bias-value update process must be *locally efficient*. We define a locally efficient bias-update process as one that satisfies two properties: (1) the update can affect the bias value of *at most* a constant number of elements and (2) updating bias takes time strictly sub-linear in the number of elements (i.e. $o(|E|)$). Since index updates are efficient independently, and locally efficient bias-value processes are bounded, index updates are efficient for this algorithm.

Two examples of locally efficient bias-value update processes are implementations of (1) recency and (2) frequency retrieval biases. To bias retrievals towards recency, the activation value of an activated element is updated to one greater than the globally largest activation value, which requires a global counter in order to avoid a linear scan of elements. To bias retrievals towards frequency, the activation value of an activated element is updated to one greater than its prior value. It is clear that for both of these processes, the update is efficient (comprising a lookup and summation) and local (affecting only the bias value of a single element). In Section 5.5.4 we describe and evaluate a locally efficient approximation to base-level activation (*Anderson and Schooler*, 1991), which summarizes the activation history of an element, incorporating both recency and frequency information.

5.4.3.2 Query Evaluation

Given the supplementary index described above, our retrieval algorithm exploits two optimizations: (1) statistical query optimization (*Chaudhuri*, 1998) and (2) bias pre-sorting. First, we develop a query plan that orders search based upon the statistics of augmentation frequency that are stored in the supplementary index, with the intent

of succeeding/failing as early as possible. Second, we order our search of candidates such that the first successful result we find is guaranteed to have the highest bias value, and thus search can terminate (i.e. we need only examine full candidate lists in failure conditions).

We first generate a sorted list, Q, of all augmentations, $c \in C$, keyed ascending on R_c (the elements that contain an augmentation c), which requires $\mid C \mid$ queries on the inverted index). Q represents a specialized query plan, sorted in ascending order of element-list size. Given the example cue $\{ \alpha, \epsilon \}$, and the example semantic memory above, the example query plan is either $[\alpha, \epsilon]$ or $[\epsilon, \alpha]$ (since R_α=2 and R_ϵ=2), and we use the former for the remainder of this analysis.

Next, we pop the first augmentation from Q (α) and retrieve a pointer, w, to the head of the sorted element list in the supplementary index (initially referring to element A). Note that since this list is updated incrementally with changes to semantic memory, we do *not* have to compute this list in response to the query. However, we may find that a set of elements have bias value of ∞; if so, we perform a lookup for its true bias value and execute insertion sort into a second list, L', iterate w, and repeat until w refers to an element with a non-infinite bias value. Now both w and [possibly empty] L' refer to sorted lists: by incrementally merging them, we iterate over the most constrained list of candidate elements. In our example, w initially refers to A, which does have a bias value of ∞. Thus, we arrive at w pointing to B=1.73 and L' containing list [A=1.41]. By merging the heads of these lists, we therefore examine element B first (because 1.73 >1.41).

Iterating over the remaining augmentations in Q ([ϵ]), we verify, using the original semantic memory (not the supplementary index), whether element B satisfies all remaining constraints. If so, return the element and *success*. If the candidate fails any of the remaining constraints, we merge the next candidate from w and L' and retry verification. If no candidate successfully verifies (i.e. we exhaust w and L'),

return *failure*. In our example, B verifies, and thus we conclude without considering element A.

If our assumption regarding small element cardinality holds, and we have chosen an appropriate value of τ, then L' will be small, and thus on-demand sorting via bias values will be minimized. In the worst case, this retrieval algorithm grows linearly with $\mid E \mid$. However, $\min(R_{c \in C})$ may be much smaller, and thus candidate search will be bound by *cue-feature selectivity*: the degree to which the most constraining cue feature limits the scope of search. The degree to which the algorithm must examine this resulting candidate list is controlled by *cue-feature co-occurrence*: the likelihood of cue features to augment the same element.

5.5 Evaluation

We instantiated the efficient implementation as the semantic memory of Soar v9.3.2, using version 3 of the SQLite in-process relational database engine (*Hipp*, 2012) to implement our relational schemas. Our goal in this section is to understand the degree to which Soar's semantic memory supports useful operation across a variety of domains while scaling to large stores of knowledge over long agent lifetimes.[3]

5.5.1 Lexical Queries

To compare to the work done by *Douglass et al.* (2009), our first evaluation issued lexical queries in the WordNet (*Miller*, 1995) lexicon. As with *Douglass et al.*, we used the WN-LEXICAL WordNet 3 data conversion (*Emond*, 2006). The data set has over 820K chunks, which includes over 212K word/sense combinations. Once imported, Soar's semantic store, including all indexing structures, is about 400MB. All experimental results below were run on a 2.8GHz Core 2 Extreme processor with 4GB of RAM using the *recency* bias.

[3]This evaluation has been published as (*Derbinsky et al.*, 2010; *Derbinsky and Laird*, 2011, 2012a).

Our first experiment was to verify (a) that retrieval time was independent of augmentation selectivity and (b) that the retrieval bias was processed efficiently in under-specified cues. We performed semantic retrievals on 100 randomly chosen, single-augmentation cues, averaged over 10 trials. Retrieval time was 0.1887 milliseconds each (0.0216 standard deviation).

Our next experiment focused on larger cues. We randomly chose 10 nouns and formed a cue from their full sense description ($|\ cue\ | = 7$). Retrieval time averaged 0.2973 milliseconds over 10 trials each (0.0108 standard deviation).

Douglass et al. used a derived subset of the WN-LEXICAL dataset, so direct replication of their work was not possible. They reported retrievals of about 40 milliseconds with cues of 1-4 augmentations on a declarative memory with about 232.5k chunks. Our results show 100x faster retrievals on a comparable set of cues scaling to a 3x larger memory store. Additionally, these results provide evidence that real-time agents can utilize this semantic-memory mechanism with WordNet, as the retrieval times were more than two orders of magnitude smaller than the 50-milliseconds reactivity requirement.

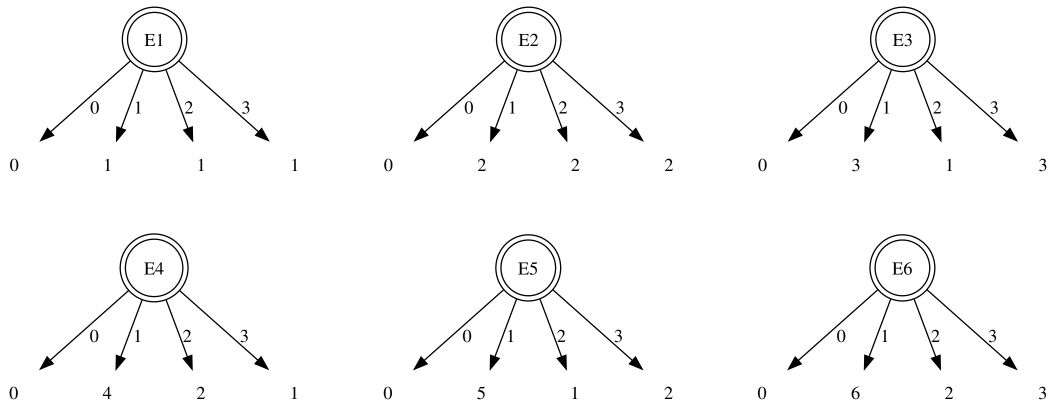

Figure 5.1: Example output of synthetic generator ($k = 3$).

5.5.2 Synthetic Data

In order to evaluate how our semantic-memory retrieval mechanism scales to large stores of knowledge, we developed a scalable, synthetic dataset generator. The generator takes as input a single parameter, k, and Figure 5.1 shows the output for $k = 3$, where nodes map to semantic elements, and edges are augmentations. As exemplified, the output is a semantic store with $k!$ elements and $(k+1)!$ total augmentations. In addition to scaling the number of semantic elements, the generator produces element augmentations such as to allow precise control over cue-feature selectivity. As illustrated in Figure 5.1, each element has a unique augmentation (label "1"), each has an augmentation that is shared with all elements (label "0"), and for the remaining augmentation labels, $label \in (2 - k)$, precisely $(1/label)$ elements share that label-value pair. For example, in Figure 5.1, the ("label"=selectivity) set is as follows: ("0"=1.0, "1"=1/6, "2"=1/2, "3"=1/3).

For this evaluation, we utilized this generator to create data sets that had sizes comparable to commonly used KBs: $(k = 7, 8) \sim$ SUMO (4.5K classes, 250K facts); $(k = 8, 9) \sim$ WordNet (212K word senses, 820K facts); and $(k = 9, 10) \sim$ Cyc (500K concepts, 5M facts). Table 5.1 lists statistics of the data sets we generated, with the associated k value. While the number of data sets is small, the last column illustrates that memory consumption is linear ($R^2 > 0.99$), costing about 50-80 bytes per element/augmentation. By comparison, a textual description costs about 20 bytes per element/augmentation, suggesting about a 3-4x overhead for indexing. All

Table 5.1: Statistics of synthetic data sets used for semantic-memory evaluation.

Parameter k	Elements $k!$	Augmentations $(k+1)!$	Store Size (MB)	Cost $\frac{bytes}{k!+(k+1)!}$
7	5,040	40,320	3.00	69.35
8	40,320	362,880	27.81	72.32
9	362,880	3,628,800	291.95	76.69
10	3,628,800	39,916,800	2,048.00	49.31

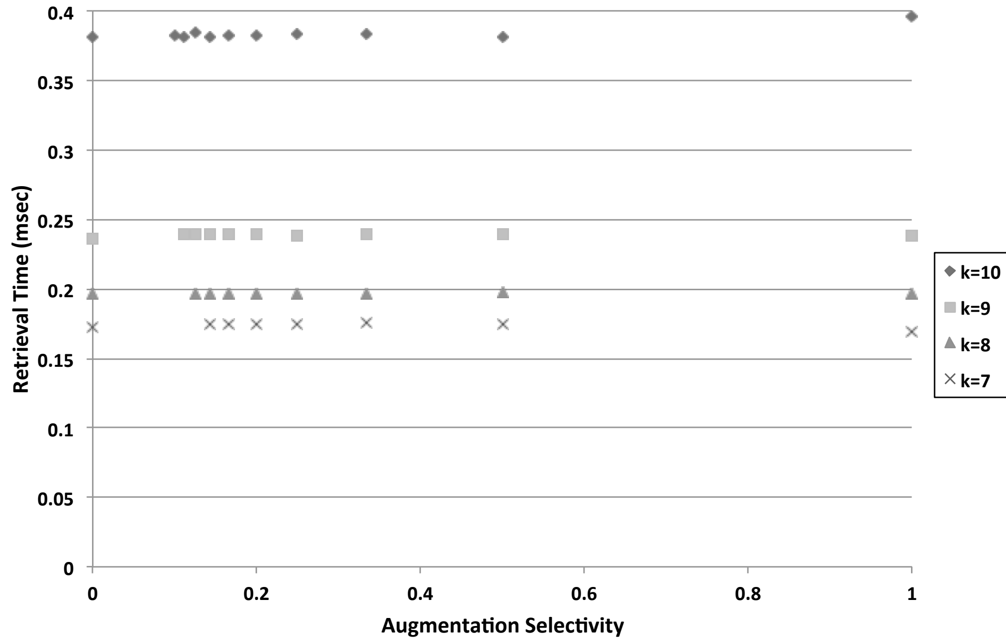

Figure 5.2: Synthetic: augmentation-selectivity sweep.

experimental results below were run on a 2.8GHz Core 2 Extreme processor with 4GB of RAM using the *recency* bias.

Our first experimented evaluated whether the retrieval mechanism provides bounded retrievals for under-specified cues, independent of the number of candidate elements. For each distinct augmentation in each of the data sets, we constructed a cue ($| C |= 1$) and measured retrieval time. Figure 5.2 plots retrieval time versus augmentation selectivity for each data set: we see nearly constant-time retrievals within each data set, independent of augmentation selectivity, measuring just under 0.4 milliseconds for the largest ($k = 10$).

Our second experiment evaluated whether combinations of augmentations result in complex cues that adversely affect retrieval time. We constructed all possible lengths of cues using all combinations of augmentation selectivity and measured retrieval time. Figure 5.3 plots retrieval time versus cue size and linearly regresses this relationship for each data set. We found that the only factor affecting retrieval time within a data set was the number of augmentations in the cue ($R^2 > 0.99$), achieving a maximum

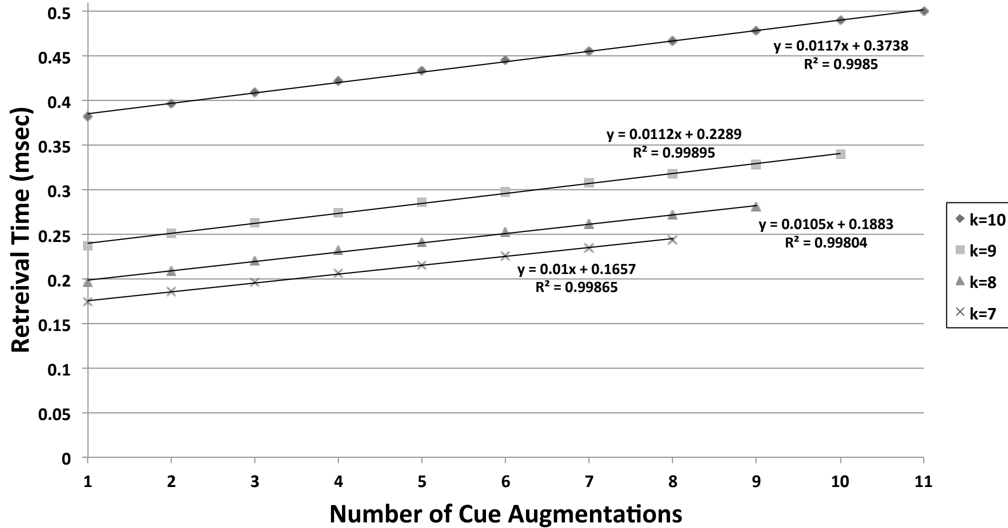

Figure 5.3: Synthetic: cue-size sweep with linear regressions.

of about 0.5 milliseconds for the largest data set ($k = 10$). Across data sets, we found that the intercept (i.e. overhead required to execute a retrieval) grows linearly with the number of elements and augmentations ($R^2 = 0.96$) and the slope (i.e. growth rate with respect to number of cue augmentations) grows logarithmically ($R^2 > 0.99$).

For both experiments, our mechanism performed two orders of magnitude faster than our reactivity requirement (50 milliseconds). These results illustrate that our mechanism can scale to very large data sets, and are not affected by cue-feature selectivity, assuming cue-feature co-occurrence is not low: in these experiments, there always existed a perfect match to the cue, but in the case of the second experiment, large cues opened up a space of candidates that was well-pruned by our query planner.

5.5.3 Mobile Robotics

In order to evaluate the degree to which our semantic-memory mechanism was efficient and effective for real-time agents, we developed a Soar agent to control a simulated mobile robot (*Laird et al.*, 2011b). Our evaluation used a simulation instead of a real robot because of the practical difficulties in running numerous, long

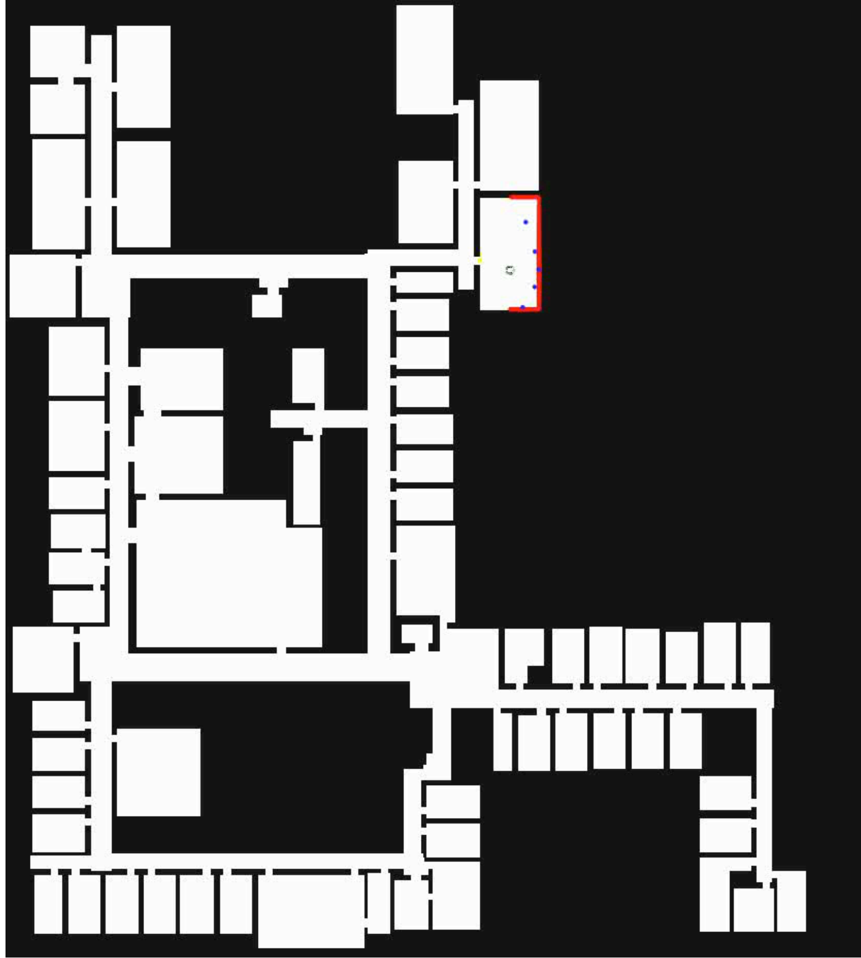

Figure 5.4: Mobile Robotics: domain map, with robotic agent located in the Soar lab (3844 BBB).

experiments in large physical spaces. However, the simulation was quite accurate and the Soar rules (and architecture) used in the simulation were exactly the same as the rules used to control the real robot.

The robot's task was to visit every room on the third floor of the Bob & Betty Beyster building at the University of Michigan (see the full map in Figure 5.4). For this task, the robot visits over 100 rooms and takes about 1 hour of real time. During exploration, it incrementally builds an internal topological map, which, when completed, requires over 10,000 working-memory elements to represent and store. In addition to storing information, the model reasons about and plans using the map

in order to find efficient paths for moving to distant rooms it has sensed but not visited. The agent uses episodic memory (Chapter IV) to recall objects and other task-relevant features during exploration.

Our focus in this domain was the use of semantic memory to improve agent reactivity, an issue that arises as the agent monotonically builds its domain map. As discussed in Chapter IV, Soar's episodic memory reconstructs prior episodes in their entirety, and thus the process scales linearly with the size of working memory at the time when the episode was encoded. In this experiment, we augmented the agent with task knowledge (represented in rules) to partition its representation of the map into short-term knowledge (stored in working memory), which is useful for immediate reasoning, and long-term knowledge (stored in semantic memory), which is necessary for planning over information not accessible to the robot's sensors. The outcome was that the agent could still complete its task, but both working-memory size and maximum decision time (comprising primarily episodic retrievals) were greatly reduced, and thus the agent benefited from improved reactivity.

It is interesting to note that the use of semantic memory to represent domain knowledge subtly changed some aspects of agent behavior. One example was when the agent was selecting from amongst known rooms to explore. When the rooms are all in working memory, a rule can match all rooms and apply arbitrary reasoning to choose from amongst them (e.g. distance from present position, expected utility, etc.). However, when using semantic memory, the agent can only constrain the room retrieval with respect to pre-encoded features, and the recency bias selects from amongst all candidates. These differences, however, did not lead to significant differences in time to complete the task.

All experimental results below were gathered on an Intel i7 2.8GHz CPU with 8GB of RAM using the *recency* bias. Because each experimental run took 1 hour, we did not duplicate our experiments sufficiently to establish statistical significance

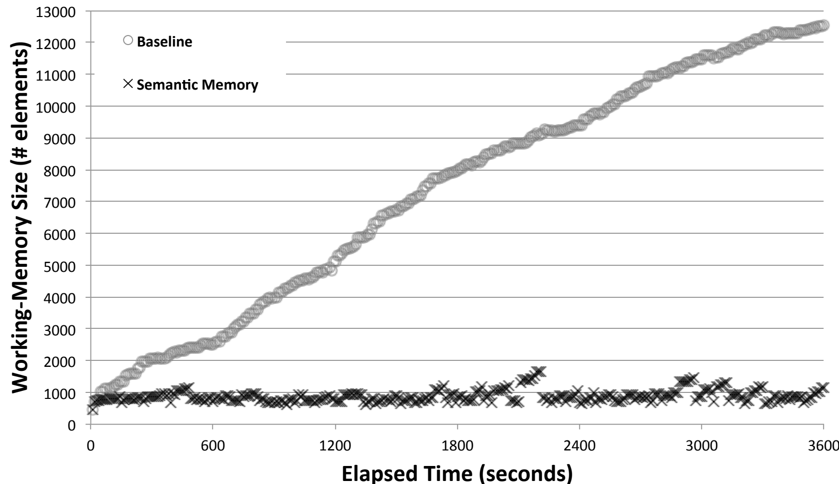

Figure 5.5: Mobile Robotics: working-memory size versus elapsed time.

and the results we present are from individual experimental runs. However, we found qualitative consistency across our runs, such that the variance between runs is small as compared to the trends we focus on below.

Figure 5.5 plots working-memory size over time, comparing a baseline agent, which stores its map in working memory, and the agent partitioning its knowledge using semantic memory. Over time we see a monotonic growth for the baseline agent, whereas the semantic-memory agent stays relatively constant, only growing temporarily when it retrieves structures for planning. We see that after one hour, the semantic-memory agent has 11,000 fewer working-memory elements, more than a 90% reduction as compared to the baseline.

