Learning General and Correct Procedural Knowledge in a Cognitive Architecture

by

Mazin Assanie

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
2022

Mazin Assanie

mazina@umich.edu

ORCID iD: 0000-0002-1690-8932

This work is dedicated to all the people who supported me when I needed it most, most notably my parents, Mohammed As-Sanie and Fatimah Barazi, my sisters, Suzie, Abir and Rudi, and my advisor and mentor, John Laird.

Acknowledgments

I would like to express my sincere gratitude to all my fellow Soar group members who helped me learn so much along the way: Nathan Derbinsky, Joseph Xu, Bob Marinier, Bob Wray, Alex Turner, Karen Coulter and, most of all, John Laird, whose kindness, sincerity and patience knows no bounds. I'd like to thank Karen Liska, Dawn Freysinger, Kelly Cormier and Cindy Watts for their indefatigable kindness and helpfulness navigating the program, and Bill Birmingham for giving me a chance my first semester by letting me GSI a class that I had no business doing. I'd like to thank Steven Jones for his friendship and being someone I can vent to and bounce ideas off of. I'd like to thank Bryan Stearns for his collaboration and showing me that people still act ethically. Finally, I would like to acknowledge some of my fellow graduate students that made the journey much more enjoyable: Dmitry Kalinchenko, Bradley Clement, Lukasz Opyrchal, Tom Bersano, Peter Lindes, Jule Schatz, Preeti Ramaraj and Elizabeth Mamatov. Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

EBBS	Explanation-Based Behavior Summarization
DIGU	Distributed Identity Graph Unification
OSK	Operator Selection Knowledge
ROSK	Relevant Operator Selection Knowledge
SI	Soar Identifier
LTI	Long-term Identifier
LTM	Long-term Memory
LTME	Long-term Memory Element
STM	Short-term Memory (= Working Memory)
WM	Working Memory
WME	Working Memory Element

ABSTRACT

For cognitive architectures that encode procedural knowledge as rules, online procedural learning algorithms exist that can analyze an agent's experiences to acquire additional procedural rule knowledge. Without careful engineering, agents using existing algorithms can learn large numbers of overly-specific rules that apply in limited situations. Moreover, these algorithms have difficulties summarizing some types of agent reasoning, producing rules that generate incorrect behavior. The limited generality of the rules learned, the uncertainty of correct behavior, and the amount of engineering required has limited the effectiveness of online procedural learning algorithms.

To better understand this problem, this work defines the two key qualities of online procedural learning: (1) correctness, whether the procedural knowledge generates the same inferences as the reasoning learned from, and (2) optimal generality, whether the procedural knowledge learned captures the generality of the reasoning learned from. These concepts are used to provide a detailed analysis of the underlying deficiencies of the most advanced procedural learning algorithm currently found in a cognitive architecture, Soar's chunking mechanism.

This work also provides an architecture-agnostic analytical framework to understand issues with online procedural learning. First, it provided four necessary conditions for correct behavior summarization, which can be used to organize the correctness issues of an online procedural learning algorithm into a taxonomy. Second, it provides four architecture-agnostic strategies that can be used to remedy issues with online procedural knowledge learning. Third, this work defines four desiderata for an online procedural learning algorithm that can be used to evaluate systems.

This work also describes a robust implementation of this approach. It describes how the four strategies were applied to Soar's chunking algorithm to create a new implementation of dependency-based procedural learning called explanation-based behavior summarization (EBBS), which includes algorithmic mechanisms that remedy or detect each of the 12 correctness issues identified. This work also describes how EBBS uses a novel unification algorithm called Distributed Identity Graph Unification (DIGU) that captures the generality in the reasoning being summarized, interfaces with the various mechanisms that improve correctness by using the formalism of object "identity," and is uniquely designed to handle the computational demands that arise from online procedural learning in a cognitive architecture.

This work concludes with an evaluation that uses data generated from 14 agents across seven different domains. These experiments show that EBBS learns more optimally general rules that capture reasoning that previously could not be captured, effects correct behavior and improves agent performance without sacrificing agent reactivity.

CHAPTER 1

Introduction

Building general AI agents that learn from their own experiences in the world poses many challenges. To achieve generality and functionality, these agents must be implemented in architectures that support a wide range of cognitive capabilities (problem solving, perception, action, planning, language processing). To maintain the reactivity that their environments or embodiments require, these agents must efficiently reason using task-specific knowledge. To maintain task performance in novel situations, they need to deliberate and use general, robust problem-solving capabilities to determine the best course of action (Laird and Rosenbloom, 1990). But, in future similar situations, these agents should use previous experiences without sacrificing reactivity. Our challenge is to develop a general cognitive architecture that supports the transition from using deliberate, general, internal problem-solving knowledge to efficient, reactive, task-specific knowledge (Laird and Mohan, 2018).