Figure 5.6 plots maximum decision-cycle time in milliseconds over time, comparing these two agents. The dominant time reflected by this data is time to reconstruct prior episodes that are retrieved from episodic memory. As a result, after less than one hour, the decision-cycle time for the baseline agent grows above the 50-milliseconds threshold, while the semantic-memory agent appears constant ($\sim 10 - 20$ milliseconds). This agent was also used in a much longer evaluation (see Section 4.5.5.4), which maintained reactivity for 12 continuous hours of operation using both semantic

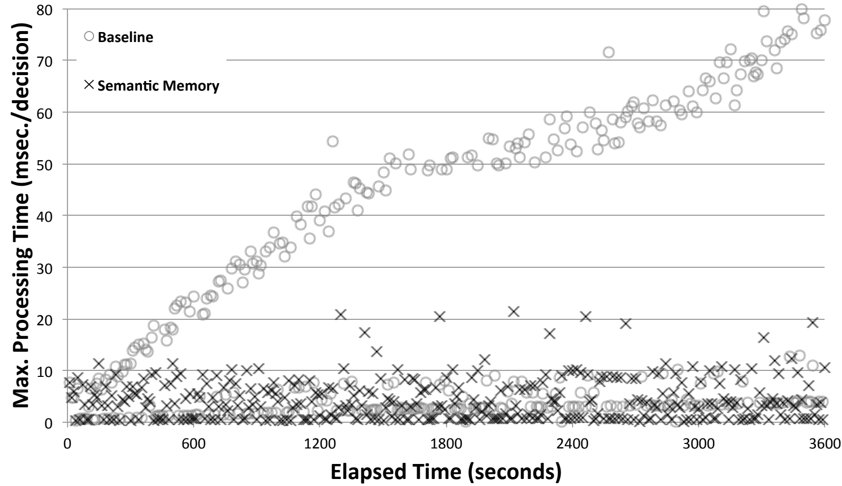

Figure 5.6: Mobile Robotics: max decision-cycle time versus elapsed time.

and episodic memory.

In summary, this experiment shows that an effective and efficient semantic memory can improve an agent's ability to reason in large domains over long periods of time while staying reactive to environmental dynamics. In this experiment, we provided the agent task knowledge of how to perform this partitioning. In Chapter VI, we will revisit this task and demonstrate an efficient and task-independent selective-retention mechanism that can effectively prune working memory, resulting in comparable size and improved decision-cycle time.

5.5.4 Word Sense Disambiguation

One of the goals of this dissertation was to develop a suite of memory-retrieval heuristics and evaluate their effectiveness and efficiency in a variety of tasks to determine which heuristics are best suited to be used for memory retrieval in a cognitive architecture. As a first step towards this goal, we focused on one specific challenge facing long-term memory: given a large store of knowledge, and an ambiguous cue that matches multiple stored memories, how does the system efficiently determine which memory to retrieve? The rational analysis of memory (*Anderson and Schooler*, 1991),

positing that human memory optimally solves this problem with respect to the history of past memory access, yielded the base-level activation model, which is widely used in the cognitive-modeling community. However, existing computational implementations of this model do not scale to tasks that require access to large bodies of knowledge (*Douglass et al.*, 2009). In this evaluation, we explored an initial set of memory retrieval heuristics that incorporate recency and frequency of memory access: we evaluated their effectiveness and scaling in the word sense disambiguation (WSD) task, an important and well-studied problem in the Natural Language Processing community (*Navigli*, 2009).

5.5.4.1 Problem Formulation

The English language contains *polysemous* words, those that have multiple, distinct meanings, or *senses*, which are interpreted differently based upon the context in which they occur. Consider the following sentences:

a. Deposit the check at the *bank*.

b. After canoeing, they rested at the *bank*.

The occurrences of the word bank in the two sentences clearly denote different meanings: 'financial institution' and 'side of a body of water,' respectively. Word sense disambiguation is the ability to identify the meaning of words in context in a computational manner (*Navigli*, 2009). The task of WSD is critical to the field of NLP and various formulations have been studied for decades.

Our interest is in general competence across a variety of domains, and so we adopted the *all-words* WSD formulation, where the system is expected to disambiguate all open-class words in a text (i.e. nouns, verbs, adjectives, and adverbs). As input, the agent receives a sequence of sentences from a text, each composed of

a sequence of words. However, as the focus of this work is memory, not unsupervised natural language processing, we supplemented the input with the following two sources of additional structure, each of which is not uncommon in the WSD literature. First, each input word is correctly tagged with its contextually appropriate part-of-speech. Second, the agent is assumed to have access to a static machine-readable dictionary (MRD), such that each lexical word/part-of-speech pair in the input corresponds to a list of word senses within the MRD. For each sense, the MRD contains a textual definition, or *gloss*, and an annotation frequency from a training corpus. Thus, for each input word, the agents task is to select an appropriate sense from the MRD, from which there may be multiple equally valid options for the given linguistic context.

5.5.4.2 Data Sets

To evaluate our work, we made use of three *semantic concordances*: each a textual corpus and lexicon linked such that every substantive word in the text is linked to its appropriate sense in the lexicon (*Miller et al.*, 1993).

The first semantic concordance is SemCor, the biggest and most widely used sense-tagged corpus, which includes 352 texts from the Brown corpus (*Kucera and Francis*, 1967). We used the 186 Brown corpus files that have all open-class words annotated, which includes more than 185,000 sense references to version 3 of WordNet (*Miller*, 1995). WordNet 3, the most utilized resource for WSD in English, includes more than 212,000 word senses. To prevent over-fitting in our results, we also utilized the Senseval-2 and Senseval-3 all-words corpora (*Edmonds and Kilgarriff*, 2002), linked with WordNet 3. These data sets are nearly two orders of magnitude smaller than SemCor, comprising only 2,260 and 1,937 sense references, respectively.[4]

[4]The SemCor, Senseval-2, and Senseval-3 data sets are available to download at `http://www.cse.unt.edu/~rada/downloads.html`. WordNet is available to download at `http://wordnet.princeton.edu`.

5.5.4.3 Task Analysis

In our formulation of the WSD task, an agent is provided a lexical word/part-of-speech pair and must select an appropriate sense, for the current context, from amongst a static list defined by the MRD. Let s represent the set of candidate senses from the MRD and let a represent the set of appropriate sense assignments in this context.

Given an arbitrary input word, an important measure that characterizes the difficulty of the task is $\mid s \mid$, the cardinality of the set of candidate senses, referred to as *polysemy level* in the literature. However, as some open-class words in all of our data sets are tagged with more than one appropriate sense (0.33% in SemCor; 4.25% Senseval-2; 1.91% Senseval-3), it is also important to consider $\mid a \mid$, the cardinality of the set of appropriate sense assignments. In this section, we characterize these measures across each of the data sets, and resolve the joint distribution of these values to derive expected task performance, given a random sense selection strategy.

To begin, we define a derived joint measure, $certainty = \frac{|a|}{|s|}$. For a given lexical word/part-of-speech pair, since the set of suitable sense assignments is non-empty ($\mid a \mid > 0$) and comprises a subset of the full set of candidate senses ($a \subseteq s$ and $\mid a \mid \leq \mid s \mid$), the range of *certainty* is $(0, 1]$, where a value of 1 is, intuitively, unambiguous (any selection from amongst the candidate set is appropriate) and as a value becomes closer to 0, it becomes increasingly ambiguous (an appropriate selection is increasingly rare).

Given this nomenclature, Figure 5.7 represents the distribution of certainty within the SemCor data set, plotting the cumulative proportion of corpus words against certainty. Both this plot and the descriptive analysis below aggregate the distribution with respect to part-of-speech; while our work does not investigate methods that are differentially sensitive to part-of-speech, we see distinctions in this distribution, which may be useful to future work. We only plot and textually analyze SemCor, the

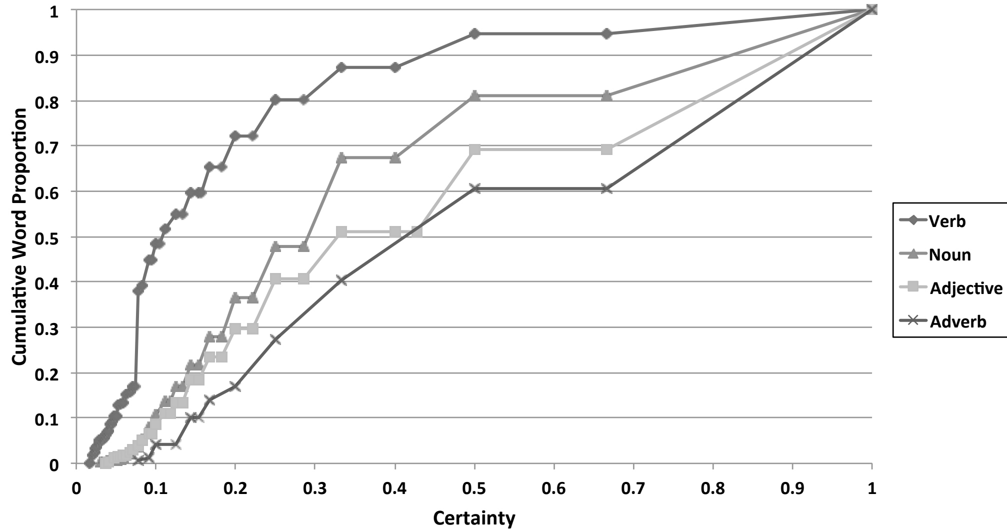

Figure 5.7: WSD: SemCor cumulative word proportion vs. certainty.

largest data set, but Table 5.2 summarizes pertinent outcomes for all three semantic concordances.

We first draw out the proportion of words with certainty value 1, those that require no disambiguation, by reading the second plotted point from the right for each part-of-speech. While for adverbs and adjectives this statistic is about 39% and 31%, respectively, for nouns and verbs it is about 19% and 5%. Across the entirety of SemCor, this statistic is 19.54%, establishing the absolute minimum for task performance in this formulation of WSD using SemCor. We next assess the median certainty for each part-of-speech by reading the x-axis as each part-of-speech intersects 50% on the y-axis. For adverbs, the median is $\frac{1}{2}$; for adjectives and nouns it is $\frac{1}{3}$; for verbs it is $\frac{1}{9}$; and the overall median certainty in SemCor is $\frac{1}{4}$. Finally, the average certainty, and thus the expected performance given a random-selection strategy, is 38.73%.

5.5.4.4 Baseline Task-Performance Results

To contextualize the performance results of a memory-based approach to the WSD task, we first implemented a set of baseline algorithms from the WSD literature. The results from these baselines are summarized in Table 5.3, including *random* selection, derived as expected performance in the previous section. Note that all algorithms we implement select a sense for all input words, and thus precision and recall are identical for all results, so we simply report them jointly as "task performance."

WordNet includes, for each word sense, an annotation frequency from the Brown corpus and the first baseline selection policy, *frequency* bias, exploits this information by choosing the most frequent sense for each lexical word/part-of-speech input pair. As the SemCor textual corpus is a subset of the Brown corpus, we expected this resource to be highly informative and, unsurprisingly, this algorithm yields nearly twice the performance of pure random selection. As the Senseval data sets were not derived from the Brown corpus, it is unsurprising that the absolute performance advantage of this heuristic is not as great when applied to these corpora. However, the relative improvement for Senseval-3 is greater than that of SemCor (98.33% versus 97.24%), which likely reflects the increased difficulty of Senseval-3 (see Table 5.2). Incorporating a frequency bias is not uncommon in the WSD literature, sometimes termed *commonest*, but it does tend to suffer, as found here, when the frequency distribution of the MRD is not representative of the corpus.

The remaining baselines were two variants of the Lesk algorithm for word sense disambiguation (*Lesk*, 1986). The Lesk algorithm is a commonly used baseline metric

Table 5.2: WSD: Semantic concordance task-analysis summary.

	SemCor	Senseval-2	Senseval-3
Unambiguous	19.54%	19.69%	15.02%
Median Certainty	0.25	0.25	0.2
Minimum Certainty	0.0169	0.0204	0.0169
Expected Performance	38.73%	40.56%	32.98%

(*Vasilescu et al.*, 2004) that assumes that words in a given "neighborhood" (such as a sentence) tend to share a common topic, and thus biases sense selection based upon shared terms in sense definitions and context. We explored the classic algorithm, with constant-sized neighborhood windows, as well as a "simplified" Lesk algorithm (*Kilgarriff and Rosenzweig*, 2000), which defines word context as simply the terms in the neighborhood, as opposed to their definitions. The performance of the Lesk family of algorithms is known to be highly sensitive to the exact wording of sense definitions, and so it is common to supplement Lesk with heuristics and additional sources of semantic meaning, such as in *Banerjee and Pedersen* (2002). Thus, for both classic and simplified Lesk, we evaluated four supplemental heuristics: (1) the use of a stop list, which excludes definition terms that are common to the target language, such as "a" and "the"; (2) excluding example sentences from sense definitions, to avoid uninformative overlapping terms; (3) the use of the Porter Stemming (*Porter*, 2006) algorithm to strip word suffixes, to facilitate overlap of words with common linguistic roots; and (4) a bias towards the corpus frequency information, applied in the case of equivalent sense evaluation. We evaluated the combinatorial set of these parameters across both algorithms. The maximum results for both classic and simplified algorithms occurred using the stop list, pruned definitions, and frequency bias, but not the Stemming algorithm. For the classic algorithm, we achieved maximum task performance with a neighborhood size of 2. However, as summarized in Table 5.3, the Lesk variants consistently underperformed, compared to frequency bias.

These baseline results are specific to our implementation and data sets, and are

Table 5.3: WSD: Baseline task-performance results.

	SemCor	**Senseval-2**	**Senseval-3**
Random	38.73%	40.56%	32.98%
Frequency Bias	76.39%	65.56%	65.41%
Lesk	63.40%	58.17%	53.46%
Simplified Lesk	65.52%	56.28%	53.66%

not intended for representation of or comparison to modern NLP techniques, but instead provide a reasonable baseline for our memory-based results.

5.5.4.5 A Memory-based Approach

The main thrust of this evaluation is to understand and develop memory mechanisms that effectively support agents in a variety of tasks, while computationally scaling as the amount of stored knowledge grows to be very large. Thus far in this section we have characterized one important task, word sense disambiguation, including an analysis of WSD across three data sets and the types of performance we can expect from baselines that do not adapt their sense selection policy to the task instance. In this section we describe and evaluate a simple approach to the WSD task that is available to those agents with a semantic memory.

Our approach is as follows: given a lexical word/part-of-speech input, the agent cues its memory for a sense that satisfies these constraints and selects the first retrieved result. We make two assumptions in our evaluation of this approach. First, the agent's long-term memory is preloaded with at least the contents of the data set's MRD, which affords the agent the potential to perform perfectly on the task, as it is not constrained by a limited vocabulary. In the case of our data sets, this assumption requires that the agent's memory mechanism scale computationally to at least the knowledge contained in WordNet (which we showed as reasonable in Section 5.5.1).

Our second assumption is that immediately after the agent attempts to disambiguate a word, it is supplied with the set of appropriate senses for that input. This evaluation paradigm is important to isolate the effect of memory bias: it eliminates unintended divergent learning and result obfuscation, which could occur sans truthful feedback.

In this evaluation, the agent's *a priori* knowledge and approach to the WSD task remains constant. However, we experimentally alter the agent's semantic-memory

retrieval mechanism, changing how correct sense assignments in the past bias future retrievals. We investigate the degree to which the recency and frequency of past assignments inform future retrievals. Our set of experimental heuristics is motivated by the rational analysis of memory (*Anderson and Schooler*, 1991), which demonstrated, across a set of linguistic tasks, reliable relationships between recency and frequency of past events and future memory accesses. When integrated within a memory system, these retrieval heuristics are independent of the WSD task, and can thus be applied and evaluated on additional WSD data sets, as well as tasks beyond WSD or linguistic settings.

We evaluate recency and frequency biases individually, and then proceed to explore the base-level activation model, which combines these heuristics. For each heuristic, we apply a greedy selection strategy, retrieving the word sense with the greatest bias value, and selecting randomly from amongst ties.

Unlike the non-adaptive WSD baselines, memory-endowed agents have the potential to improve with added corpus exposure, and thus we performed 10 sequential runs for each experimental condition, where the agent attempts to disambiguate the entirety of the corpus during each run. We report performance on the first, second, and tenth run of each of the data sets: the first run affords direct comparison to baseline results, the second run illustrates speed of learning, given relatively little corpus exposure, and the tenth speaks to asymptotic performance. Tables 5.4, 5.5, and 5.6 report expected performance on SemCor, Senseval-2, and Senseval-3, respectively, as opposed to the sample average of individual probabilistic runs, and thus even small differences should be considered relevant.

5.5.4.6 Individual Memory Bias Task Results

The first heuristic we evaluate is *recency*, which biases ambiguous retrievals towards the last selected sense for each input. This bias, related to the *one sense per*

discourse heuristic (*Gale et al.*, 1992), performs well if the same sense is used repeatedly in immediate succession, but does not improve performance after the first full exposure to the data set, as demonstrated by no difference in performance between runs 2 and 10, independent of the semantic concordance.

The next heuristic is *frequency*, which biases ambiguous retrievals towards the sense that has been retrieved most often. This bias performs well if particular senses of words are generally more common than others in a corpus, as opposed to being highly dependent upon sentence context.

Finally, to establish an upper bound on the degree to which recency and frequency can individually contribute to WSD performance, we implemented an *oracle* bias. For each input, this heuristic scores both the recency and frequency algorithms described above and returns the result from the two algorithms that provided the best score.

We draw two conclusions from the data in Tables 5.4, 5.5, and 5.6. First, under the assumptions of our evaluation, the run 10 results suggest that memory retrieval agents perform better than the baselines from Table 5.3, with the potential for additional reasoning mechanisms to improve performance further. And second, nearly all memory bias results for run 10 are better than all baselines in the respective test set (excl. SemCor/Recency). This suggests that history of sense assignment, with relatively little corpus exposure, yields a performance benefit in the WSD task, an advantage that is not dependent upon MRD definition quality (unlike Lesk).

5.5.4.7 Base-Level Activation Task Results

The top two rows in Tables 5.4, 5.5, and 5.6 present evidence that recency and frequency of word-sense assignment can individually yield performance benefits in the WSD task. Additionally, the relative gain in the *oracle* results (up to nearly 8% in SemCor) indicates that there is room for improvement. However, applying these findings to a memory system requires a model of how these heuristics combine in a

task-independent fashion to bias memory retrieval (recall that the oracle algorithm is not possible to implement, as it requires the memory system to evaluate correct sense assignments during word sense selection). In this section, we explore base-level activation (*Anderson and Schooler*, 1991), a memory bias model that jointly incorporates recency and frequency of memory access.

Base-level activation computes the activation bias of a memory using an exponentially decaying memory-access history:

$$ln(\sum_{j=1}^{n} t_j^{-d})$$

where n is the number of accesses of the memory, t_j is the time since the jth access, and d is a free decay parameter.

We performed exploratory sweeps within each data set for the decay parameters and evaluated the base-level activation model in the same fashion as the individual memory biases above (see the bottom row of Tables 5.4, 5.5, and 5.6): it performs competitively and bests recency and frequency run 10 results in SemCor. The SemCor results used a decay parameter of 0.7, whereas both Senseval corpora used 0.4.

Table 5.4: WSD: Memory bias task performance results for SemCor.

	Run 1	Run 2	Run 10
Recency	72.34%	74.43%	74.43%
Frequency	71.69%	76.21%	76.53%
Oracle	79.51%	83.77%	84.08%
Base-level	74.45%	77.90%	78.47%

SemCor

Table 5.5: WSD: Memory bias task performance results for Senseval-2.

	Run 1	Run 2	Run 10
Recency	61.74%	84.02%	84.02%
Frequency	62.13%	88.89%	89.28%
Oracle	63.68%	89.93%	90.23%
Base-level	62.17%	87.01%	88.47%

Senseval-2

Table 5.6: WSD: Memory bias task performance results for Senseval-3.

	Run 1	Run 2	Run 10
Recency	54.32%	79.29%	79.29%
Frequency	54.85%	84.30%	84.86%
Oracle	57.25%	86.23%	86.77%
Base-level	54.41%	82.19%	83.84%

Senseval-3

Table 5.7: WSD: Individual bias evaluation: maximum query time.

	SemCor	Senseval-2	Senseval-3
Recency	0.85 msec.	0.82 msec.	0.80 msec.
Frequency	0.87 msec.	0.82 msec.	0.78 msec.

5.5.4.8 Scalability Results

Our goal in this evaluation is to explore memory heuristics that are both effective and efficient in the WSD task. The task-performance results suggest that incorporating recency and frequency of past memory access to bias future retrievals, both individually and jointly, supports WSD task performance, as compared to non-memory baselines. We now evaluate the degree to which these heuristics support efficient WSD memory retrievals: we report the maximum time in milliseconds, averaged over ten trials, for a Soar agent to retrieve a memory on a 2.8GHz Intel Core i7 processor.

As discussed in Section 5.4.3.1, the recency and frequency heuristics are *locally efficient*: both rely upon memory statistics that can be maintained incrementally and only update a single memory at a time. Hence, it is not surprising that both biases perform faster than the 50-milliseconds threshold across all data sets (see Table 5.7).

However, base-level activation does not appear to be locally efficient: it includes a time-decay component that changes frequently for all memories. We implemented a highly optimized version and were able to achieve 13.25-milliseconds retrievals in SemCor; however, this time is not bounded, growing with the store size. This heuristic, however, also affords a useful monotonicity: from the time that bias is calculated for a memory, that value is guaranteed to *over-estimate* the true bias until the memory is accessed again in the future. This characteristic affords a locally efficient approximation: the memory system updates activation only when a memory is accessed.

To evaluate this approximation, we consider query time, WSD task performance (comparable to results in Tables 5.4, 5.5, and 5.6), and model fidelity (a measure of interest to cognitive modelers), which we define as the smallest proportion of senses

Table 5.8: WSD: Base-level activation approximation results.

	SemCor	Senseval-2	Senseval-3
Max. Query Time	1.34 msec.	1.00 msec.	0.67 msec.
Run 10 WSD Perf.	77.65%	89.03%	84.56%
Min. Model Fidelity	90.30%	95.70%	95.09%

that the model selected within a run that matched the results of the original model.

We implemented this approximation within the semantic memory module of Soar and the results are summarized in Table 5.8 ($d = 0.5$). To guarantee constant time bias calculation (*Petrov*, 2006), we used a history of size 10. We also applied an incremental maintenance routine that updated memories that had not been accessed for 100 time steps, so as to avoid stale bias values. The query times across all data sets are far below our requirement of 50 milliseconds and an order of magnitude faster than the original. The run 10 WSD results are comparable to the original, and model fidelity is at or above 90% for all runs of all data sets. These results show that our base-level activation approximation can support effective and efficient retrievals across large stores of knowledge.

5.6 Discussion

In this chapter, we presented and evaluated techniques to enhance intelligent agents with semantic memory. Our abstract representation (symbolic triples) is sufficiently general (R3) for use in a variety of tasks, as evinced in the prolific use of the declarative module of ACT-R. It also incorporates a task-independent (R6) processes for the agent to deliberately and incrementally (R1) encode semantic knowledge, as well as flexible (R5) retrievals that, in practice, scale to large stores of knowledge (R4). We implemented this mechanism in the Soar cognitive architecture and evaluated it in a variety of problem domains, including lexical queries, mobile robotics, and word sense disambiguation. We showed that while the algorithms are not immune

to properties of cues that negatively affect performance (e.g. low feature selectivity and co-occurrence), the mechanism does support many cues and retrieval biases that agents can apply to support task performance (R2).

5.6.1 Future Work

In order for a semantic-memory mechanism to fulfill the requirements imposed by generally intelligent agents (Chapter II), computational-resource utilization must be bounded. To bound computation time for cue matching, there is existing work (*Terrovitis et al.*, 2006) that incorporates additional indexing methods to account for problematic cases of cue-feature co-occurrence and selectivity. However, we expect that the greatest threat to bounded retrieval times is not feature matching, but instead more complex retrieval biases, such as incorporation of context (e.g. spreading activation; *Anderson and Schooler*, 1991). There is evidence that for these biases, some benefits may come from parallel algorithms (e.g. *Douglass and Myers*, 2010), but ultimately it is likely that our memory model will require a relaxed specification of retrievals (i.e. heuristic search). However, as with all memory research, it will be important to balance scaling (R4) with agent task performance (R5), and thus much more work must be done to develop and evaluate agents that use semantic memory in a variety of problem domains. This will entail research into general methods for integrating semantic retrievals with agent reasoning in known tasks, as well as how strategies for agents to learn to utilize semantic knowledge in novel situations.

A major weakness in our current approach is that we do not have good theory for how agents accumulate semantic knowledge. In most of our evaluations, agents deliberately encode task-relevant information. The implications of deliberate storage are (1) that the agent must have task knowledge of what structures are useful to encode over time; (2) this encoding is intertwined with, and may interrupt, agent reasoning; and (3) there may be limitations as to the kinds of knowledge and reasoning

that can be brought to bear on estimating the value of future knowledge (e.g. it can be difficult to perform complex probabilistic reasoning in rules). In the mobile-robotics evaluation (Section 5.5.3), we implemented a "mirroring" policy, whereby changes to semantic objects in Soar's working memory were automatically paralleled in semantic memory. However, while mirroring ameliorates problem #2 (i.e. the agent does not need to deliberately issue as many storage commands), it still requires that the agent reason about initial storage (#1) and does not address issue #3. The declarative module of ACT-R implements an automatic-encoding policy that captures all changes to a set of working-memory buffers as new chunk instances; it is then activation that helps to identify useful chunks for later retrieval (i.e. memory bias serves as a form of selective utilization). Further research should explore the viability of this approach for agents that persist for long periods of time, especially with respect to the implications for bounding computational memory usage. Another interesting avenue of research is the automatic encoding, or *consolidation*, of a subset of episodic-memory structures to semantic memory. This approach has surface-level similarities to the Standard model of memory consolidation in humans (*Squire and Alvarez*, 1995) and relates to our intuitions as to the relative purpose of the dissociated memory mechanisms: episodic automatically captures the details of experience and, over time, the more common and context-independent features are transitioned to an abstracted semantic memory.

More generally, consolidation is just one direction for investigation of how a semantic-memory mechanism integrates with other components and processes in a general cognitive architecture. We have done some initial work to investigate how the supplemental index of our semantic-memory mechanism can serve as a frugal source of heuristic knowledge for recognition judgements (*Li et al.*, 2012). Another fruitful avenue will be to investigate the degree to which meta-data from other cognitive mechanisms and processes (e.g. emotional appraisals; *Marinier et al.*, 2009) can improve the robustness of semantic retrievals across a variety of problem domains.

CHAPTER VI

Competence-Preserving Retention of Learned Knowledge

This chapter documents our progress in understanding the computational challenges involved in selectively retaining, or *forgetting*, learned knowledge while maintaining agent proficiency across a variety of tasks.[1] We begin with a motivational description of forgetting in context of learning systems (Section 6.1); then discuss related work (Section 6.2); continue to our functional specification of a selective-retention policy as applied to memory mechanisms (Section 6.3); describe data structures and algorithms that efficiently implement the mechanism (Section 6.4); evaluate the mechanism, as implemented within the Soar cognitive architecture (Section 6.5); and conclude with a summary and discussion of future work (Section 6.6).

6.1 Motivation

Typical AI systems persist for short periods of time and/or have modest learning requirements. Generally intelligent agents, however, contend with complex, protracted tasks and must amass and effectively draw upon large stores of experience. For these systems, accumulation of large amounts of information over long agent lifetimes can lead to the computational-performance degradation.

[1]This work was previously published in (*Derbinsky and Laird*, 2012a,b).

This issue, where more knowledge can harm problem-solving performance, has been dubbed the *utility* problem, and has been studied in several contexts. *Markovitch and Scott* (1988) have characterized different strategies for dealing with the utility problem in terms of information filters applied at different stages in the problem-solving process. One common strategy that is relevant to generally intelligent agents is selective retention, or forgetting, of learned knowledge. The benefit of this approach, as opposed to selective utilization, is that all available knowledge is brought to bear on problem solving, a property that may be crucial for agent competence in complex tasks. However, it can be challenging to devise forgetting policies that work well across a variety of problem domains, effectively balancing the task performance of agents with reductions in retrieval time and storage requirements of learned knowledge. In context of the memory requirements for generally intelligent agents, forgetting knowledge is a challenging problem that arises at the intersection of scaling computational performance (R4) while still providing effective access (R5) in a task-independent fashion (R6).

In response, we present and evaluate a task-independent framework for selective retention of learned knowledge, which investigates a core hypothesis: it is rational for an agent's cognitive architecture to forget a unit of knowledge when there is a high degree of certainty that it is not of use, as calculated by base-level activation (*Anderson and Schooler*, 1991), and that it can be reconstructed in the future if it becomes relevant. In our evaluation, we demonstrate two instances of this framework which, when implemented in Soar, improve agent reactivity and scaling, while maintaining problem-solving competence.

6.2 Related Work

Previous cognitive-modeling research has investigated forgetting in order to account for human behavior and experimental data. As a prominent example, memory

decay has long been a core commitment of the ACT-R theory (*Anderson et al.*, 2004), as it has been shown to account for a class of memory retrieval errors (*Anderson et al.*, 1996). Similarly, research in Soar investigated task-performance effects of forgetting short-term (*Chong*, 2003) and procedural (*Chong*, 2004) knowledge. By contrast, the motivation for and outcome of this work is to investigate the degree to which selective retention can support long-term, real-time agents in complex tasks.

Prior work has demonstrated the potential for cognitive benefits of memory decay, such as in task-switching (*Altmann and Gray*, 2002) and heuristic inference (*Schooler and Hertwig*, 2005). In this work, we focus on improved reactivity and scaling.

We extend prior investigations of long-term symbolic learning in Soar (*Kennedy and Trafton*, 2007), where the source of learning was primarily from internal problem solving. We present evaluation domains that accumulate large amounts of information from interaction with external environments.

6.3 Functional Specification

As described, there are two components to our selective-retention framework with respect to an item of knowledge: (1) estimating whether the knowledge is of use and (2) whether it can be reconstructed in the future if necessary. This section details the first of these components, whereas the latter is left for the evaluation section, in which we describe task-independent methods that are specific to memory implementations.

We use the base-level decay model in order to identify those items of knowledge that have not been used recently and/or frequently, which is an indication of their importance to agent reasoning. To provide a basis for our efficient implementation, we now provide a formulation of this forgetting problem.

6.3.1 The Forgetting Problem

Let memory M be a set of elements, $\{m1, m2, \ldots\}$. Let each element m_i be defined as a set of pairs (a_{ij}, k_{ij}), where k_{ij} refers to the number of times element m_i was activated at time a_{ij}. We assume $|m_i| \leq c$: the number of activation events for any element is bounded. These assumptions are consistent with the ACT-R declarative memory when bounding chunk-history size (*Petrov*, 2006). This is also consistent with the semantic memory in Soar, as described in Chapter V.

We assume that activation of an element m at time t is computed according to the base-level activation model (*Anderson and Schooler*, 1991), where d is a fixed decay parameter:

$$B(m,t,d) = ln(\sum_{j=1}^{|m|} k_j \cdot [t - a_j]^{-d})$$

We define an element as *decayed*, with respect to a threshold parameter Θ if $B(m,t,d) < \Theta$. Given a static element m, we define L as the fewest number of time steps required for the element to decay, relative to time step t:

$$L(m,t,d,\Theta) := inf\{t_d \in \mathbb{N} : B(m, t + t_d, d) < \Theta\}$$

For example, element $x = \{(3,1),(5,2)\}$ was activated once at time step three and twice at time step five. Assuming decay rate 0.5 and threshold -2, x has activation about 0.649 at time step 7 and is not decayed: $L(x, 7, 0.5, -2) = 489$.

During a time step t, the following actions can occur with respect to memory M:

S1. A new element is added to M.

S2. An existing element is removed from M.

S3. An existing element is activated y times.

If S3 occurs with respect to element m_i, a new pair (t, y) is added to m_i. To maintain a bounded history size, if $|m_i| > c$, the pair with smallest a (i.e. the oldest) is removed from m_i.

Thus, given a memory M, we define that the *forgetting* problem, at each time step, t, is to identify the subset of elements, $D \subseteq M$, that have decayed since the last time step.

6.4 Efficient Implementation

Given this problem definition, a naïve approach is to determine the decay status of each element every time step. This test requires computation $O(|M|)$, scaling linearly with average memory size. The computation expended upon each element, m_i, will be linear in the number of time steps where $m_i \in M$, estimated as $O(L)$ for a static element.

Our approach draws inspiration from the work of *Nuxoll et al.* (2004): rather than checking memory elements for decay status, "predict" the future time step when the element will decay. First, at each time step, examine elements that either (S1) weren't previously in the memory or (S3) were activated. The number of items requiring inspection is bounded by the total number of elements ($|M|$), but may be a small subset. For each of these elements, predict the time of future decay (discussed shortly) and add the element to a map, where the map key is the predicted time step and the value is a set of elements predicted to decay at that time. If the element was already within the map (S3), remove it from its old location before adding to its new location. All insertions/removals require time at most logarithmic in the number of distinct decay time steps, which is bounded by the total number of elements ($|M|$). At any time step, the set D is those elements in the set indexed by the current time step that are decayed.

Given an arbitrary activation history of bounded size, it is likely that there is no

closed form solution for calculating L. Thus, to predict element decay, we perform a novel, two-phase process. After a new activation (S3), we first employ an approximation that is guaranteed to underestimate the true value of L. If, at a future time step, we encounter the element in D and it has not decayed, we then compute the exact prediction using a binary parameter search. This approach solves the forgetting problem correctly with respect to the base-level activation model, but, as our evaluation demonstrates, has the potential to considerably improve computational performance.

We approximate L for an element m as the sum of L for each independent pair $(a, k) \in m$. Here we derive the closed-form calculation: given a single element pair at time t, we solve for t_p, the future time of element decay:

$$ln(k \cdot [t-a]^{-d}) = \Theta$$
$$ln(k) - d \cdot ln(t_p + (t-a)) = \Theta$$
$$t_p = e^{\frac{\Theta - ln(k)}{-d}} - (t-a)$$

Since k refers to a single time point, a, we can rewrite the summed terms as a product. Furthermore, we time shift the decay term by the difference between the current time step, t, and that of the element pair, a, thereby predicting L.

The primary source of underestimation in our approximation is that, with respect to the exponential decay of linearly increasing time spans, the sum of logs is less than the log of a sum: for $x > 2$ and $y > 2$, $ln(x^{-d}+y^{-d}) > ln(x^{-d})+ln(y^{-d})$. Additionally, for each pair, we require that the solution of the approximation computation be a non-negative integer (i.e. a prediction of a future, discrete time step), which may not occur if the time step, a, is sufficiently old, and the number of activations at that time step, k, is relatively small. To satisfy this requirement, we apply the floor function to computations and discard negative values; these operations represent the secondary source of underestimation in this approach.

As an illustrative example, let us reconsider element $x = \{(3,1),(5,2)\}$ (see Section 6.3.1). Previously we computed that $L(x, 7, 0.5, -2) = 489$, and thus without further activation, element x should be considered decayed at time point $7 + 489 = 496$. If we were to apply our two-phase approach at time step $t = 7$, we first compute the approximation: 50 time steps for pair $(3,1)$ ($\lfloor 50.598 \rfloor$) and 216 time steps for pair $(5,2)$ ($\lfloor 216.393 \rfloor$), therefore approximating $L = 266$. Assuming no further activity, we would then re-examine element x at time step $t = 7 + 266 = 273$: $B(x, 273, 0.5) = -1.698$. Since the element is not decayed ($-1.698 > \Theta = -2$), we perform binary parameter search. Establishing the search range will require $\lceil log_2 223 \rceil = 8$ evaluations ($496 - 273 = 223$), and search within the range of $[128, 256]$ will require, on average, an additional $log_2 128 = 7$ evaluations. If the element had been removed (S2) before time point $t = 273$, the binary parameter search would not have been performed. If the element had been activated (S3) before time point $t = 273$, we would have used the approximation again, postponing the search. If our approach did not make use of the approximation, the binary parameter search would have required 1 additional evaluation to establish the range ($\lceil log_2 496 \rceil = 9$) and an additional 1 evaluation to search ($log_2 256 = 8$). Appendix I provides a concise description of this algorithm.

6.4.1 Analysis

Computing the approximation for a single pair takes constant time, $O(1)$. However, the absolute time for this computation may be significant as compared to other memory operations. Fortunately, for fixed values of parameters d and Θ, the result of the relatively expensive component of the computation, the exponential, can be cached for common values of k.

Overall approximation computation is linear in the number of pairs, which is bounded by c, and therefore $O(1)$. The computation required for binary parameter

search of an element is $O(log_2 L)$. However, this computation is only necessary if the element has not decayed, or removed from M.

This approach assumes that the number of memory activations in a single time step will be few, relative to the overall size of memory. This assumption regarding memory dynamics is closely related to *temporal contiguity* (discussed in context of episodic memory, Chapter IV), which claims that the world changes slowly, and thus changes to agent state, from episode to episode, will be few relative to the overall size of state. If temporal contiguity does not hold, this approach will likely perform comparably with a naïve forgetting method.

6.5 Evaluation

Our goal in this section is to understand the degree to which this forgetting framework supports useful operation across a variety of domains while scaling to large stores of knowledge and long agent lifetimes. We begin with a focussed evaluation, where we evaluate the quality and efficiency of our novel decay approximation on synthetic data. We then present two tasks where effective behavior requires that the agent accumulate large amounts of information from the environment, and where over time this learned knowledge overwhelms reasonable computational limits. In response, we have implemented our forgetting framework within the working and procedural memory mechanisms of Soar v9.3.2. These evaluations demonstrate how task-independent policies improve agent reactivity and scaling, while maintaining problem-solving competence.

6.5.1 Synthetic Data

In this section, we focus on the quality and efficiency of our prediction approach and utilize synthetic data. Our data set comprises 50,000 memory elements, each with a randomly generated pair set. The size of each element was randomly selected

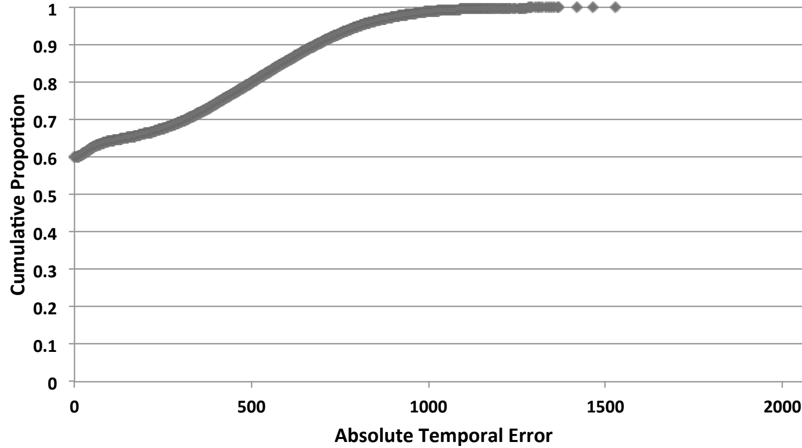

Figure 6.1: Synthetic: evaluation of decay-approximation quality.

from between 1 and 10, the number of activations per pair (k) was randomly selected between 1 and 10, and the time of each pair (a) was randomly selected between 1 and 999. We verified that each element had a valid history with respect to time step 1000, meaning that each element would not have decayed before $t = 1000$. Also, each element contained a pair with at least one access at time point 999, which simulated a fresh activation (S3). For all synthetic experiments we used decay rate $d = 0.8$ and threshold $\Theta = -1.6$. Given these constraints, the largest possible value of L for an element is 3,332.

We first evaluate the quality of the decay approximation. In Figure 6.1, the y-axis is the cumulative proportion of the elements and the x-axis plots absolute temporal error of the approximation, where a value of 0 indicates that the approximation was correct, and non-zero indicates how many time steps the approximation under-predicted. We see that the approximation was correct for over 60% of the elements, but did underestimate over 500 time steps for 20% of the elements and over 1000 time steps for 1% of the elements. Under the constraints of this data set, it is possible for this approximation to underestimate up to 2084 time steps.

We also compared the prediction time, in microseconds, of the approximation to an exact calculation using binary parameter search. The maximum computation

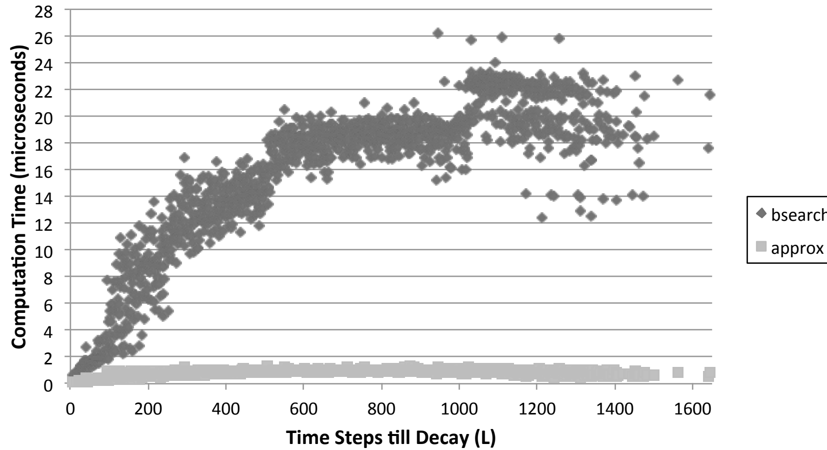

Figure 6.2: Synthetic: evaluation of decay-approximation complexity ("approx") as compared to binary parameter search ("bsearch").

time across the data set was $> 19x$ faster for the approximation (1.37 vs. 26.28 μsec./element) and the average time was $> 15x$ faster (0.31 vs. 4.73 μsec./element). We also confirmed our complexity analysis, as shown in Figure 6.2, which plots computation time versus L. We did not compare these results with a nave approach, as results would depend upon a model of memory size ($|M|$).

In summary, this evaluation provides evidence that our decay approximation improves computational performance of forgetting, as compared to binary parameter search: the experimental results show that the first phase of our implementation is a high-quality approximation and is an order of magnitude less costly than the exact calculation in the second phase.

6.5.2 Mobile Robotics

In this evaluation, we revisit the work with mobile robotics that was described in Section 5.5.3. The robot's task is to visit every room in a large building, for which it visits over 100 rooms and takes about 1 hour of real time. During exploration, it incrementally builds an internal topological map, which it reasons about and plans using in order to find efficient paths for moving to distant rooms it has sensed but

not visited. The agent uses episodic memory to recall objects and other task-relevant features during exploration.

In the previous experiment, we showed that semantic memory improved the agent's ability to reason in large domains over long periods of time while staying reactive to environmental dynamics. This was achieved by partitioning knowledge into local/short-term and distal/long-term, thereby improving episodic-reconstruction time. In that experiment, we provided the agent the necessary task knowledge of how to perform this partitioning. In this evaluation, we apply our task-independent forgetting framework to Soar's working memory, resulting in automatic working-memory pruning that is comparable to task-specific rules and improves decision-cycle time. We first describe the integration with Soar and then relate the results of our empirical evaluation.

6.5.2.1 Selective Retention in Working Memory

The core intuition of our working-memory retention policy is to remove the augmentations of objects that are not actively in use and that the model can later reconstruct from long-term semantic memory, if they become relevant. We characterize WME usage via the base-level activation model, whereby the primary activation event for a working-memory element is the firing of a rule that tests or creates that WME. Also, when a rule first adds an element to working memory, the activation of the new WME is initialized to reflect the aggregate activation of the set of WMEs responsible for its creation (such as to make sure newly created elements have an opportunity to participate in reasoning). This model of activation sources, events, and decay is task independent.

We now specify our working-memory selective-retention policy. At the end of each decision cycle, Soar removes from working memory each element that satisfies all of the following requirements, with respect to τ, a static, architectural threshold

parameter:

W1. The WME was not encoded directly from perception.