For cognitive architectures that encode procedural knowledge as rules, online learning algorithms exist that can analyze an agent's experiences to acquire additional procedural rule knowledge. Even though these algorithms have been successful in some applications (Zhong et al., 2012; Mohan and Laird, 2014), their use has been limited. For reasons we discuss in this thesis, they require careful agent design and task knowledge encoding to ensure correct, general, and high performance learning. Without such careful engineering, agents can learn large numbers of overly-specific rules that apply in limited situations. Moreover, existing algorithms have difficulties summarizing some types of agent reasoning, producing rules that generate incorrect behavior. In summary, the limited generality of the rules learned, the uncertainty of correct behavior, and the amount of engineering required has limited their effectiveness, as reflected in a 2008 paper that considered using procedural learning to improve the performance of autonomous agents that populated a simulation of unmanned aerial vehicles:

> "There is a drawback though; in fact such a characteristic would drastically decrease the predictability of the system, especially if new rules are to be added automatically. This would certainly lead to certification issues. For this reason, the learn-

ing capabilities of Soar have not been used at this stage of the project." (Gunetti and Thompson, 2008)

Robust procedural learning could be invaluable for the many projects that are attempting to use cognitive architectures to build increasingly complex autonomous agents that interactively learn their own task knowledge, encoded in procedural rules, from instruction and exploration (Laird et al., 2017). To provide the capabilities needed to achieve that type of dynamic learning, we need a more complete understanding of the procedural learning process and the limitations that have hampered progress in this area. This thesis attempts to provide such an explication, proposes an approach that addresses the need for a robust, computationally efficient, online procedural learning mechanism with increased generality and reliable correctness, presents a working implementation and evaluates that implementation's effectiveness.

1.1 Online Procedural Knowledge Learning

In this thesis, we limit our discussion to automatic, online algorithms that learn new procedural knowledge based on an agent's problem-solving and task execution. To facilitate the discussion, we will use the shorter term *online procedural learning* to describe computational systems that provide such capabilities. Examples of modern systems that do this are ACT-R (Anderson, 1987) and Soar (Laird et al., 1986).

In general, an online procedural learning algorithm uses a history of step-by-step procedural reasoning (a behavior or execution trace) to deductively construct rules that summarize the necessary aspects of the reasoning described by the trace. In future similar situations, the learned rule fires and adds knowledge in a single step, eliminating the need for multi-step problem-solving. In other words, rather than contemplating and figuring out what to do, the agent immediately knows what to do. Speed-up learning occurs because the learned rule can be applied in one step, which can be significantly less costly than repeating the original problem-solving. Thus, *online procedural learning converts deliberate reasoning into memory retrieval*. Transfer learning occurs when the procedural knowledge learned in one situation applies to other situations that the agent has not reasoned about before.

There are several different ways that an online procedural learning system can learn a rule. The simplest mechanism is memoization, a dynamic programming optimization technique that caches calculations performed and reuses them when the same situation reoccurs. For online procedural learning, a memoization mechanism would learn new rules by storing mappings from the complete, specific state of the agent when reasoning began to the inference made at the end of the chain of reasoning. While such a system is unlikely to be tractable in a cognitive architecture, it is a simple form of dynamic procedural knowledge learning.

Online procedural learning systems found in state of the art cognitive architectures are more complex. For example, ACT-R's online procedural learning mechanism, *production composition*, learns new rules when it detects that two rules have repeatedly fired in sequence. Production composition performs simple variable unification across the two rules to learn a single rule that combines them. Over time, this process can chain together learned rules to summarize longer sequences of rule applications (Taatgen and Lee, 2003). Production composition also incorporates a memoization mechanism that caches retrievals from ACT-R's long-term memory. If one of the rules being combined both retrieves information from long-term declarative memory and performs a problem-solving step, a process called *proceduralization* performs the same step without the retrieval by embedding the retrieved information directly into the learned rule.