W2. The WME is operator-supported.

W3. The activation level of the WME is less than τ.

W4. The WME augments an object, o, in semantic memory.

W5. The activation of all augmentations of o are less than τ.

We adopted requirements W1-W3 from *Nuxoll et al.* (2004), whereas W4 and W5 are novel. Requirement W1 distinguishes between the decay of representations of perception, and any dynamics that may occur with actual sensors, such as refresh rate, fatigue, noise, or damage. Requirement W2 is a conceptual optimization: as operator-supported WMEs are persistent, while instantiation-supported structures are direct entailments, if we properly manage the former, the latter are handled automatically. This means that if we properly remove operator-supported WMEs, any instantiation-supported structures that depend on them will also be removed, and thus our mechanism only manages operator-supported structures. The concept of a fixed lower bound on activation, as defined by W3, was adopted from activation limits in ACT-R (*Anderson et al.*, 1996), and dictates that working-memory elements will decay in a task-independent fashion as their use for reasoning becomes less recent/frequent.

Requirement W4 dictates that our mechanism only removes elements from working memory that can be reconstructed from semantic memory. From the perspective of cognitive modeling, this constraint on decay resembles a working memory that is in part an activated subset of long-term memory (*Jonides et al.*, 2008). Functionally, requirement W4 serves to balance the degree of working-memory decay with support for sound reasoning. Knowledge in Soar's semantic memory is persistent,

though may change over time. Depending on the task and the models knowledge-management strategies, it is possible that any removed knowledge may be recovered via deliberate reconstruction from semantic memory. Additionally, knowledge that is not in semantic memory can persist indefinitely to support agent reasoning.

Requirement W5 supplements W4 by providing partial support for the *closed-world* assumption. W5 dictates that either all object augmentations are removed, or none. This policy leads to an object-oriented representation whereby procedural knowledge can distinguish between objects whose augmentations have been forgotten, and thus have no features or relations, and those that simply are not augmented with a particular feature or relation. W5 makes an explicit tradeoff, weighting more heavily model competence at the expense of the speed of working-memory decay. This requirement resembles the declarative module of ACT-R, where activation is associated with each chunk and not individual slot values.

6.5.2.2 Results

In our experiments, we aggregate working-memory size and maximum decision time for each 10 seconds of elapsed time, all of which is performed on an Intel i7 2.8GHz CPU with 8GB RAM, running Soar v9.3.2. Because each experimental run takes 1 hour, we did not duplicate our experiments sufficiently to establish statistical significance and the results we present are from individual experimental runs. However, we found qualitative consistency across our runs, such that the variance between runs is small as compared to the trends we focus on below.

We make use of the same agent for all experiments, but modify small amounts of procedural knowledge and change architectural parameters, as described here. The baseline model (A0) maintains all declarative map information both in Soar's working and semantic memories. A slight modification to this baseline (A1) includes hand-coded rules to prune away rooms in working memory that are not required for

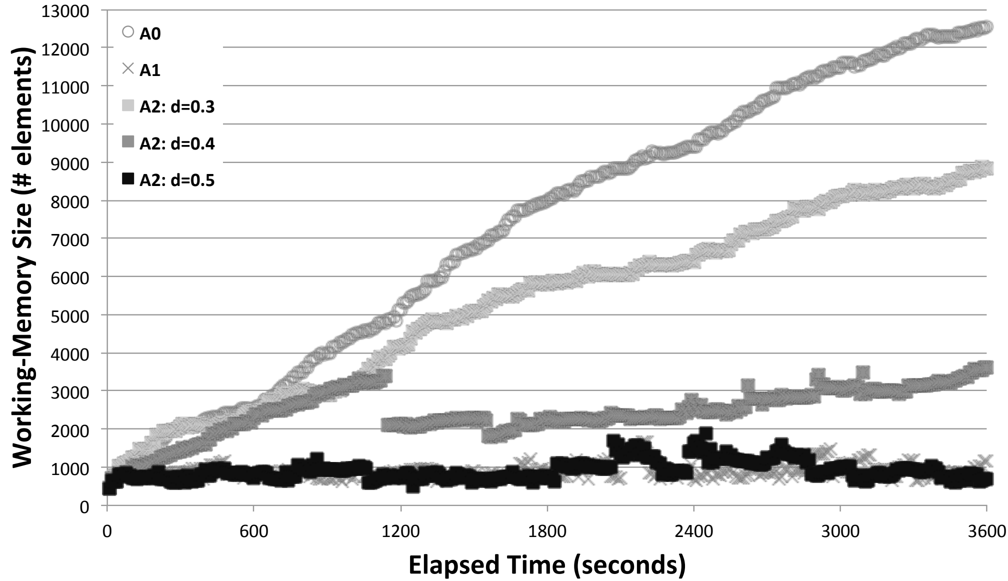

Figure 6.3: Mobile Robotics: working-memory size comparison.

immediate reasoning or planning (this was the "Semantic Memory" agent in Section 5.5.3). The experimental model (A2) makes use of our working-memory retention policy and we explored different values of the base-level decay rate ($c = 10$ and $\tau = -2$ for all models).

Figure 6.3 compares working-memory size between conditions A0, A1, and A2 over the duration of the experiment. We find that the greater the decay-rate parameter for A2, the smaller the working-memory size, where a value of 0.5 qualitatively tracks A1. This finding suggests that our policy, with an appropriate decay, keeps working-memory size comparable to that maintained by hand-coded rules.

Figure 6.4 compares maximum decision-cycle time in milliseconds, between conditions A0, A1, and A2 as the simulation progresses. The dominant cost reflected by this data is time to reconstruct prior episodes that are retrieved from episodic memory. We see a growing difference in time between A0 and A2 as working memory is more aggressively managed (i.e. greater decay rate), demonstrating that episodic reconstruction, which scales with the size of working memory at the time of episodic encoding, benefits from selective retention. We also find that with a decay rate of 0.5,

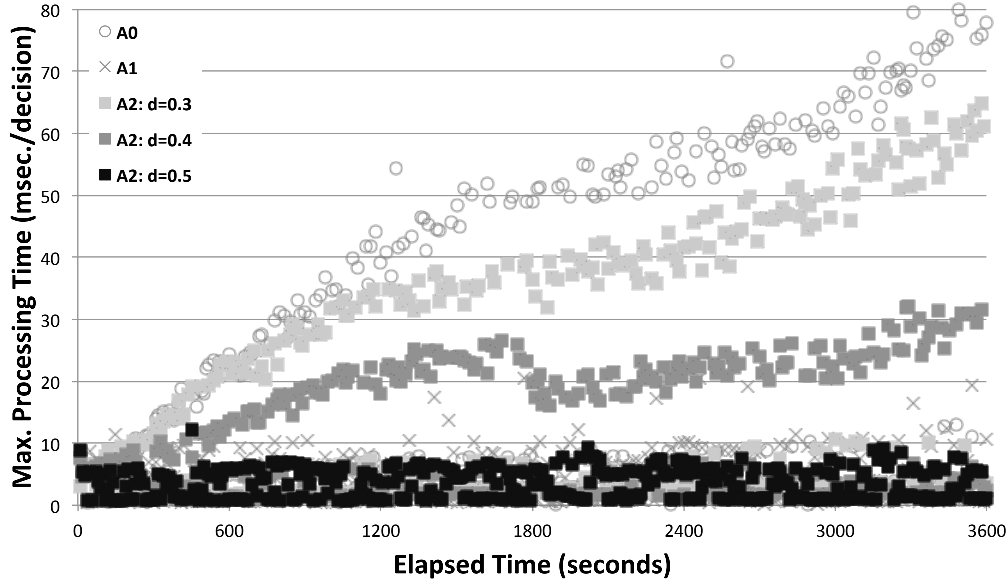

Figure 6.4: Mobile Robotics: maximum decision-time comparison.

our mechanism performs comparably to A1. We note that without sufficient working-memory management (A0; A2 with decay rate 0.3), episodic-memory retrievals are not tenable for a model that must reason with this amount of acquired information, as the maximum required processing time exceeds the reactivity threshold of 50 milliseconds.

6.5.2.3 Discussion

It is possible to write rules that prune Soars working memory; however, this task-specific knowledge is difficult to encode and learn, and interrupts deliberate processing.

In this evaluation, we presented and evaluated a novel approach that utilizes a memory hierarchy to bound working-memory size while maintaining sound reasoning. This approach assumes that the amount of knowledge required for immediate reasoning is small relative to the overall amount of knowledge accumulated by the model. Under this assumption, our policy scales even as learned knowledge grows large over long trials. We note that since Soar's semantic memory can change over

time and is independent of working memory, our selective-retention policy does admit a class of reasoning error wherein the contents of semantic memory are changed so as to be inconsistent with decayed WMEs. However, this corruption requires deliberate reasoning in a relatively small time window and has not arisen in our agents.

While the model completed this task for all conditions reported here, at larger decay rates (≥ 0.6) the model thrashed because sufficient map information was not held in working memory long enough to complete deep look-ahead planning. This suggests additional research is needed on either adaptive decay-rate settings or planning approaches that are robust in the face of memory decay.

6.5.3 Multiplayer Dice Game

In this evaluation, we extended an existing system (*Laird et al.*, 2011a) where Soar plays Liar's Dice, a multi-player game of chance. The rules of the game are numerous and complex, yielding a task that has rampant uncertainty and a large state space (millions-to-billions of relevant states for games of 2-4 players). Prior work has shown that RL allows Soar models to significantly improve performance after playing a few thousand games. However, this involves learning large numbers of RL rules to represent the state space and while Soar can remain reactive even with millions of rules, the associated computational memory growth is intractable. In this evaluation, we apply our task-independent forgetting framework to Soar's procedural memory, resulting in automatic rule forgetting that greatly decreases memory requirements, while maintaining efficient processing and competence in the task. We first describe the integration with Soar and then relate the results of our empirical evaluation.

6.5.3.1 Selective Retention in Procedural Memory

The intuition of our procedural-memory retention policy is to remove productions that are not actively used and that the model can later reconstruct via deliberate

subgoal reasoning, if they become relevant. We utilize the base-level activation model to summarize the history of rule firing.

At the end of each decision cycle, Soar removes from procedural memory each rule that satisfies all of the following requirements, with respect to parameter τ:

P1. The rule was learned via chunking.

P2. The rule is not actively firing.

P3. The activation level of the rule is less than τ.

P4. The rule has not been updated by RL.

We adopted P1-P3 from *Chong* (2004), whereas P4 is novel. *Chong* was modeling human skill decay, and did not delete productions, so as to not lose each rules activation history. Instead, decayed rules were prevented from firing, similar to below-utility-threshold rules in ACT-R. P1 is a practical consideration to distinguish learned knowledge from "innate" rules developed by the agent designer, which, if modified, would likely break the agent. P2 recognizes that matched rules are in active use and thus should not be forgotten. P3 dictates that rules will decay in a task-independent fashion as their use for reasoning becomes less recent/frequent. We note that for fixed parameters (d and τ) and a single activation, the base-level activation model is equivalent to the use-gap heuristic of *Kennedy and Trafton* (2007). However, the time between sequential rule firings ignores firing frequency, which the base-level activation model incorporates.

Requirement P4 attempts to retain only those rules that the agent cannot regenerate via chunking. Chunking is a process that compiles existing knowledge applied in subgoal reasoning. Chunked rules that have been updated by RL encode expected utility information, which is not captured by other learning mechanisms. Because this information is difficult, if not impossible, to reconstruct, these rules are retained.

6.5.3.2 Results

The agent we use for all experiments learns two classes of rules: RL rules, which capture expected action utility, and symbolic game heuristics. Our experimental baseline (B0) does not include selective retention. The first experimental modification (B1) implements our selective-retention policy, but does not enforce requirement P4 and is thereby comparable to prior work (*Kennedy and Trafton*, 2007; *Chong*, 2004). The second modification (B2) fully implements our policy. We experiment with a range of representative decay rates, including 0.999, where rules not immediately updated by RL are deleted ($c = 10$, $\tau = -2$ for all).

We alternated 1,000 2-player games of training then testing, each against a non-learning version of the agent. After each testing session, we recorded maximum memory usage (dominated, in this task, by procedural memory), task performance (% games won), and average decisions/task action. We do not report maximum decision time, as this was below 6 milliseconds for all conditions (Intel i7 2.8GHz CPU, 8GB RAM, Mac OS v10.7.3, Soar v9.3.2), which is well within our 50-milliseconds threshold. We collected data for all conditions in at least three independent trials of 40,000 games. For conditions that used selective retention, we were able to gather more data in parallel, due to reduced memory consumption (six trials for $d = 0.35$, seven for remaining).

Figure 6.5 presents average memory growth, in megabytes, as the agent trains. For all agents, the memory growth of games 1-10K follows a power law ($R^2 \geq 0.96$), whereas for 11-40K, growth is linear ($R^2 \geq 0.99$). These plots indicate that memory usage for the baseline (B0) and the slowly decaying agent (B2, $d = 0.3$) is much greater, and faster growing, than agents that more aggressively decay. It also shows that there is a diminishing benefit from faster decay (e.g. $d = 0.5$ and $d = 0.999$ for B2 are indistinguishable). Even small differences in this plot are significant: error bars of ± 1 standard deviation were not plotted, as they were not visible for all data

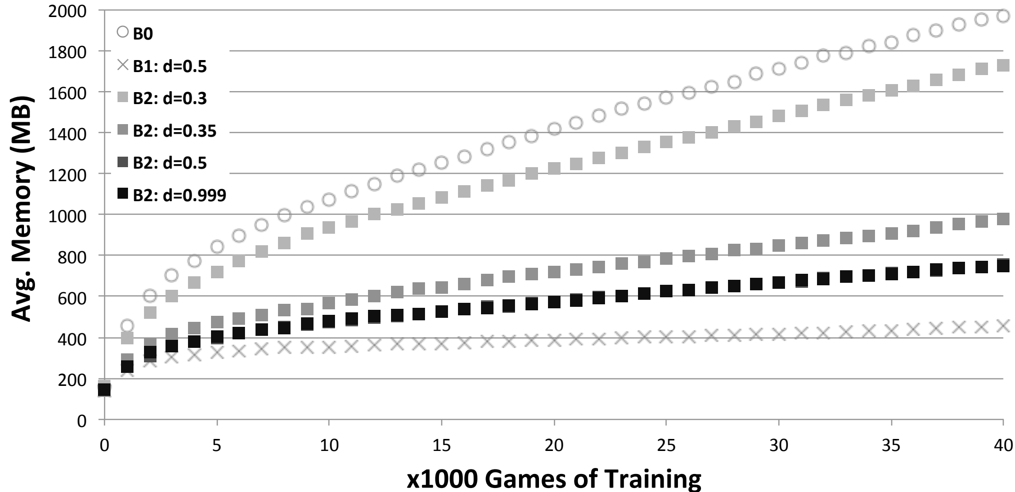

Figure 6.5: Dice: Average memory usage. Values of $d = 0.5$ and $d = 0.999$ for B2 are indistinguishable. Differences are significant: error bars of ± 1 standard deviation were not plotted, as they were not visible.

points.

Figure 6.6 presents average task performance after 1,000 games of training, where the error bars represent ± 1 standard deviation. This data shows that given the inherent stochasticity of the task, there is little, if any, difference between the performance of the baseline (B0) and decay levels of B2. There appears to be a slight trend of improvement in B2 as the decay rate increases; however, we have no mechanistic explanation for this difference and attribute it to fluctuations in the task. However, by comparing B0 and B2 to B1, it is clear that without P4, the agent suffers a dramatic loss of task competence. For clarity, the agent begins by playing a non-learning copy of itself and learns from experience with each training session. While the B0 and B2 agents improve from winning 50% of games to 75-80%, the B1 agent improves to below 55%. We conclude that a selective-retention policy that only incorporates production-firing history (e.g. *Chong*, 2004; *Kennedy and Trafton*, 2007) will negatively impact performance in tasks that involve informative interaction with an external environment. Our policy incorporates both rule-firing history and rule reconstruction, and thus retains this source of feedback.

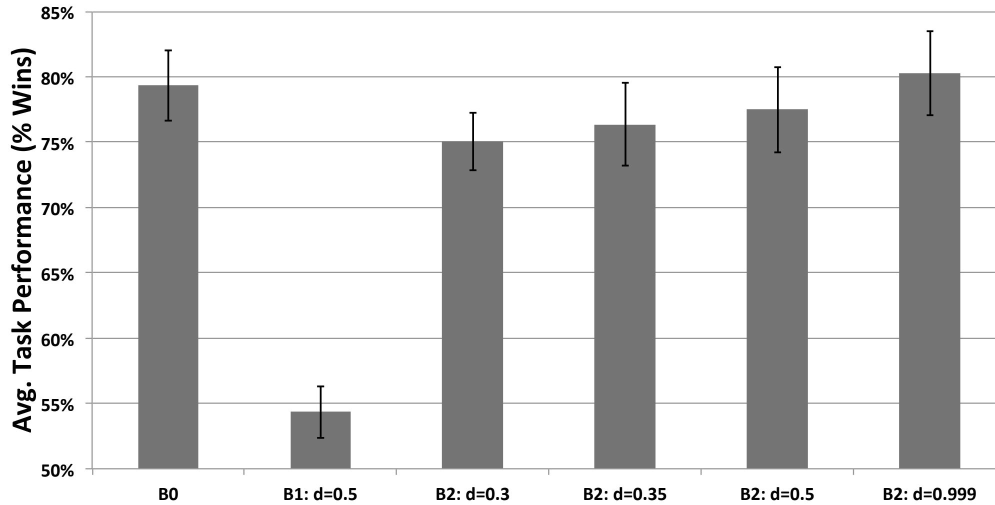

Figure 6.6: Dice: Average task performance (% wins) ±1 standard deviation.

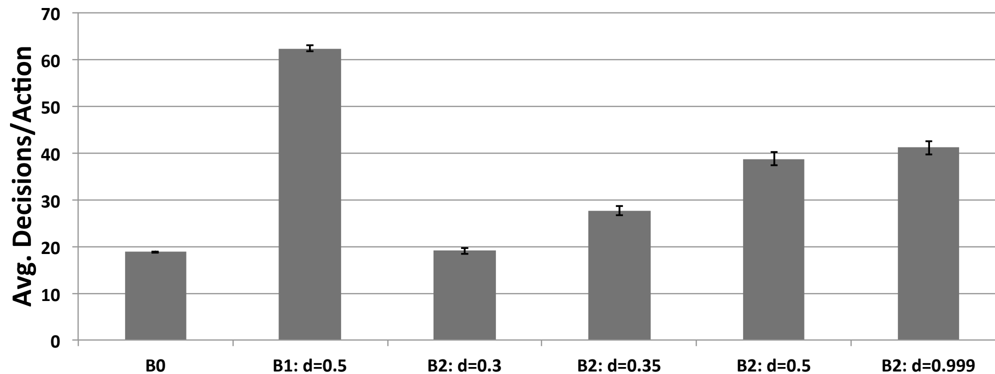

Figure 6.7: Dice: Average decisions/task action ±1 standard deviation.

Finally, Figure 6.7 presents average number of decisions for the model to take an action in the game after training for 10,000 games. In prior work (e.g. *Kennedy and Trafton*, 2007), this value was a major performance metric, as it reflected the primary reason for learning new rules. In this work, each decision takes very little time, and so the number of decisions to choose an action is not as crucial to task performance as the selected action. However, these data show that there exists a space of decay values (e.g. $d = 0.35$) in which memory usage is relatively low and grows slowly (Figure 6.5), task performance is relatively high (Figure 6.6), and the model makes decisions relatively quickly (Figure 6.7).

6.5.3.3 Discussion

This evaluation proposes an approach to developing agents that improve using RL in tasks with large state spaces. Currently, it is typical to explicitly represent the entire state space in the value function, which is not feasible in complex problems. Instead, Soar learns rules to represent only those portions of the space it experiences, and our policy retains only those rules that include feedback from environmental reward. Future work needs to validate this approach in other domains.

6.6 Discussion

In this chapter, we presented a framework to forget items of knowledge in memory mechanisms while maintaining agent task proficiency. The main hypothesis of the framework is that a cognitive architecture should remove knowledge that is likely not useful, according to the base-level activation model, and that can likely be reconstructed if necessary. We also presented and evaluated task-independent (R6) algorithms to efficiently integrate this within a cognitive architecture; we evaluated this mechanism within the working- and procedural-memory systems of Soar, and showed that they reduced computational-resource consumption (R4), while maintaining agent task proficiency (R5), in two complex tasks that required a great deal of environmental learning over long periods of time.

6.6.1 Future Work

There are many other facets of forgetting that are ripe for research. For example, in this work, we estimated importance via the base-level activation mode, which estimates future importance solely on the history of past usage. However, within a general cognitive architecture there likely are additional sources of predictive information about knowledge utilization and importance. For example, it is well documented

that for humans, emotionally charged events are better remembered than neutral ones (e.g. *Paré*, 2003), and so it would be useful to explore the ways in which forgetting policies that incorporate emotional appraisals (*Marinier et al.*, 2009) and agent arousal (*Nuxoll et al.*, 2004) would impact the computational resource versus task performance tradeoff. This work also assumes that forgetting should function uniformly over time. However, given a hierarchy of multiple memory mechanisms over long periods of time, there may be benefits in gradational approaches to forgetting policies, with respect to time scales. Additionally, this work posits a static threshold of forgetting, but more adaptive policies could be productive, as well as agents that can learn strategies to exploit knowledge about their own selective-retention policies.

Markovitch and Scott (1988) laid out a framework for contending with large-scale learning systems, of which selective retention is just one layer. Chapters (IV and V) allude to the possibility that selective acquisition is another important avenue for research, as well as the interaction between encoding and forgetting processes in a long-lived agent. It seems likely that jointly considering the benefits of both these processes, in context of the requirements of generally intelligent agents, would lead to a space of robust policies for incrementally building up useful knowledge, while adhering to the computational limits of bounded agents.

Of course, to make progress on these lines of inquiry will require a much broader empirical evaluation than was presented in this chapter. A specific weakness of this work was that the agents only worked on a single task: to make viable progress towards effective and efficient selective-policies for generally intelligent agents, however, we need much more complex sets of evaluation tasks, including those that are novel to agents.

CHAPTER VII

Summary and Conclusion

In this chapter, we revisit the memory requirements imposed by generally intelligent agents and discuss the progress we've made, as well as directions for future work. We then summarize the contributions of this dissertation and conclude.

7.1 Requirements Revisited

- **R1. Incremental Learning**

  In Chapter IV we presented and evaluated a task-independent episodic-memory mechanism that incrementally encodes an agent's autobiographical history. We demonstrated that for many tasks and domains, taking advantage of temporal contiguity and structural regularity allows an agent's cognitive architecture to keep pace with environmental dynamics, supporting online storage and effective retrievals.

  In Chapter V we presented and evaluated a task-independent semantic-memory mechanism that allows an agent to deliberately encode facts and relations about the world that are independent of the context in which they were learned. We demonstrated that for many tasks and domains, taking advantage of small object cardinality and locally efficient biases allows an agent's cognitive architecture to keep pace with environmental dynamics, supporting online storage and

effective retrievals.

In Chapter VI we presented and evaluated a task-independent framework for allowing an agent to reduce the computational burden of large-scale learning over long lifetimes. We demonstrated in two complex tasks that applications of this framework can improve an agent's ability to keep pace with environmental dynamics, while not impeding task competence.

- **R2. Comprehensive Learning**

  Chapters IV and V present and evaluate methods to integrate episodic and semantic learning that scale to large amounts of knowledge over long agent lifetimes. Prior work had recognized these forms of learning as functional in specific problems, whereas this work demonstrated that generally intelligent agents could make use of these mechanisms across a variety of tasks in complex domains.

- **R3. Diverse Representations**

  Chapters IV, V, and VI support symbolic representations that were demonstrated as sufficiently general for a variety of tasks. However, future work should investigate additional classes of representation, including those that support continuous modalities (e.g. visual, auditory, etc.).

- **R4. Scale Efficiently**

  The functional specifications that we presented in Chapters IV and V applied strong constraints regarding the fidelity of stored knowledge and correctness of retrievals with respect to relatively general query semantics. Consequently, both the episodic and semantic memory models we presented have complexity profiles that will not scale efficiently in the general case. However, we showed that under certain assumptions of tasks, domains, and agent cues, these mechanisms can work well in practice for a broad range of tasks, and we have provided predictive

performance models to understand these bounds.

We expect that these methods will open up a new class of tasks and domains with which researchers and system builders can apply intelligent agents. As agent developers and cognitive modelers scale up, they will better understand common use cases and regularities of domains and tasks, informing the next wave of research in effective and efficient memory mechanisms, which will likely entail heuristic and approximate methods. The work in Chapter VI, for example, illustrates the flavor of research that is motivated by the need to scale existing memory systems to complex domains and long agent lifetimes.

- **R5. Effective Access**

  The functional specifications that we presented in Chapters IV and V supported qualitative query semantics, which we demonstrated are useful for a variety of problems and domains. As agent developers increasingly apply these methods, however, they will likely identify new classes of useful retrievals. This flavor of iterative research and development is common in cognitive-architecture investigations, wherein functional requirements drive effective and efficient methods which open up the exploration of new, more challenging problems and capabilities (e.g. *Laird and Rosenbloom*, 1996). Chapter VI is one example of the challenges associated with simultaneously focussing on scalable methods and effective access to task-relevant knowledge.

- **R6. Task Independence**

  Chapters IV, V, and VI support representations and processes that are independent of task and domain. To improve average-case performance, we have made very general assumptions regarding state representations and dynamics and have shown that these regularities are common in practice, and lead to crucial advances in effective and efficient mechanisms.

7.2 Future Work

The concluding sections of Chapters IV, V, and VI discussed focussed directions for future research. Here we discuss more broad, integrative avenues of work.

7.2.1 Memory Space

We have investigated just two points in a large space of memory mechanisms (see Appendix A). There are several other interesting regions for investigation, such as effective and efficient declarative memory mechanisms for long-term goals (e.g. *Li and Laird*, 2011) and emotion (e.g. *Gomes et al.*, 2011). Additional work must also be done for modality-specific memories, such as visual and auditiory. It is likely that stores of knowledge from continuous modalities will require very different query semantics for useful retrievals, as well as underlying implementations for efficient scaling.

7.2.2 Mechanism Integration

As we explore additional regions of memory space, and develop effective and efficient implementations for generally intelligent agents, thoughtful integration of multiple, dissociated mechanisms will become an important research consideration. One important issue that will arise is knowledge consistency: what should a cognitive architecture and/or generally intelligent agent do when knowledge exists from multiple stores that are contradictory? To contend with this problem, research for generally intelligent agents will likely have to pool approaches from multiple fields, such as work in knowledge-based systems that have dealt with automated methods for aspects of this problem in unified stores (e.g. *Lenat*, 1995; *Fahlman*, 2006); database systems that have considered this problem in context of human interaction for data integration (e.g. *Lenzerini*, 2002); and multi-agent systems that consider methods for solving constrained problems in a distributed fashion (e.g. *Yokoo and Hirayama*, 2000).

7.2.3 Agent Development

Most intelligent systems today contend with a single, well-defined task and/or a short lifetime. As these systems persist for long periods of time, utilizing memory mechanisms for work on multiple, complex tasks in dynamic domains, there are numerous research challenges that will arise in the development of agents. For instance, how do agents develop strategies to best make use of multiple mechanisms, given an evolving knowledge of their tradeoffs in efficiency, utility, and fidelity? Furthermore, how can agents pool knowledge from dissociated stores and generalize experience that can transfer to new problems? There are also numerous practical considerations of experimenting with, debugging, and evaluating long-living, learning systems, especially when embedded within stochastic environments.

7.3 Research Contributions

The following list summarizes our major research contributions:

- **Analysis of general properties of environments, tasks, and agents** that impact the development and utilization of effective and efficient memory mechanisms. Chapters IV and V describe how these regularities lead to efficient algorithms, as well as describe and evaluate predictive models of how they impact performance of running systems.

- Development of novel, and adaptation of existing, **algorithms** for memory mechanisms that support effective access to agent experience while scaling, for many tasks, to large amounts of knowledge over long agent lifetimes. Chapters IV, V, and VI describe knowledge representation, data structures, algorithms, complexity analysis, and issues of integration within a general cognitive architecture. For episodic memory, we developed a novel dynamic-graph index to efficiently support storage of state changes, as well as a novel discrimination

network (the DNF Graph) to support efficient, incremental scoring of episodes in surface cue matching. For semantic memory, we extended work on inverted indexes and statistical query optimization to scalably and efficiently support a useful class of memory-retrieval bias functions (those that are *locally efficient*). We also developed an algorithm to efficiently and correctly forget memories in large memory stores according to the base-level decay model. We have implemented all of this work in Soar v9.3.2, which is available as open source software for free download, and researchers have already made use of these mechanisms for independent work (e.g. *Laird et al.*, 2010; *Xu and Laird*, 2010; *Gorski and Laird*, 2011; *Li and Laird*, 2011; *Xu and Laird*, 2011).

- **Broad evaluation** of our methods across a variety of problem domains, including linguistic tasks (e.g. word sense disambiguation, lexical queries), planning problems, games (e.g. Infinite Mario, TankSoar, Eaters, Liar's Dice), and mobile robotics. Chapters IV, V, and VI describe experimental conditions and results from scaling the amount of knowledge and runtime in these domains to orders of magnitude larger and longer than previously reported in the research.

- **Demonstrations of how agents that are endowed with effective and efficient memory benefit across numerous tasks**. These benefits include support of general cognitive capabilities (e.g. Chapter IV), better reasoning in task domains (e.g. Chapter V), and improved ability to scale to complex problems (e.g. Chapter VI).

7.4 Conclusion

The goal of this dissertation has been to investigate architectural mechanisms to support memory that scales to large amounts of knowledge and long agent lifetimes. This has led to scalable, task-independent algorithms that improve the scope

of learning and effectiveness of agent reasoning, as demonstrated in a variety of problem domains. Much like Rete (*Forgy*, 1982), these algorithms commit to very general knowledge representations and operations, but provide efficient and useful access to kinds of knowledge that are vital for generally intelligent agents. As a result, the analyses and algorithms from this dissertation extend to cognitive architectures other than Soar, as well as other agent-based systems that need to incorporate effective and efficient memory functionality.

Theoretically, it is only those agents with access to prior experience that can understand and exploit regularities of the world around them. More practically, effective and efficient memory facilitates the development of intelligent systems that are more adaptive, more robust, and longer lived, as well as modeling cognition in a wider variety of task domains. By supporting large stores of knowledge, and long agent lifetimes, this work expands the types of agents that can be developed and the types of phenomena that can be modeled. While it is difficult to predict the degree to which this dissertation will affect future research, I hope that this work provides a computationally sound infrastructure upon which to understand and develop human-level intelligence.

APPENDICES

APPENDIX A

The Space of Memory Models

Here we detail an initial characterization of the space of long-term memory systems, borrowing heavily from and generalizing Nuxolls breakdown of the space of episodic memory systems 2007. We define a memory-system implementation as a commitment to each feature from within the space defined by these dimensions, represented as (*encoding*, *storage*, *retrieval*). [1]

A.1 Encoding

Initiation – Initiation encompasses the event conditions that trigger the encoding and storage processes. These events may condition upon fixed architectural characteristics of state (such as a temporal frequency) or may be accessible to agent control knowledge.

Determination – Once initiated, the memory mechanism selects features of agent state (or derivation thereof) that compose the knowledge to be stored, as well as any additional context (temporal, spatial, etc) that may also be associated with the knowledge.

[1] This space was originally published in *Derbinsky and Gorski* (2010).

A.2 Storage

Granularity – Stored experience varies with the grain size at which knowledge can accessed and modified. This may range from minute (such as the symbol level), to moderate (an episode), to coarse (such as the entire knowledge store).

Dynamics – Knowledge in the memory system may change over time, such as to bias retrieval or forget knowledge. The mechanisms that cause this change may be fixed, condition upon agent knowledge, or deliberate agent action.

A.3 Retrieval

Accessibility – Experience encoded within the memory system may vary in the degree to which it is exposed to other architectural mechanisms, such as to maintain overall agent reactivity. For instance, a declarative long-term memory may allow for enumeration of all stored memories.

Initiation – Initiation encompasses the event conditions that trigger the retrieval process. As with encoding initiation, these events may condition upon fixed characteristics of state or may be accessible to agent knowledge/control.

Cue Determination – Once initiated, the memory system composes agent state, knowledge, context, and/or [possibly inaccessible] meta-data to select or create a retrieval cue.

Selection – When supplied a cue, the memory system implements a policy for how stored knowledge is matched with respect to the cue, which may be restricted by time, computation, and/or number of results, as well as include bias from agent state, context, and/or meta-data.

Result – When the memory system selects stored experience for retrieval, it

may arbitrarily represent the knowledge, associated context, and aspects of the retrieval process, such as match quality, for agent inspection.

APPENDIX B

Episodic Memory: Relational Schemas

The tables below detail the relational tables for episodic memory (Chapter IV). A "type" of "any" denotes no field type constraint.

B.1 Episode Registration (*times*)

Stores the temporal ids of stored episodes.

Field	Type	Notes
id	integer	temporal id

B.2 Temporal Symbol Hash (*temporal_symbol_hash*)

Stores hashes from a (constant, type) pair to a single integer.

Field	Type	Notes
id	integer	symbol hash
sym_const	any	symbolic constant
sym_type	integer	symbol type

B.3 Persistent Variables (*vars*)

Stores persistent variables as key/value pairs.

Field	Type	Notes
id	integer	variable key
value	any	variable value

B.4 WMG: Identifier-Valued WMEs (*edge_unique*)

Stores distinct WMEs that have identifiers (i.e. non-constant) values.

Field	Type	Notes
parent_id	integer	unique WME id
q0	integer	existing graph node (value 0 is root; refers to other values of *q1*)
w	integer	WME attribute (refers to *id* in temporal symbol hash)
q1	integer	existing/new graph node (value 0 is root; refers to other values of *q1* for existing)

B.5 WMG: Constant-Valued WMEs (*node_unique*)

Stores distinct WMEs that have constant values.

Field	Type	Notes
child_id	integer	unique WME id
parent_id	integer	existing graph node (value 0 is root; refers to other values of *q1*)
attrib	integer	WME attribute (refers to *id* in temporal symbol hash)
value	integer	WME value (refers to *id* in temporal symbol hash)

B.6 Long-Term Identifier Registration (*lti*)

Stores long-term identifier information.

Field	Type	Notes
parent_id	integer	unique id (refers to *parent_id* in edge_unique)
letter	integer	ASCII value of identifier letter
num	integer	identifier number
time_id	integer	temporal id of promotion from short- to long-term identifier (refers to *id* in times)

B.7 Intervals: Identifier-Valued WMEs, Now (*edge_now*)

Stores the temporal id of the time at which identifier-valued WMEs that are presently in working-memory were added.

Field	Type	Notes
id	integer	unique id (refers to *parent_id* in edge_unique)
start	integer	temporal id when WME was added to working memory (refers to *id* in times)

B.8 Intervals: Identifier-Valued WMEs, Point (*edge_point*)

Stores the temporal interval for identifier-valued WMEs that persisted for only a single episode.

Field	Type	Notes
id	integer	unique id (refers to *parent_id* in edge_unique)
start	integer	temporal id when WME was added to working memory (refers to *id* in times)

B.9 Intervals: Identifier-Valued WMEs, Range (*edge_range*)

Stores the temporal interval for identifier-valued WMEs that persisted for more than a single episode.

Field	Type	Notes
id	integer	unique id (refers to *parent_id* in edge_unique)
start	integer	temporal id when WME was added to working memory (refers to *id* in times)
end	integer	temporal id when WME was removed from working memory (refers to *id* in times)
rit_node	integer	related to relational-interval tree hashing

B.10 Intervals: Constant-Valued WMEs, Now (*node_now*)

Stores the temporal id of the time at which constant-valued WMEs that are presently in working-memory were added.

Field	Type	Notes
id	integer	unique id (refers to *child_id* in node_unique)
start	integer	temporal id when WME was added to working memory (refers to *id* in times)

B.11 Intervals: Constant-Valued WMEs, Point (*node_point*)

Stores the temporal interval for node-valued WMEs that persisted for only a single episode.

Field	Type	Notes
id	integer	unique id (refers to *child_id* in node_unique)
start	integer	temporal id when WME was added to working memory (refers to *id* in times)

B.12 Intervals: Constant-Valued WMEs, Range (*node_range*)

Stores the temporal interval for constant-valued WMEs that persisted for more than a single episode.

Field	Type	Notes
id	integer	unique id (refers to *child_id* in node_unique)
start	integer	temporal id when WME was added to working memory (refers to *id* in times)
end	integer	temporal id when WME was removed from working memory (refers to *id* in times)
rit_node	integer	related to relational-interval tree hashing

B.13 Relational-Interval Tree: Left (*rit_left_nodes*)

Temporary table for relational-interval tree (left).

Field	Type	Notes
min	integer	
max	integer	

B.14 Relational-Interval Tree: Right (*rit_right_nodes*)

Temporary table for relational-interval tree (right).

Field	Type	Notes
node	integer	

APPENDIX C

Episodic Memory: Algorithms

The sections below detail the algorithms for episodic memory (Chapter IV).

C.1 Assumptions

I. Agent state is represented as a directed, connected graph rooted at node n_0.

II. The state graph is a set: no two edges can have the same triple (parent, label, value).

III. Episodes are automatically encoded and stored at some interval, which will include zero or more changes to the graph, which can be captured as a sequence of edge additions or removals.

IV. A retrieval cue is a rooted, directed, connected, acyclic graph.

V. Episodic retrieval returns the most recent episode that has the greatest number of edges in common with cue leaves, with respect to root-to-leaf path.

VI. Symbolic constants (i.e. numbers and strings) have persistent identity that must be maintained for accurate episodic reconstruction, while graph nodes do not (all that is important is their relations to other nodes).

C.2 Storage

```
list EdgeAdditions, EdgeRemovals
WorkingMemoryGraph WMG
state.root.wmgNode ← WMG.newNode()

function OnEdgeAddition(edge e)
  EdgeAdditions.insert(e)
end function

function OnEdgeRemoval(edge e)
  if EdgeAdditions.contains(e): EdgeAdditions.remove(e)
  else: EdgeRemovals.insert(e)
end function

function GetWMGEdge(edge e)
  for wmge in e.parent.wmgNode.outgoingEdges()
    if ((wmge.label is e.label) and (wmge.inUse is false)) and
        (isNode(wmge.value) is isNode(e.value))
      if (isNode(e.value) and (e.value.wmgNode is null)): return wmge
      elseif (wmge.value is e.value): return wmge
    end if
  end for
  return null
end function
```

```
function StoreEpisode(time t)
  list IntervalStarts, IntervalEnds

  for e in EdgeRemovals
    IntervalEnds.insert(e.wmgEdge)
    e.wmgEdge.inUse ← false
  end for
  EdgeRemovals.clear()

  while not EdgeAdditions.empty()
    for e in EdgeAdditions
      if (not (e.parent.wmgNode is null))
        WMGEdge wmge ← GetWMGEdge(e)
        if (not (wmge is null))
          e.wmgEdge ← wmge
          wmge.inUse ← true
          if (IntervalEnds.contains(wmge)): IntervalEnds.remove(wmge)
          else: IntervalStarts.insert(wmge)
        else
          if (isNode(e.value) and (e.value.wmgNode is null)):
            e.value.wmgNode ← WMG.newNode()
          wmge ← WMG.insertEdge(e)
          e.wmgEdge ← wmge
          wmge.inUse ← true
          IntervalStarts.insert(wmge)
        end if
```

```
        EdgeAdditions.remove(e)
      end if
    end for
  repeat

  for wmge in IntervalStarts
    wmge.startInterval(t)
  end for
  for wmge in IntervalEnds
    wmge.endInterval(t)
  end for
end function
```

C.2.1 Notes

Encoding occurs incrementally in response to state changes. As state edges are added/removed, episodic memory maintains lists of these changes.

The storage process inspects the state-change lists. For added edges, those that were not previously stored in episodic memory are added to the Working-Memory Graph (WMG). Once all state edges are associated with WMG edges, the interval lists are appropriately annotated with respect to the current time. Intervals are added to a common interval tree, utilized during the reconstruction process.

This algorithm includes two optimizations. First, before creating a new WMG edge, prior edges that are not in use for the current episode are tested (for possible representational sharing). Second, new state edges can *continue* an existing interval if the WMG edge is shared.

C.3 Cue Matching

```
ValueMappingSAT values

EdgeSAT triggers

EndpointPriorityQueue epq

currentscore ← 0

currentepisode ← null


function GetWMGEdges(WMGNode wmgn, edge e)
  ret = {}
  for wmge in wmgn.outgoingEdges()
    if (wmge.label is e.label)
      if (isNode(e.value) and isNode(wmge.value)): ret.insert(wmge)
      elseif (wmge.value is e.value): ret.insert(wmge)
    end if
  end for
  return ret
end function


function AddValueMapping(WMGEdge wmge, CueEdge ce)
  if (not values.containsKey(<ce.value,wmge.value>)):
    values.insert(<ce.value,wmge.value>, [cueedges={}, prop={},
                                          ct=0, perfect=0])

  values[<ce.value,wmge.value>].cueedges[ce] ← 0
  values[<ce.value,wmge.value>].perfect ←
    values[<ce.value,wmge.value>].perfect + 1
end function
```

```
function AddTrigger(WMGEdge wmge, CueEdge ce)
  if (not triggers.containsKey(wmge)): triggers.insert(wmge, {})
  triggers[wmge].insert(<<ce.value,wmge.value>, ce>)
end function

function AddDNFMapping(CueNode cn, WMGNode wmgn)
  propset ← {}
  for ce in cn.outgoingEdges()
    for wmge in GetWMGEdges(wmgn, ce)
      AddValueMapping(wmge, ce)
      AddTrigger(wmge, ce)
      propset.insert(<ce, wmge>)
      if (isNode(ce.value) and (not isLeaf(ce)))
        backpropset ← AddDNFMapping(ce.value, wmge.value)
        for p in backpropset
          values[<ce.value,wmge.value>].prop.insert(p)
        end for
      end if
    end for
  end for
  return propset
end function

function BuildDNFGraph(cue c)
  AddDNFMapping(c.root, n0)
end function
```

```
function LoadPriorityQueue()
  for t in triggers
    pq.insertWithPriority(
      <t.key, t.key.intervals.size()>,
       EpFromEndpointIndex(t.key, t.key.intervals.size()))
  end for
end function

function UpdateValueMappingScore(CueEdge ce, WMGEdge wmge, bool nowSatisfied)
  inSAT ← values[<ce.value,wmge.value>].cueedges[ce]
  if nowSatisfied
    literalOn ← (inSAT is 0)
    values[<ce.value,wmge.value>].cueedges[ce] ← (inSAT + 1)
    if literalOn
      oldCt ← values[<ce.value,wmge.value>].ct
      propagate ← (oldCt is (values[<ce.value,wmge.value>].perfect - 1))
      values[<ce.value,wmge.value>].ct ← (oldCt + 1)
      if propagate
        if isLeaf(ce): currentscore ← (currentscore + 1)
        else
          for p in values[<ce.value,wmge.value>].propset
            UpdateValueMappingScore(p.first, p.second, true)
          end if
        end for
      end if
    end if
```

```
else
  literalOff ← (inSAT is 1)
  values[<ce.value,wmge.value>].cueedges[ce] ← (inSAT - 1)
  if literalOff
    oldCt ← values[<ce.value,wmge.value>].ct
    propagate ← (oldCt is values[<ce.value,wmge.value>].perfect)
    values[t.first].ct ← (oldCt - 1)
    if propagate
      if isLeaf(ce): currentscore ← (currentscore - 1)
      else
        for p in values[<ce.value,wmge.value>].propset
          UpdateValueMappingScore(p.first, p.second, false)
        end if
      end for
    end if
  end if
end if
end function

function UpdateEpisodeScore(WMGEdge wmge, bool nowSatisfied)
for t in triggers[wmge]:
  UpdateValueMappingScore(t.first.first, t.first.second, nowSatisfied)
end function

function GotoNextRelevantEpisode()
p ← pq.peek
currentepisode ← EpFromEndpointIndex(p.first.intervals, p.second)
```

```
  while (EpFromEndpointIndex(pq.peek.first.intervals, pq.peek.second)
         is currentepisode)
    p ← pq.pop()
    if (p.second > 0):
      pq.insertWithPriority(<p.first, p.second-1>,
        EpFromEndpointIndex(p.first.intervals, p.second-1))
    UpdateEpisodeScore(p.first, (p.first.intervals[p.second] is end))
  repeat
end function

function MatchCue(cue c)
  time king ← null
  highscore ← −∞
  perfectscore ← 0
  done ← false

  for ce in c
    if isLeaf(ce): perfectscore ← perfectscore + 1
  end for
  BuildDNFGraph(c)
  LoadPriorityQueue()

  while (done is false)
    GotoNextRelevantEpisode()
    if (currentscore > highscore)
      highscore ← currentscore
      king ← currentepisode
      if (currentscore is perfectscore)
```

```
      if GraphMatch(c, currentepisode): done ← true
    end if
  end if
  if pq.empty(): done ← true
repeat

if (king is null): failure
else: return king
end function
```

C.3.1 Notes

The DNF Graph builds incremental SAT structures that associate values of edges in the WMG and cue. These structures include satisfaction of incoming cue edges, propagation directives, and state of satisfaction (current value and perfect).