In contrast, Soar's online procedural learning algorithm, *chunking*, leverages the architecture's automatic subgoaling mechanism to learn a rule. Whenever an agent does not know how to proceed (an *impasse*), Soar creates a subgoal to generate knowledge that allows progress in the parent goal. For example, ROSIE, a Soar-based robotic agent that learns new tasks using a combination of problem-solving and interaction with an instructor, enters a subgoal whenever it encounters an instruction that references a concept it is not familiar with (Mohan and Laird, 2014). In that subgoal, ROSIE initiates an interaction with the instructor to define the missing concept. For example, consider a situation in which ROSIE is instructed to "put the red block on a clear block" but does not know what the concept "clear" means. Because Rosie cannot determine which blocks are "clear," it impasses, which generates a subgoal to acquire the concept that it does not understand. In the subgoal, rules fire that initiate an interaction with the instructor. For example, these rules may cause ROSIE to say, "I do not know what the concept 'clear' means. Please define." The instructor tells ROSIE that "if an object is not on top of a block then it is clear." Using that instruction, rules fire in the subgoal that create a new knowledge structure in the supergoal that provides the knowledge needed for the agent to proceed. Armed with this new information, the impasse is resolved and ROSIE then proceeds to put the red block on a clear block. *This is when procedural learning occurs*. Whenever a subgoal returns a knowledge structure, or *result*, to a supergoal, chunking analyzes the trace of rules that fired in the subgoal to creates a rule that summarizes the reasoning that led to that result. In future situations, the learned rule will fire, automatically providing the definition of the concept, so that ROSIE does not need to ask again.

There are additional rule-based architectures that learn procedural knowledge dynamically, but they are not good candidates for our research. One reason is that many systems claim online procedural learning but provide neither descriptions of their algorithms nor source code. For example, Polyscheme (Cassimatis, 2002) alludes to a rule learning specialist that is capable of learning new procedural knowledge, but the system is not publicly available and no description of either how it learns rules or how well it learns rules is provided. Another reason is that some systems constrain how knowledge is represented. For example, the PRIMS model (Taatgen, 2013) requires that procedural knowledge is broken down into what Taatgen calls primitive information processing elements, which are small rules that perform the most fundamental operations of a cognitive architecture. While PRIMS is a compelling model of transfer learning, we do not consider it as a viable approach because it would limit the representational flexibility that a general cognitive architecture provides. We also do not consider online procedural learning techniques that learn rules using inductive algorithms. For example, CHREST learns rules that associate long-term memories with patterns in short-term memory by performing statistical analysis of changes in working memory (Gobet and Lane, 2017). Similarly, inductive logic programming generates theories of domain knowledge by considering all possible hypotheses that a set of negative and positive training examples support and choosing the ones that it calculates are most likely to be useful. While important, it is not clear whether these approaches lend themselves to the correctness and real-time requirements of an online procedural learning system in a cognitive architecture and, thus, they fall outside the scope of our examination.

In this work, we study online procedural learning by returning to research on Soar's general learning mechanism, chunking, an approach which has roots in over 30 years of research on cognitive architecture. Soar is a general cognitive architecture for developing systems that exhibit intelligent behavior that strives to be a unified theory of cognition (Laird et al., 1987; Newell, 1992; Laird and Rosenbloom, 1996). It is a mature, well-understood, architecture that possesses multiple additional general learning mechanisms (reinforcement, semantic and episodic), and is actively being used by a community of over 20 research institutions world-wide on a broad range of tasks. While a significant body of work has used Soar to create detailed cognitive models that describe and predict aspects of human behavior in the performance of a task, much of the active work in Soar, both academic and commercial, involves building intelligent agents that encode significant expert knowledge to effect autonomous, capable task-execution in real-world environments, precisely the types of agents that could benefit from a robust and effective online procedural learning system.

Note that ACT-R's production composition algorithm is also a viable candidate for such research, but both the architecture and its online procedural learning algorithm have qualities that suggest chunking is a better fit for the types of agents that we are interested in building. One of the main reasons is that ACT-R is used purely for cognitive modeling, which focuses on simpler behavior and leads to agents that encode less complex forms of reasoning. In contrast, the agents whose performance we are interested in improving execute tasks in demanding, real-world domains that often require complex reasoning, such as hierarchical task decomposition and planning, to achieve desired task performance. Examining this problem in the context of ACT-R may not reveal all the challenges that need to be overcome in a more general architecture.

It is important to point out that there are numerous cognitive architectures in existence today. Kotseruba and Tsotsos (2016) surveyed the last 40 years of research across multiple disciplines, most notably AI, cognitive science and neuroscience, and identifies over 195 different cognitive architectures. Of those, 96 are currently active and nearly all of them have some form of procedural memory. While many of them do not yet have the ability to learn new procedural knowledge, they could one day benefit from online procedural learning. Even some researchers who are exploring neural network based approaches to cognition have identified online procedural learning as a potential, promising capability; for example, Lake et al. (2016) suggests that recent neural net approaches could use an explanation-based learning capability to explain how people are rapidly able to understand the world well:

> "Although neural networks are usually aiming at pattern recognition rather than model-building, we will discuss ways in which these 'model-free' methods can accelerate slow model-based inferences in perception and cognition. By learning to recognize patterns in these inferences, the outputs of inference can be predicted without having to go through costly intermediate steps. Integrating neural networks that 'learn to do inference' with rich model-building learning mechanisms offers a promising way to explain how human minds can understand the world so well, so quickly."