Based upon these data structures, the trigger map identifies which interval lists are to be walked, and which value maps will be updated by each endpoint. These triggers are loaded into a priority queue, keyed on the episode of the most recent endpoint.

The interval-walking algorithm pulls all interval endpoints for the next episode relevant to the cue. For each endpoint that is walked, the episode is scored incrementally via DNF-Graph updates.

Once all the endpoints for the episode are pulled from the priority queue, the episode is assessed as a surface match (i.e. all features independently via DNF Graph). If it is greater than any previous episodes, it becomes the best match thus far. If it is perfect as a surface match, it will undergo structural match (i.e. GraphMatch) and, if it passes, search is concluded. Otherwise, search continues until the priority queue is devoid of endpoints to walk.

C.4 Reconstruction

```
function ReconstructEpisode(time t)
  list wmgedges, epedges

  wmgedges = IntervalIntersectionQuery(t)
  for wmge in wmgedges
    epedges.insert(<wmge.parent, wmge.label, wmge.value>)
  end for

  return epedges
end function
```

C.4.1 Notes

Finding all edges that comprise an episode is an interval-intersection query (i.e. find all temporal intervals that intersect a single point in time). To support executing this query efficiently, we implement a Relational-Interval Tree (*Kriegel et al.*, 2000) over Working-Memory Graph intervals. Once the intervals are identified, we use the structure of the WMG to reproduce edge labels/relations.

APPENDIX D

Episodic Memory: Evaluation Cues

These cues were used for the evaluation in Chapter IV.

D.1 TankSoar

1. (<cue> ^superstate nil)

2. (<cue> ^io.input-link.health 1000)

3. (<cue> ^io.input-link.x 1)

4. (<cue> ^io.input-link.y 1)

5. (<cue> ^io.input-link <il>)

   (<il> ^x 1 ^y 1)

6. (<cue> ^io.input-link.radar <r>)

7. (<cue> ^io.input-link.radar.open <o>)

8. (<cue> ^io.input-link.radar.open.position center)

9. (<cue> ^square <sq>)

10.
```
(<cue> ^square <sq>)
(<sq> ^x 1 ^y 1)
```

11.
```
(<cue> ^map.square <sq>)
```

12.
```
(<cue> ^map.square <sq>)
(<sq> ^x 1 ^y 1)
```

13.
```
(<cue> ^io.input-link <il>)
(<il> ^x 1 ^y 8 ^direction south ^radar-status on ^radar-setting 13)
```

14.
```
(<cue> ^io.input-link.radar.missiles <m>)
```

15.
```
(<cue> ^io <io>)
(<io> ^input-link <il>
      ^output-link <ol>)
(<ol> ^rotate.direction left ^radar.switch on ^radar-power.setting 13)
(<il> ^blocked <b>)
(<b> ^forward yes ^left no)
```

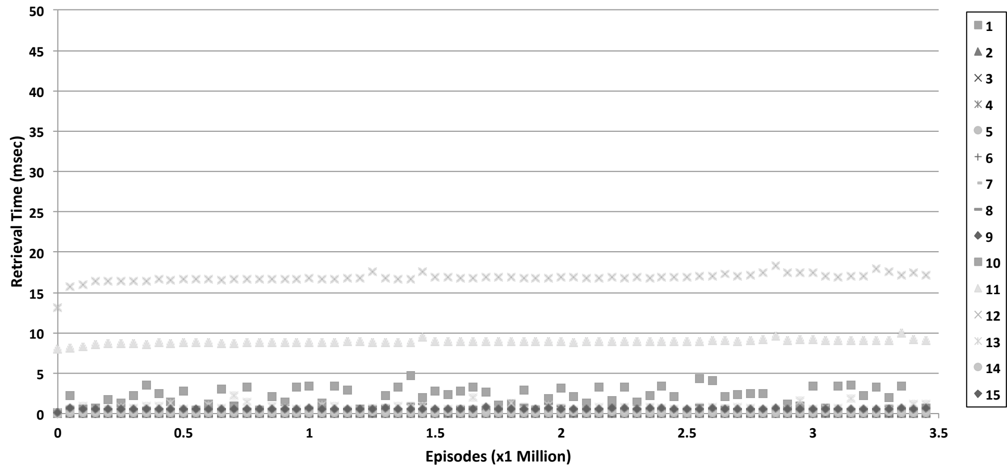

Figure D.1: TankSoar: timing data for all cues.

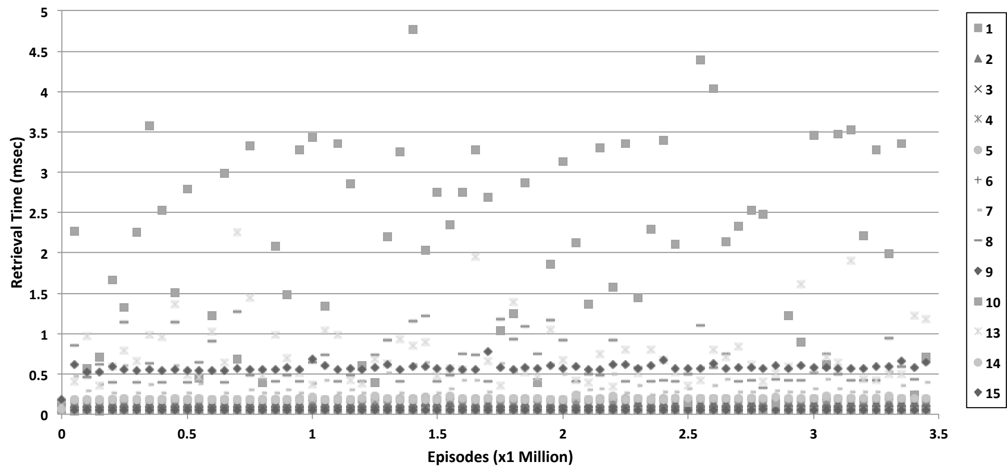

Figure D.2: TankSoar: timing data for all cues
(sans map squares, #11-12; y-axis reduced to 5 msec.).

D.2 Eaters

1. ```
   (<cue> ^superstate nil)
   ```

2. ```
   (<cue> ^directions <d1> <d2>)
   (<d1> ^opposite north ^value south)
   (<d2> ^opposite south ^value north)
   ```

3. ```
   (<cue> ^last-direction south)
   ```

4. ```
   (<cue> ^io.input-link <il>)
   (<il> ^my-location.south.east.content normalfood)
   ```

5. ```
   (<cue> ^io <io> ^last-direction south)
   (<io> ^input-link <il>
         ^output-link <ol>)
   (<ol> ^move.direction south)
   (<il> ^my-location <m>)
   (<m> ^east.content wall ^west.content wall)
   ```

6. ```
   (<cue> ^io.input-link.my-location <m>)
   (<m> ^east.content normalfood ^west.content bonusfood)
   ```

7. ```
   (<cue> ^io.input-link.eater <e>)
   (<e> ^x 14 ^y 1)
   ```
   ```

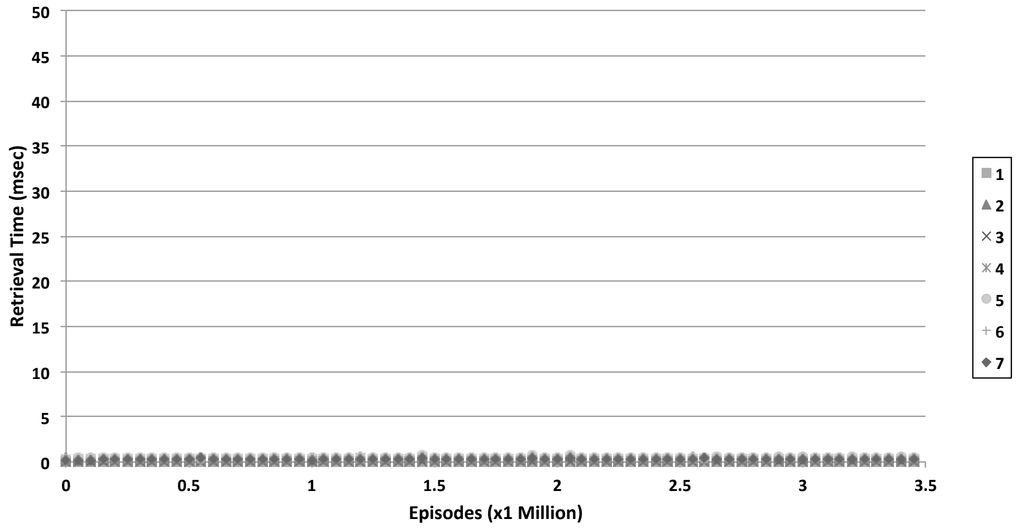

Figure D.3: Eaters: timing data for all cues.

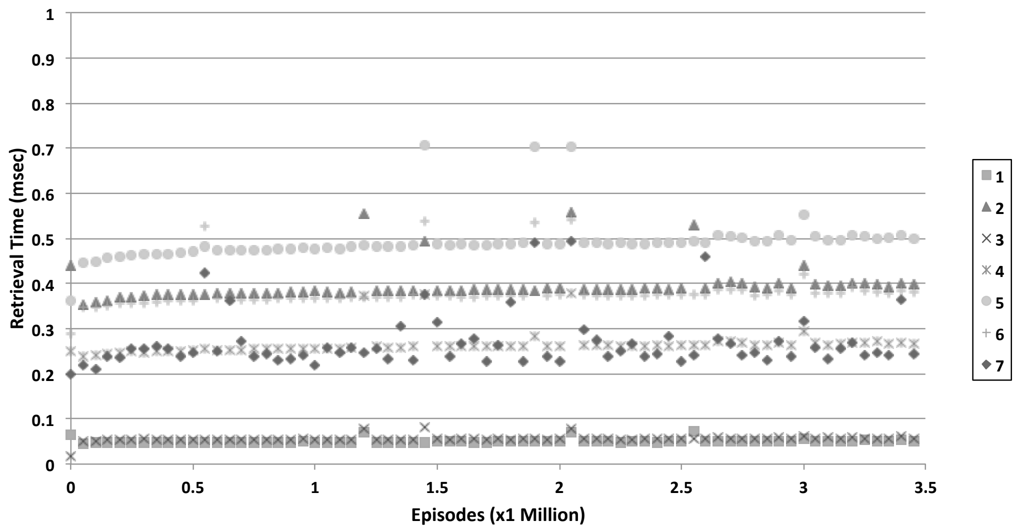

Figure D.4: Eaters: timing data for all cues (y-axis reduced to 1 msec.).

D.3 Infinite Mario

1. (<cue> ^num_monsters 2)

2. (<cue> ^num_monsters 3)

3. (<cue> ^monster <m>)

4. (<cue> ^monster <m>)

   (<m> ^type |Green Koopa| ^winged no ^isthreat yes)

5. (<cue> ^platform <p>)

6. (<cue> ^coin.disty 0)

7. (<cue> ^question <q>)

   (<q> ^isreachable yes ^distx 2)

8. (<cue> ^io.input-link.block-objects <bo>)

   (<bo> ^block.type b ^block.type |?|)

9. (<cue> ^io.input-link.mario.type |Small|)

10. (<cue> ^io.input-link.mario.type |Big|)

11. (<cue> ^io.input-link.mario.type |Fiery|)

12. (<cue> ^io.input-link.monsters.monster <m>)

    (<m> ^type |Goomba| ^winged yes ^vert-direction negative ^isthreat yes)

13. (<cue> ^io.input-link.visual-scene.tile-row.tile.type |$|)

14. (<cue> ^io.output-link <ol> ^question <q>)

    (<ol> ^jump <j> ^move.direction right ^speed.degree high)

    (<q> ^isreachable yes)

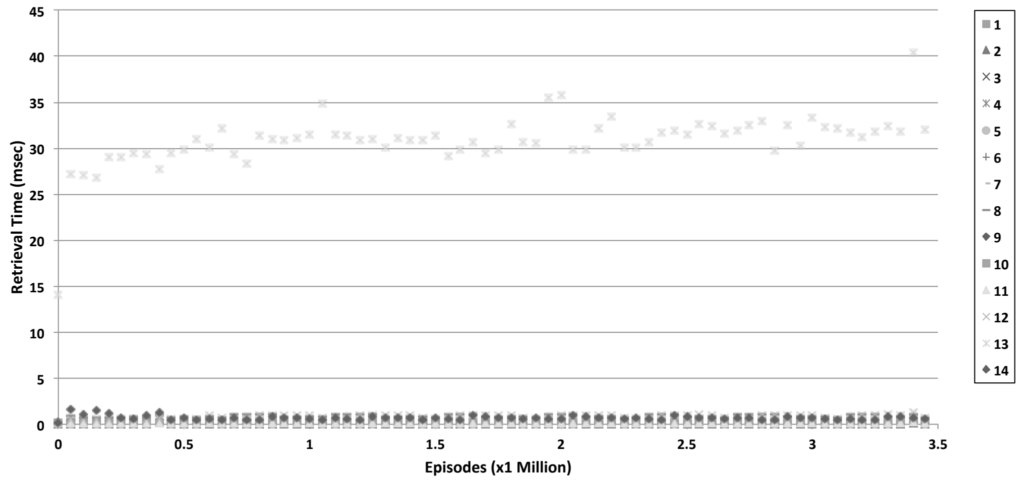

Figure D.5: Infinite Mario: timing data for all cues.

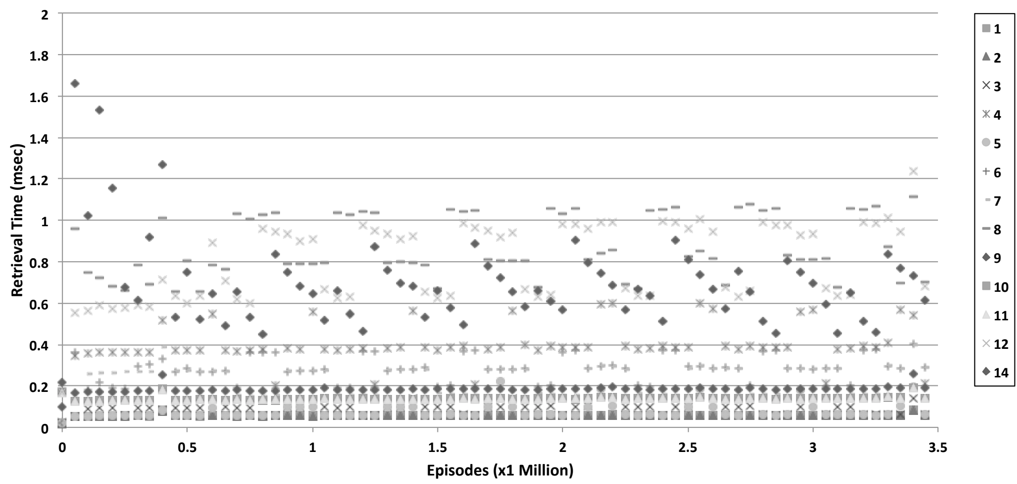

Figure D.6: Infinite Mario: timing data for all cues
(sans visual scene, #13; y-axis reduced to 2 msec.).

D.4 Mobile Robotics

1. ```
   (<cue> ^superstate nil)
   ```

2. ```
   (<cue> ^io.input-link.area-description <ad>)
   (<ad> ^light true ^type room)
   ```

3. ```
   (<cue> ^io.input-link.area-description <ad>)
   (<ad> ^light true ^type room ^wall <w>)
   ```

4. ```
   (<cue> ^io.input-link.self <s>)
   ```

5. ```
   (<cue> ^io <io>)
   (<io> ^input-link.area-description.id 10
         ^output-link.set-velocity.angular-velocity 0)
   ```

6. ```
   (<cue> ^description-area <da>)
   (<da> ^id 5 ^type door)
   ```

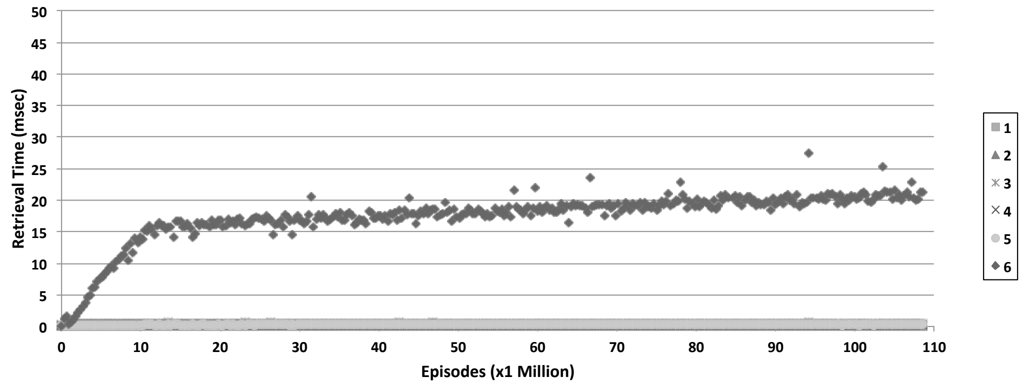

Figure D.7: Mobile Robotics: timing data for all cues.

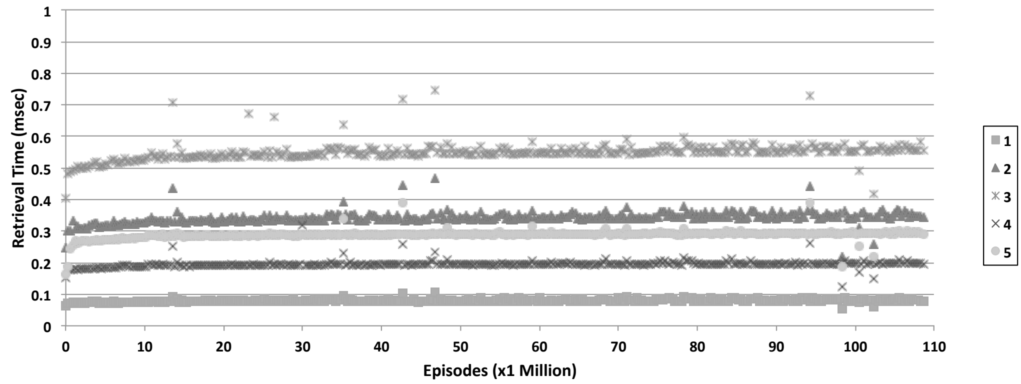

Figure D.8: Mobile Robotics: timing data for all cues
(sans goal management, #6; y-axis reduced to 1 msec.).

APPENDIX E

Semantic Memory: Negative Cues

We define a *negative* retrieval cue as a set of symbols corresponding to the set of augmentations that an element must *not* contain. In the same fashion that we mapped the symbolic retrieval cue to the positive tests in ACT-R (see Section 5.3.2), we can map the negative cue to negative tests.

We have struggled with how to efficiently support negative cues for large semantic stores given the approach outlined in Chapter V. The difficulty arises in large semantic-memory stores where augmentations are *selective*: that is, the set of elements that do *not* contain an augmentations tends to be relatively large (i.e. $|R_c|$ is small). As a result, changes to elements (i.e. augmentation addition/removal or activation event) require updates to supplementary-index element lists that represent a large portion of the memory, and thus storage scales linearly with the number of elements, $O(|E|)$.

E.1 Initial Implementation

Our approach to negative cues draws inspiration from production-matching algorithms (*Doorenbos*, 1995), which suffer from an analogous issue regarding negated

conditions. At a high level, we make a closed-world assumption with regard to memory elements, and thus invert positive operations over existing indices to support negative cues. For this analysis, we define R'_c as the set of elements that do not contain an augmentation c.

First, we add one element list to the supplementary index: a global list of elements, sorted by bias in descending order. For locally efficient biases (i.e. a constant number of activations change per time step), maintaining this list is efficient.

Next, during the first phase of query evaluation (query plan; see Section 5.4.3.2), we derive R'_c for negative augmentation c as $(|E| - |R_c|)$: that is, the number of elements that do not contain the augmentation is equal to the total number of elements minus the number of elements that *do* contain that augmentation. If no element contains the negative-cue augmentation (i.e. all elements do not contain that augmentation), then $|R'_c| = |E| - |\{\}| = |E| - 0 = |E|$. This process adds only one additional subtraction calculation, and thus maintains the same efficiency as the positive cue.

If the most constraining augmentation was from a positive-cue augmentation, the remainder of query evaluation is straightforward, and we need only invert the result of candidate verification for negative-cue augmentations. However, if the most constraining augmentation is from a negative-cue augmentation, we have two options to support the remainder of query evaluation. The first option is to use the global element list as the candidate list w and require additional verification of the first negative-cue augmentation. This option amounts to a list merge, and scales linearly with the number of elements, $O(|E|)$; thus a single-augmentation cue would have retrieval time that was dependent upon augmentation selectivity, a complexity property not true of positive cues (see Figure 5.2).

The second option is to initialize w as the head of the list of the first *positive*-cue augmentation in Q. Using this option, single-augmentation cues retain selectivity

independence, but multi-augmentation cues may require longer search if there is a significant difference in the constraint imposed by the most constraining positive-cue augmentation, d, as compared to the negative-cue augmentation, c: $|R_d| \gg |R'_c|$. We have not implemented this approach, nor have we studied agents that use negative cues, and thus we cannot make claims as to the relative frequency of this situation.

APPENDIX F

Semantic Memory: Relational Schemas

The lists below detail the relational tables for semantic memory (Chapter V). A "type" of "any" denotes no field type constraint.

F.1 Persistent Variables (*vars*)

Stores persistent variables as key/value pairs.

Field	Type	Notes
id	integer	variable key
value	any	variable value

F.2 Symbol Hash: Type (*symbols_type*)

Stores hashes from (type) to a single integer. Values are separated into specialized relations below.

Field	Type	Notes
id	integer	symbol hash
sym_type	integer	symbol type

F.3 Symbol Hash: String (*symbols_str*)

Stores hashes from (string) to a single integer.

Field	Type	Notes
id	integer	symbol hash
sym_const	string	symbol value

F.4 Symbol Hash: Integer (*symbols_int*)

Stores hashes from (integer) to a single integer.

Field	Type	Notes
id	integer	symbol hash
sym_const	integer	symbol value

F.5 Symbol Hash: Float (*symbols_float*)

Stores hashes from (float) to a single integer.

Field	Type	Notes
id	integer	symbol hash
sym_const	float	symbol value

F.6 Element Registration (*lti*)

Stores long-term identifier information.

Field	Type	Notes
id	integer	unique id
letter	integer	ASCII value of identifier letter
num	integer	identifier number
child_ct	integer	augmentation cardinality
act_value	float	bias value
access_n	integer	number of activations
access_t	integer	time step of last activation
access_1	integer	time step of first activation

F.7 Element Activation History (*history*)

Stores a bounded long-term identifier activation history (k=10 in Soar).

Field	Type	Notes
id	integer	unique id (refers to *id* in lti)
t1	integer	time step of most recent activation
t2	integer	time step of 2nd most recent activation
		...
tk	integer	time step of kth most recent activation

F.8 Inverted Table (*web*)

Stores augmentation information.

Field	Type	Notes
parent_id	integer	unique id (refers to *id* in lti)
attrib	integer	augmentation attribute (refers to *id* in symbols_type)
val_const	integer	augmentation value, if constant (refers to *id* in symbols_type)
val_lti	integer	augmentation value, if element (refers to *id* in lti)
act_value	float	bias value

F.9 Statistics: Attribute (*ct_attr*)

Stores frequency of augmentation attributes.

Field	Type	Notes
attr	integer	augmentation attribute (refers to *id* in symbols_type)
ct	integer	number of occurrences

F.10 Statistics: Attribute-Constant (*ct_const*)

Stores frequency of augmentation attribute-value pairs, where value is a constant.

Field	Type	Notes
attr	integer	augmentation attribute (refers to *id* in symbols_type)
val_const	integer	augmentation value (refers to *id* in symbols_type)
ct	integer	number of occurrences

F.11 Statistics: Attribute-LTI (*ct_lti*)

Stores frequency of augmentation attribute-value pairs, where value is an LTI.

Field	Type	Notes
attr	integer	augmentation attribute (refers to *id* in symbols_type)
val_lti	integer	augmentation value (refers to *id* in lti)
ct	integer	number of occurrences

APPENDIX G

Semantic Memory: Algorithms

The sections below detail the algorithms for semantic memory (Chapter V).

G.1 Assumptions

I. Semantic memory is represented as a directed graph, where each node has an associated real-valued *bias*.

II. Semantic memory is a set: no two edges can have the same triple (parent, label, value).

III. A retrieval cue is a set of edges (label, value).

IV. Semantic retrieval returns a single object (node and outgoing edges) that contains the retrieval cue and has the highest bias value of all nodes that contain the cue.

G.2 Storage

```
map objects, biases, inverted
real cardinalitythresh ← τ

function BiasViaThresh(real bias, int cardinality)
  if (cardinality < cardinalitythresh): return bias
  else: return ∞
end function

function AddEdge(node objectid, edge newe)
  oldcardinality ← objects[objectId].size()
  bias ← biases[objectId]

  if (oldcardinality is (cardinalitythresh - 1))
    for e in objects[objectId]
      RemoveFromSortedList(inverted[e], objectId, bias)
      InsertIntoSortedList(inverted[e], objectId, ∞)
    end for
  end if

  objects[objectId].insert(newe)
  InsertIntoSortedList(inverted[newe], objectId,
                       BiasViaThresh(bias, (oldcardinality + 1)))
end function
```

```
function RemoveEdge(node objectid, edge olde)
  oldcardinality ← objects[objectId].size()
  bias ← biases[objectId]

  objects[objectId].remove(olde)
  RemoveFromSortedList(inverted[olde], objectId,
                       BiasViaThresh(bias, oldcardinality))

  if (oldcardinality is cardinalitythresh)
    for e in objects[objectId]
      RemoveFromSortedList(inverted[e], objectId, ∞)
      InsertIntoSortedList(inverted[e], objectId, bias)
    end for
  end if
end function

function UpdateBias(node objectid, real newbias)
  oldbias ← biases[objectId]
  biases[objectId] ← newbias

  if (objects[objectId].size() < cardinalitythresh)
    for e in objects[objectId]
      RemoveFromSortedList(inverted[e], objectId, oldbias)
      InsertIntoSortedList(inverted[e], objectId, newbias)
    end for
  end if
end function
```

```
function StoreObject(node objectid, set edges, real newbias)
  if (not objects.contains(objectid))
    objects[objectId] ← {}
    biases[objectId] ← 0
  end if

  for e in objects[objectId]
    RemoveEdge(objectId, e)
  end for

  UpdateBias(objectId, newbias)

  for e in edges
    AddEdge(objectId, e)
  end for
end function
```

G.2.1 Notes

Storage and bias-value updates are done incrementally. When an object's edge cardinality changes with respect to the threshold, we shift between sort-on-query and static-sort for that object.

G.3 Cue Matching

```
function CueMatch(set cue)
  list queryplan, highcardinality
  edge mostconstraining
  node ret ← null

  for e in cue:
    InsertIntoSortedList(queryplan, e, inverted[e].size())

  mostconstraining ← queryplan.head
  queryplan.removeHead()

  for cand in inverted[mostconstraining]
    if (objects[cand].size() < cardinalitythresh): break
    else: InsertIntoSortedList(highcardinality, cand, biases[cand])
  end for

  nextCand ← IncrementalMerge(inverted[mostconstraining], highcardinality)
    bool goodcand ← true
    for e in queryplan
      if (not objects[nextCand].contains(e)): goodcand ← false
    end for
    if (goodcand is true): return goodcand
  ContinueMerge

  failure
end function
```

APPENDIX H

Semantic Memory: Augmentation Cardinality CDFs for SUMO, OpenCyc, and WordNet

The figures in this appendix plot the cumulative proportion of elements in common knowledge bases versus augmentation cardinality (i.e. number of features per element). We bounded the x-axis of each chart at a value of 50; however, maximum values were much larger (SUMO=10,302; OpenCyc=279; and WordNet=679). This data supports the assumption of small element cardinality.

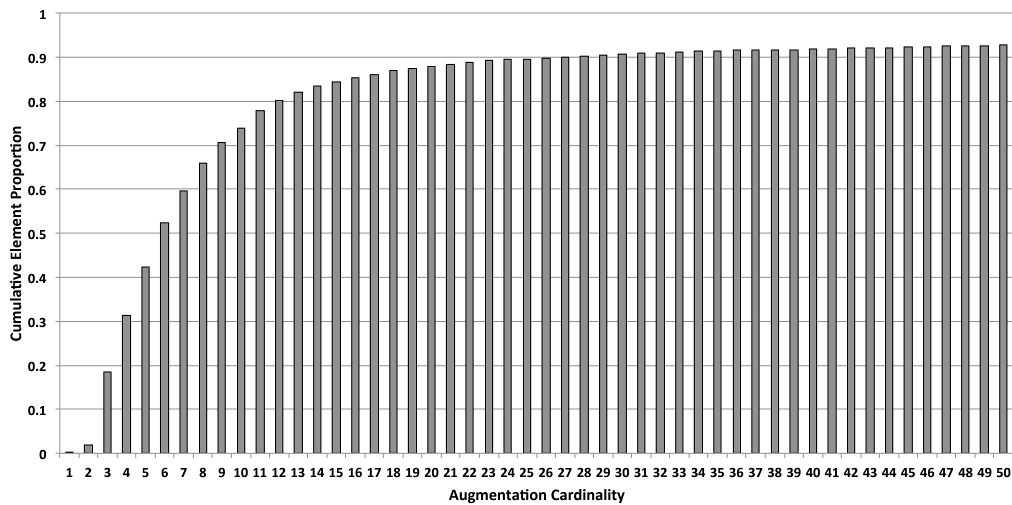

Figure H.1: CDF of augmentation cardinality in SUMO (*Niles and Pease*, 2001).

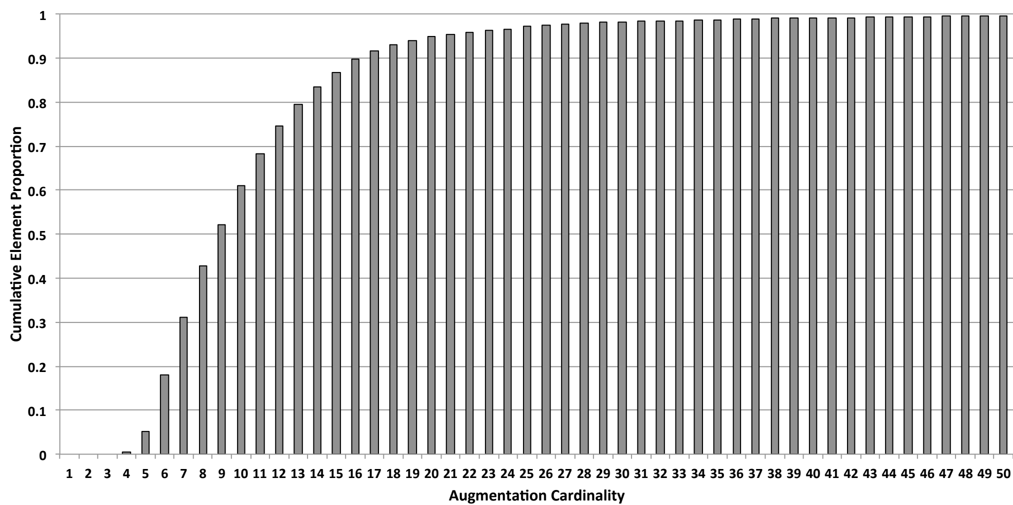

Figure H.2: CDF of augmentation cardinality in OpenCyc, a subset of Cyc (*Lenat*, 1995).

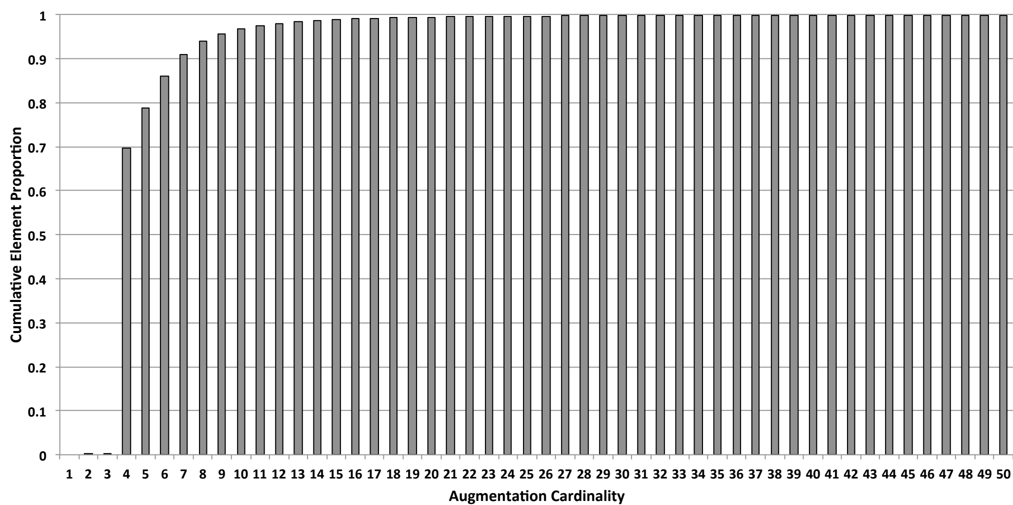

Figure H.3: CDF of augmentation cardinality in WordNet (*Miller*, 1995).

APPENDIX I

Forgetting: Algorithms

The sections below detail the forgetting algorithms (Chapter VI).

I.1 Assumptions

I. Memory comprises a set of symbolic elements.

II. Time increments monotonically in discrete time steps.

III. Each element has a constant-sized (c) history of prior access {(time of access, number of accesses at that time)}.

IV. Base-level activation has a constant decay parameter, d, and a constant forgetting threshold, Θ.

I.2 Element Activation

```
map histories
map decay
int maxwindow ← c
real minactivation ← Θ
real decay ← d

decay[0] ← {}

function DecayApproximation(access a, time t)
  ret ← ceiling( exp( [minactivation - ln(a.second)] / -decay ) )
  timediff ← (t - a.first)
  if (timediff > ret): return (timediff - ret)
  else: return 0
end function

function PredictDecay(element m, time t, bool newaccess)
  time td ← t

  if (newaccess is true):
    for a in histories[m].window:
      td ← (td + DecayApproximation(a, t))

  if (td is t): return BinaryParameterSearch(histories[m], t)
  else: return td
end function
```

```
function ActivateMemory(element m, time t, int n)
  if (not histories.containsKey(m))
    histories[m] ← {window={}, firstt=t, total=0, decaykey=0}
    decay[0].insert(m)
  end if

  histories[m].window.append(<t, n>)
  if (histories[m].window.size() > maxwindow):
    histories[m].window.remove(histories[m].window.head)

  histories[m].total ← (histories[m].total + n)

  decay[histories[m].decaykey].remove(m)
  histories[m].decaykey = PredictDecay(m, t, true)
  decay[histories[m].decaykey].insert(m)
end function
```

I.2.1 Notes

The binary parameter search seeks the time at which base-level activation (via a tail approximation (*Petrov*, 2006), which incorporates total number of accesses, *total*, and time of first access, *firstt*) falls below threshold (*minactivation*).

I.3 Element Forgetting

```
function GetDecayedElements(time t)
  if (not decay.containsKey(t)): return {}

  for m in decay[t]
    if (PetrovBaseLevelActivation(histories[m], t) ≥ minactivation)
      decay[t].remove(m)
      histories[m].decaykey = PredictDecay(m, t, false)
      decay[histories[m].decaykey].insert(m)
    end if
  end for

  return decay[t]
end function
```

I.3.1 Notes

The tail approximation (*Petrov*, 2006) incorporates total number of accesses (*total*) and time of first access (*firstt*).

BIBLIOGRAPHY

BIBLIOGRAPHY

Agrawal, R., and R. Srikant (1994), Fast algorithms for mining association rules, in *Proceedings of the 20th International Conference on Very Large Databases*, pp. 487–499, Santiago, Chile.

Agrawal, S., S. Chaudhuri, and V. Narasayya (2000), Automated selection of materialized views and indexes in SQL databases, in *Proceedings of the 26th International Conference on Very Large Databases*, pp. 496–505, Cairo, Egypt.

Altmann, E. M., and W. D. Gray (1998), Pervasive episodic memory: Evidence from a control-of-attention paradigm, in *Proceedings of the 20th Annual Conference of the Cognitive Science Society*, pp. 42–47.

Altmann, E. M., and W. D. Gray (2002), Forgetting to remember: The functional relationship of decay and interference, *Psychological Science*, *13*(1), 27–33.

Anderson, J. R. (1983), *The Architecture of Cognition*, Harvard University Press, Cambridge, MA, USA.

Anderson, J. R. (1991), The place of cognitive architectures in a rational analysis, in *Architectures for Intelligence: The 22nd Carnegie Mellon Symposium on Cognition*, edited by K. VanLehn, pp. 1–24, Erlbaum.

Anderson, J. R., and L. M. Reder (1999), The fan effect: New results and new theories, *Journal of Experimental Psychology: General*, *128*, 186–197.

Anderson, J. R., and B. H. Ross (1980), Evidence against a semantic-episodic distinction, *Journal of Experimental Psychology: Human Learning and Memory*, *6*(5), 441–466.

Anderson, J. R., and L. J. Schooler (1991), Reflections of the environment in memory, *Psychological Science*, *2*(6), 396–408.

Anderson, J. R., L. M. Reder, and C. Lebiere (1996), Working memory: Activation limits on retrieval, *Cognitive Psychology*, *30*, 221–256.

Anderson, J. R., D. Bothell, C. Lebiere, and M. Matessa (1998), An integrated theory of list memory, *Journal of Memory and Language*, *38*, 341–380.

Anderson, J. R., D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin (2004), An integrated theory of the mind, *Psychological Review*, *111*(4), 1036–1060.

Ball, J., M. D. Freiman, S. M. Rodgers, and C. W. Myers (2010), Toward a functional model of human language processing, in *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, pp. 1583–1588, Portland, OR, USA.

Banerjee, S., and T. Pedersen (2002), An adapted lesk algorithm for word sense disambiguation using wordnet, in *Computational Linguistics and Intelligent Text Processing*, *Lecture Notes in Computer Science*, vol. 2276, edited by A. Gelbukh, pp. 117–171, Springer.

Brom, C., O. Burkert, and R. Kadlec (2010), Timing in episodic memory for virtual characters, in *Proceedings of the IEEE 2010 Conference on Computational Intelligence and Games*, pp. 305–312, Copenhagen, Denmark.

Buro, M. (2003), Real-time strategy games: A new AI research challenge, in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, edited by G. Gottlob and T. Walsh, pp. 1534–1535, Acapulco, Mexico.

Chaudhuri, S. (1998), An overview of query optimization in relational systems, in *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 34–43, Seattle, WA, USA.

Choi, D., T. Konik, N. Nejati, C. Park, and P. Langley (2007), A believable agent for first-person shooter games, in *Papers from the Third Annual Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 71–73, Palo Alto, CA, USA.

Chong, R. (2003), The addition of an activation and decay mechanism to the Soar architecture, in *Proceedings of the Fifth International Conference on Cognitive Modeling*, pp. 45–50, Bamberg, Germany.

Chong, R. (2004), Architectural explorations for modeling procedural skill decay, in *Proceedings of the Sixth International Conference on Cognitive Modeling*, Pittsburgh, PA, USA.

Codd, E. F. (1970), A relational model of data for large shared data banks, *Communications of the ACM*, *13*(6), 377–387.

Combi, C., and Y. Shahar (1997), Temporal reasoning and temporal data maintenance in medicine: Issues and challenges, *Computers in Biology and Medicine*, *27*(5), 353–368.

Crawford, J. M., and B. J. Kuipers (1991), Algernon – a tractable system for knowledge representation, *SIGART Bulletin*, *2*(3), 35–44.

Cummins, L., and D. Bridge (2009), Maintenance by a committee of experts: the MACE approach to case-base maintenance, in *Case-Based Reasoning Research and Development*, *Lecture Notes in Computer Science*, vol. 5650, edited by L. McGinty and D. Wilson, pp. 120–134, Springer.

Curtis, J., J. Cabral, and D. Baxter (2006), On the application of the Cyc ontology to word sense disambiguation, in *Proceedings of the 19th International Florida Artificial Intelligence Research Society Conference*, pp. 652–657, Melbourne Beach, FL, USA.

Daengdej, J., D. Lukose, E. Tsui, P. Beinat, and L. Prophet (1996), Dynamically creating indices for two million cases: A real world problem, in *Advances in Case-Based Reasoning*, *Lecture Notes in Computer Science*, vol. 1168, edited by I. Smith and B. Faltings, pp. 105–119, Springer.

Davis, R., H. Shrobe, and P. Szolovits (1993), What is a knowledge representation?, *AI Magazine*, *14*(1), 17–33.

Dejong, G., and R. Mooney (1986), Explanation-based learning: An alternative view, *Machine Learning*, *1*, 145–176.

Derbinsky, N., and G. Essl (2011), Cognitive architecture in mobile music interactions, in *Proceedings of the 11th International Conference on New Interfaces for Musical Expression*, pp. 104–107, Oslo, Norway.

Derbinsky, N., and G. Essl (2012), Exploring reinforcement learning for mobile percussive collaboration, in *Proceedings of the 12th International Conference on New Interfaces for Musical Expression*, Ann Arbor, MI, USA.

Derbinsky, N., and N. A. Gorski (2010), Exploring the space of computational memory models, in *Proceedings of the 1st Symposium on Human Memory for Artificial Agents*, pp. 38–41, Leicester, UK.

Derbinsky, N., and J. E. Laird (2009), Efficiently implementing episodic memory, in *Proceedings of the 8th International Conference on Case-Based Reasoning*, pp. 403–417, Seattle, WA, USA.

Derbinsky, N., and J. E. Laird (2011), A functional analysis of historical memory retrieval bias in the word sense disambiguation task, in *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, pp. 663–668, San Francisco, CA, USA.

Derbinsky, N., and J. E. Laird (2012a), Competence-preserving retention of learned knowledge in Soar's working and procedural memories, in *Proceedings of the 11th International Conference on Cognitive Modeling*, pp. 205–210, Berlin, Germany.

Derbinsky, N., and J. E. Laird (2012b), Computationally efficient forgetting via base-level activation, in *Proceedings of the 11th International Conference on Cognitive Modeling*, pp. 109–110, Berlin, Germany.

Derbinsky, N., J. E. Laird, and B. Smith (2010), Towards efficiently supporting large symbolic declarative memories, in *Proceedings of the 10th International Conference on Cognitive Modeling*, pp. 49–54, Philadelphia, PA, USA.

Derbinsky, N., J. Li, and J. E. Laird (2012a), A multi-domain evaluation of scaling in a general episodic memory, in *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, Toronto, Canada.

Derbinsky, N., J. Li, and J. E. Laird (2012b), Algorithms for scaling in a general episodic memory, in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, Valencia, Spain.

Deutsch, T., A. Gruber, R. Lang, and R. Velik (2008), Episodic memory for autonomous agents, in *Proceedings of the Fifth International Conference on Human Systems Interaction*, pp. 621–626, Krakow, Poland.

Doorenbos, R. B. (1995), Production matching for large learning systems, Ph.D. thesis, Carnegie Mellon University.

Douglass, S. A., and C. W. Myers (2010), Concurrent knowledge activation calculation in large declarative memories, in *Proceedings of the Tenth International Conference of Cognitive Modeling*, pp. 55–60, Philadelphia, PA, USA.

Douglass, S. A., J. Ball, and S. Rodgers (2009), Large declarative memories in ACT-R, in *Proceedings of the Ninth International Conference of Cognitive Modeling*, pp. 222–227, Manchester, UK.

Edmonds, P., and A. Kilgarriff (2002), Introduction to the special issue on evaluating word sense disambiguation systems, *Natural Language Engineering*, *8*(4), 279–291.

Emond, B. (2006), WN-LEXICAL: An ACT-R module built from the WordNet lexical database, in *Proceedings of the Seventh International Conference on Cognitive Modeling*, pp. 359–360, Trieste, Italy.

Fahlman, S. E. (2006), Marker-passing inference in the Scone knowledge-base system, in *Knowledge Science, Engineering and Management*, *Lecture Notes in Computer Science*, vol. 4092, edited by J. Lang, F. Lin, and J. Wang, pp. 114–126, Springer.

Forbus, K. D., and T. R. Hinrichs (2006), Companion cognitive systems: A step towards human-level AI, *AI Magazine*, *27*(2), 83–95.

Forbus, K. D., D. Gentner, and K. Law (1995), MAC/FAC: A model of similarity-based retrieval, *Cognitive Science*, *19*(2), 141–205.

Forbus, K. D., M. Klenk, and T. R. Hinrichs (2009), Companion cognitive systems: Design goals and lessons learned so far, *IEEE Intelligent Systems*, *24*(4), 36–46.

Forgy, C. L. (1982), Rete: A fast algorithm for the many pattern/many object pattern match problem, *Artificial Intelligence*, *19*(1), 17–37.

Fox, S., and D. B. Leake (1995), Using introspective reasoning to refine indexing, in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, vol. 1, pp. 391–397, Montreal, Quebec, Canada.

Franklin, S., and F. G. Patterson, Jr. (2006), The LIDA architecture: Adding new modes of learning to an intelligent, autonomous, software agent, in *Proceedings of the Ninth World Conference on Integrated Design & Process Technology*, San Diego, CA, USA.

Gale, W. A., K. W. Church, and D. Yarowsky (1992), One sense per discourse, in *Proceedings of the Workshop on Speech and Natural Language*, pp. 233–237, Stroudsburg, PA, USA.

Gentner, D., J. Loewenstein, L. Thompson, and K. D. Forbus (2009), Reviving inert knowledge: Analogical abstraction supports relational retrieval of past events, *Cognitive Science*, *33*(8), 1343–1382.

Gomes, P. F., C. Martinho, and A. Paiva (2011), I've been here before! location and appraisal in memory retrieval, in *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems*, pp. 1039–1046, Taipei, Taiwan.

Gorski, N. A., and J. E. Laird (2011), Learning to use episodic memory, *Cognitive Systems Research*, *12*(2), 144–153.

Gray, J., S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh (1997), Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals, *Data Mining and Knowledge Discovery*, *1*(1), 29–53.

Hipp, D. R. (2012), SQLite, `http://www.sqlite.org`.

Jære, M. D., A. Aamodt, and P. Skalle (2002), Representing temporal knowledge for case-based prediction, in *Advances in Case-Based Reasoning*, *Lecture Notes in Computer Science*, vol. 2416, edited by S. Craw and A. Preece, pp. 225–234, Springer.

Jones, R. M., J. E. Laird, P. E. Nielsen, K. J. Coulter, P. Kenny, and F. V. Koss (1999), Automated intelligent pilots for combat flight simulation, *AI Magazine*, *20*(1), 27–41.

Jonides, J., R. L. Lewis, D. E. Nee, C. A. Lustig, M. G. Berman, and K. S. Moore (2008), The mind and brain of short-term memory, *Annual Review of Psychology*, *59*, 193–224.

Kennedy, W. G., and J. G. Trafton (2007), Long-term symbolic learning, *Cognitive Systems Research*, *8*, 237–247.

Kilgarriff, A., and J. Rosenzweig (2000), English Senseval: Report and results, in *Proceedings of the Second International Conference on Language Resources & Evaluation*, Athens, Greece.

Kolodner, J. L. (1993), *Case-Based Reasoning*, Morgan Kaufmann.

Kriegel, H.-P., M. Pöthe, and T. Seidl (2000), Managing intervals efficiently in object-relational databases, in *Proceedings of the 26th International Conference on Very Large Databases*, pp. 407–418, Cairo, Egypt.

Kucera, H., and W. N. Francis (1967), *Computational Analysis of Present-Day American English*, Brown University Press, Providence, RI, USA.

Kuppuswamy, N. S., S.-H. Cho, and J.-H. Kim (2006), A cognitive control architecture for an artificial creature using episodic memory, in *Proceedings of the Third SICE-ICASE International Joint Conference*, pp. 3104–3110, Busan, South Korea.

Laird, J. E. (2001), It knows what you're going to do: Adding anticipation to a Quakebot, in *Proceedings of the Fifth International Conference on Autonomous Agents*, pp. 385–392, Montreal, Quebec, Canada.

Laird, J. E. (2008), Extending the Soar cognitive architecture, in *Proceedings of the First Conference on Artificial General Intelligence*, pp. 1–6224–235, Memphis, TN, USA.

Laird, J. E. (2012), *The Soar Cognitive Architecture*, MIT Press, Cambridge.

Laird, J. E., and P. S. Rosenbloom (1996), The evolution of the Soar cognitive architecture, in *Mind Matters: A Tribute to Allen Newell*, edited by D. M. Steier and T. M. Mitchell, pp. 1–48, Lawrence Erlbaum Associates.

Laird, J. E., and R. E. Wray, III (2010), Cognitive architecture requirements for achieving AGI, in *Proceedings of the Third Conference on Artificial General Intelligence*, pp. 1–6, Lugano, Switzerland.

Laird, J. E., P. S. Rosenbloom, and A. Newell (1986), Chunking in Soar: The anatomy of a general learning mechanism, *Machine Learning*, *1*(1), 11–46.

Laird, J. E., A. Newell, and P. S. Rosenbloom (1987), SOAR: An architecture for general intelligence, *Artificial Intelligence*, *33*(1), 1–64.

Laird, J. E., J. Z. Xu, and S. Wintermute (2010), Using diverse cognitive mechanisms for action modeling, in *Proceedings of the 10th International Conference on Cognitive Modeling*, pp. 133–138, Philadelphia, PA, USA.

Laird, J. E., N. Derbinsky, and M. Tinkerhess (2011a), A case study in integrating probabilistic decision making and learning in a symbolic cognitive architecture: Soar plays dice, in *Papers from the 2011 AAAI Fall Symposium Series: Advances in Cognitive Systems*, pp. 162–169, Arlington, VA, USA.

Laird, J. E., N. Derbinsky, and J. Voigt (2011b), Performance evaluation of declarative memory systems in Soar, in *Proceedings of the 20th Behavior Representation in Modeling and Simulation Conference*, pp. 33–40, Sundance, UT, USA.

Langley, P., and D. Choi (2006), A unified cognitive architecture for physical agents, in *Proceedings of the 21st National Conference on Artificial Intelligence*, pp. 1469–1474, Boston, MA, USA.

Langley, P., K. Cummings, and D. Shapiro (2004), Hierarchical skills and cognitive architectures, in *Proceedings of the 26th Annual Conference of the Cognitive Science Society*, pp. 779–784, Chicago, IL, USA.

Langley, P., J. E. Laird, and S. Rogers (2009), Cognitive architectures: Research issues and challenges, *Cognitive Systems Research*, *10*(1), 141–160.

Lenat, D. B. (1995), CYC: A large-scale investment in knowledge infrastructure, *Communications of the ACM*, *38*(11), 33–38.

Lenat, D. B., M. Witbrock, D. Baxter, E. Blackstone, C. Deaton, D. Schneider, J. Scott, and B. Shepard (2010), Harnessing Cyc to answer clinical researchers' *ad hoc* queries, *AI Magazine*, *31*(3), 13–32.

Lenz, M., and H.-D. Burkhard (1996), Case retrieval nets: Basic ideas and extensions, in *KI-96: Advances in Artificial Intelligence*, *Lecture Notes in Computer Science*, vol. 1137, edited by G. Görz and S. Hölldobler, pp. 227–239, Springer.

Lenzerini, M. (2002), Data integration: A theoretical perspective, in *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 233–246, Madison, WI, USA.

Lesk, M. (1986), Automatic sense disambiguation using machine readable dictionaries: How to tell a pine cone from an ice cream cone, in *Proceedings of the Fifth Annual International Conference on Systems Documentation*, pp. 24–26, Toronto, Canada.

Li, J., and J. E. Laird (2011), Preliminary evaluation of long-term memories for fulfilling delayed intentions, in *Papers from the 2011 AAAI Fall Symposium Series: Advances in Cognitive Systems*, pp. 170–177, Arlington, VA, USA.

Li, J., N. Derbinsky, and J. E. Laird (2012), Functional interactions between memory and recognition judgments, in *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, Toronto, Canada.

Lim, M. Y., R. Aylett, P. A. Vargas, W. C. Ho, and J. Dias (2011), Human-like memory retrieval mechanisms for social companions (extended abstract), in *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems*, pp. 1117–1118, Taipei, Taiwan.

Ma, J., and B. Knight (2003), A framework for historical case-based reasoning, in *Case-Based Reasoning Research and Development*, *Lecture Notes in Computer Science*, vol. 2689, edited by K. Ashley and D. Bridge, pp. 246–260, Springer.

Macedo, L., and A. Cardoso (2004), Exploration of unknown environments with motivational agents, in *Proceedings of the Third International Conference on Autonomous Agents and Multiagent Systems*, pp. 328–335, New York, NY, USA.

Madl, T., B. J. Baars, and S. Franklin (2011), The timing of the cognitive cycle, *PLoS ONE*, *6*(4), e14,803.

Marinier, R. P., III, J. E. Laird, and R. L. Lewis (2009), A computational unification of cognitive behavior and emotion, *Cognitive Systems Research*, *10*(1), 48–69.

Markovitch, S., and P. D. Scott (1988), The role of forgetting in learning, in *Proceedings of the Fifth International Conference on Machine Learning*, pp. 459–465, Ann Arbor, MI, USA.

Meyer, D. E., and D. E. Kieras (1997), A computational theory of executive control processes and human multiple-task performance: Part I. Basic mechanisms, *Psychological Review*, *104*(1), 3–65.

Miller, G. A. (1995), WordNet: A lexical database for english, *Communications of the ACM*, *28*(11), 29–41.

Miller, G. A., C. Leacock, R. Tengi, and R. T. Bunker (1993), A semantic concordance, in *Proceedings of the Workshop on Human Language Technology*, pp. 303–308.

Mohan, S., and J. E. Laird (2011), An object-oriented approach to reinforcement learning in an action game, in *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 164–169, Palo Alto, CA, USA.

Nandi, A., and H. V. Jagadish (2011), Guided interaction: Rethinking the query-result paradigm, *Proceedings of the VLDB Endowment*, *4*(12), 1466–1469.

Nason, S., and J. E. Laird (2004), Soar-RL: Integrating reinforcement learning with Soar, *Cognitive Systems Research*, *6*(1), 51–59.

Navigli, R. (2009), Word sense disambiguation: A survey, *ACM Computing Surveys*, *41*(2), 10:1–10:69.

Newell, A. (1990), *Unified Theories of Cognition*, Harvard University Press, Cambridge.

Newell, A., and H. A. Simon (1972), *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, USA.

Niles, I., and A. Pease (2001), Towards a standard upper ontology, in *Proceedings of the International Conference on Formal Ontology in Information Systems*, pp. 2–9, Ogunquit, ME, USA.

Nuxoll, A. M. (2007), Enhancing intelligent agents with episodic memory, Ph.D. thesis, University of Michigan.

Nuxoll, A. M., and J. E. Laird (2012), Enhancing intelligent agents with episodic memory, *Cognitive Systems Research*, *17-18*, 34–48.

Nuxoll, A. M., J. E. Laird, and M. James (2004), Comprehensive working memory activation in Soar, in *Proceedings of the Sixth International Conference on Cognitive Modeling*, pp. 226–230, Pittsburgh, PA, USA.

O'Reilly, R. C. (1996), The Leabra model of neural interactions and learning in the neocortex, Ph.D. thesis, Carnegie Mellon University.

Paré, D. (2003), Role of the basolateral amygdala in memory consolidation, *Progress in Neurobiology*, *70*, 409–420.

Patterson, D. W., N. Rooney, and M. Galushka (2003), Efficient retrieval for case-based reasoning, in *Proceedings of the 16th International Florida Artificial Intelligence Research Society Conference*, pp. 144–149, St. Augustine, Florida, USA.

Patterson, D. W., M. Galushka, and N. Rooney (2004), An effective indexing and retrieval approach for temporal cases, in *Proceedings of the 17th International Florida Artificial Intelligence Research Society Conference*, Miami Beach, Florida, USA.

Petrov, A. A. (2006), Computationally efficient approximation of the base-level learning equation in ACT-R, in *Proceedings of the Seventh International Conference on Cognitive Modeling*, pp. 391–392, Trieste, Italy.

Porter, M. F. (2006), An algorithm for suffix stripping, *Program: Electronic Library and Information Systems*, *40*(3), 211–218.

Rajagopalan, R. (1995), Qualitative reasoning about dynamic change in the spatial properties of a physical system, Ph.D. thesis, University of Texas at Austin.

Ramakrishnan, R., and J. Gehrke (2003), *Database Management Systems*, third ed., McGraw-Hill Higher Education.

Remolina, E. (2001), A logical account of causal and topological maps, Ph.D. thesis, University of Texas at Austin.

Rickel, J. W. (1995), Automated modeling of complex systems to answer prediction questions, Ph.D. thesis, University of Texas at Austin.

Rieh, S. Y., and H. I. Xie (2006), Analysis of multiple query reformulations on the web: The interactive information retrieval context, *Information Processing and Management*, *42*, 751–768.

Sánchez-Marré, M., U. Cortés, M. Martínez, J. Comas, and I. Rodríguez-Roda (2003), An approach for temporal case-based reasoning: Episode-based reasoning, in *Case-Based Reasoning Research and Development*, *Lecture Notes in Computer Science*, vol. 3620, edited by H. Muñoz-Ávila and F. Ricci, pp. 465–476, Springer.

Sanner, S., J. R. Anderson, C. Lebiere, and M. Lovett (2000), Achieving efficient and cognitively plausible learning in backgammon, in *Proceedings of the 17th International Conference on Machine Learning*, pp. 823–830, Palo Alto, CA, USA.

Schooler, L. J., and J. R. Anderson (1997), The role of process in the rational analysis of memory, *Cognitive Psychology*, *32*, 219–250.

Schooler, L. J., and R. Hertwig (2005), How forgetting aids heuristic inference, *Psychological Review*, *112*(3), 610–628.

Siegel, N., B. Shepard, J. Cabral, and M. Witbrock (2005), Hypothesis generation and evidence assembly for intelligence analysis: Cycorp's Noöscape application, in *Proceedings of the 2005 International Conference on Intelligence Analysis*, pp. 2–6, McLean, VA, USA.

Simon, H. A. (1991), Cognitive architectures and rational analysis: Comment, in *Architectures for Intelligence: The 22nd Carnegie Mellon Symposium on Cognition*, edited by K. VanLehn, pp. 25–39, Erlbaum.

Singhal, A. (2001), Modern information retrieval: A brief overview, *IEEE Data Engineering Bulletin*, *24*(4), 35–43.

Smyth, B., and P. Cunningham (1996), The utility problem analysed - a case-based reasoning perspective, in *Proceedings of the Third European Workshop on Case-Based Reasoning*, pp. 392–399, Lausanne, Switzerland.

Snaider, J., and S. Franklin (2011), Extended sparse distributed memory, in *Biologically Inspired Cognitive Architectures 2011: Proceedings of the Second Annual Meeting of the BICA Society*, *Frontiers in Artificial Intelligence and Applications*, vol. 233, edited by A. V. Samsonovich and K. R. Jóhannsdóttir, pp. 351–357, IOS Press.

Snaider, J., R. McCall, and S. Franklin (2011), The lida framework as a general tool for agi, in *Artificial General Intelligence*, *Lecture Notes in Computer Science*, vol. 6830, edited by J. Schmidhuber, K. R. Thorisson, and M. Looks, pp. 133–142, Springer.

Squire, L. R., and P. Alvarez (1995), Retrograde amnesia and memory consolidation: A neurobiological perspective, *Current Opinion in Neurobiology*, *5*, 169–177.

Stottler, R. H., A. L. Henke, and J. A. King (1989), Rapid retrieval algorithms for case-based reasoning, in *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pp. 233–237, Detroit, MI, USA.

Stracuzzi, D. J., N. Li, G. Cleveland, and P. Langley (2009), Representing and reasoning over time in a unified cognitive architecture, in *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, pp. 2986–2991, Amsterdam, Netherlands.

Strosnider, J. K., and C. J. Paul (1994), A structured view of real-time problem solving, *AI Magazine*, *15*(2), 45–66.

Sun, R. (2006), The CLARION cognitive architecture: Extending cognitive modeling to social simulation, in *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, edited by R. Sun, pp. 79–99, Cambridge University Press, New York.

Taatgen, N. A., D. Huss, and J. R. Anderson (2006), How cognitive models can inform the design of instructions, in *Proceedings of the Seventh International Conference on Cognitive Modeling*, pp. 304–309, Trieste, Italy.

Tambe, M., A. Newell, and P. S. Rosenbloom (1990), The problem of expensive chunks and its solution by restricting expressiveness, *Machine Learning*, *5*, 299–349.

Tanner, B., and A. White (2009), RL-Glue: Language-independent software for reinforcement-learning experiments, *The Journal of Machine Learning Research*, *10*, 2133–2136.

Tecuci, D., and B. Porter (2007a), A generic memory module for events, in *Proceedings of the 20th Florida Artificial Intelligence Research Society Conference*, pp. 152–157, Key West, FL, USA.

Tecuci, D., and B. Porter (2007b), Memory based goal schema recognition, in *Proceedings of the 22nd Florida Artificial Intelligence Research Society Conference*, Sanibel Island, FL, USA.

Terrovitis, M., S. Passas, P. Vassiliadis, and T. Sellis (2006), A combination of trie-trees and inverted files for the indexing of set-valued attributes, in *Proceedings of the 15th International Conference on Information and Knowledge Management*, pp. 728–737, Arlington, VA, USA.

Tribble, A., and C. Rosé (2006), Usable browsers for ontological knowledge acquisition, in *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, pp. 1451–1456, New York, NY, USA.

Trivedi, N., P. Langley, P. Schermerhorn, and M. Scheutz (2011), Communicating, interpreting, and executing high-level instructions for human-robot interaction, in *Papers from the 2011 AAAI Fall Symposium on Advances in Cognitive Systems*, pp. 321–328, Washington, DC, USA.

Tulving, E. (1972), Episodic and semantic memory, in *Organization of Memory*, edited by E. Tulving and W. Donaldson, pp. 381–403, Academic Press, New York.

Tulving, E. (1983), *Elements of Episodic Memory*, Clarendon Press, Oxford.

Vasilescu, F., P. Langlais, and G. Lapalme (2004), Evaluating variants of the Lesk approach for disambiguating words, in *Proceedings of the Conference of Language Resources and Evaluations*, pp. 633–636.

Wess, S., K.-D. Althoff, and G. Derwand (1994), Using k-d trees to improve the retrieval step in case-based reasoning, in *Topics in Case-Based Reasoning*, *Lecture Notes in Computer Science*, vol. 837, edited by S. Wess, K.-D. Althoff, and M. Richter, pp. 167–181, Springer.

Wilson, D. R., and T. R. Martinez (2000), Reduction techniques for instance-based learning algorithms, *Machine Learning*, *38*(3), 257–286.

Wintermute, S. (2010), Using imagery to simplify perceptual abstraction in reinforcement learning agents, in *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pp. 1567–1573, Atlanta, GA, USA.

Wintermute, S., J. Xu, and J. E. Laird (2007), SORTS: A human-level approach to real-time strategy AI, in *Proceedings of the Third AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 55–60, Palo Alto, CA, USA.

Xu, J. Z., and J. E. Laird (2010), Instance-based online learning of deterministic relational action models, in *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pp. 1574–1579, Atlanta, GA, USA.

Xu, J. Z., and J. E. Laird (2011), Combining learned discrete and continuous action models, in *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, pp. 1449–1454, San Francisco, CA, USA.

Yokoo, M., and K. Hirayama (2000), Algorithms for distributed constraint satisfaction: A review, *Autonomous Agents and Multi-Agent Systems*, *3*, 185–207.

Zobel, J., and A. Moffat (2006), Inverted files for text search engines, *ACM Computing Surveys*, *38*(2), 1–56.