Having a deep understanding of the strengths and weaknesses of online procedural learning, both theoretical and practical, and how that can be improved over the current state of the art could be useful to the entire community. Moreover, an actual implementation of a general and robust online procedural learning algorithm in a modern cognitive architecture affords immediate concrete benefits. Research in autonomous agents can use our work to achieve exciting new learning results, a benefit that is already happening (Kirk and Laird, 2014, 2016; Mininger and Laird, 2016, 2018), while research in cognitive architectures can use this work as a basis to explore further improvements to online procedural learning.

1.2 Contributions of This Thesis

In this thesis, we make the following contributions:

1. We explicate 14 underlying deficiencies in the original implementations of chunking that arose from many of its core assumptions. Specifically, we provide detailed descriptions of two issues that underlie why chunking does not always learn general rules and 12 scenarios in which chunking may learn incorrect behavior summarizations.

2. We provide four architecture-agnostic necessary conditions for correct behavior summarization when performing online procedural learning. These four requirements can be used to organize the varied correctness issues of an online procedural learning algorithm into a taxonomy whose categories correspond to mitigation strategies.

3. We present a general approach to improving online procedural knowledge learning that uses five architecture-agnostic strategies. We demonstrate this approach by applying the five strategies to chunking's generality and correctness issues.

4. To evaluate the effectiveness of our proposed approach, we have developed mechanisms that implement the five strategies of (3) to the problems described in (1). We use these mechanisms as the basis for a completely new implementation of procedural learning called *explanation-based behavior summarization* (EBBS).

    - We describe a novel unification algorithm called Distributed Identity Graph Unification (DIGU) that is informed by insights we've had on the tight relationship between generality and correctness in online procedural learning systems within cognitive architectures. We show how DIGU can capture more of the generality in the reasoning being summarized, interfaces with the various mechanisms that improve correctness by using the formalism of semantic "identity," and is designed to handle the computational demands that arise from online procedural learning in a cognitive architecture.
    - For each of the 14 deficiencies described in our chunking analysis (12 correctness and 2 generality issues), we describe the new algorithmic mechanisms that implement our aforementioned strategies. For example, we describe how correctness issues can be avoided by creating missing dependencies in a behavior trace, selectively specializing problematic aspect of an agent's reasoning, rule repair or modifying the architecture.

5. We define four desiderata for an ideal online procedural learning algorithm that are architecture-agnostic and can be used to evaluate an online procedural learning system.

6. Finally, we examine how well our approach performs against the desiderata by empirically evaluating our implementation to determine its impact on correctness, generality and performance. Specifically, we examine how well EBBS ameliorates Soar's online procedural learning deficiencies by performing experiments in seven different domains.

1.3 Structure of Remainder of Thesis

In Chapter 2, we lay the groundwork of our evaluation by defining correctness and generality, two key concepts needed to explore online procedural learning. We also describe four requirements that must be met for correct online procedural knowledge learning. We conclude the chapter by providing a list of desiderata for an ideal online procedural learning system.

In Chapter 3, we describe the problem this thesis tackles by evaluating Soar's online procedural learning algorithm, chunking. We first provide a quick overview of how Soar works and how chunking learns rules. Then, to illustrate chunking's strength, we describe several past projects that have successfully used chunking to improve agent performance. Finally, we present the core arguments of the chapter by explicating the deficiencies chunking has with generality and correctness and showing how those deficiencies have hampered the effectiveness and adoption of Soar's online procedural learning capabilities.

In Chapter 4, we present our approach. We detail five general strategies that our proposed system uses to eliminate or mitigate the problematic issues that were described in Chapter 3.

In Chapter 5, we explain how we apply our strategies to learns rules that capture more of the underlying generality in an agent's reasoning. We explain our semantic identity formalism, provide an overview of the EBBS's implementation and describe in detail how the Distributed Identity Graph Unification algorithm works.

In Chapter 6, we discuss the twelve types of reasoning that we have identified that can lead to chunking learning incorrect rules. For each of these twelve correctness issues, we show how EBBS applies one of our five strategies to mitigate the underlying problem to learn rules that are correct summaries of an agent's reasoning.

In Chapter 7, we evaluate our implemented system using experiments in 8 different domains. This evaluation includes metrics that correspond to each of the desiderata described in Chapter 2, which are used to measure the effectiveness of our approach.

In Chapter 8, we summarize our work and conclude the thesis.

CHAPTER 2

Evaluating Online Procedural Knowledge Learning

To discuss the strengths and limitations of any online procedural learning algorithm, this chapter first defines and the discusses *two key metrics of the quality of procedural summarization*: correctness and generality.

2.1 Correct Behavior Summarization

Correctness describes whether a learned rule accurately summarizes the reasoning that was learned from.

A learned rule is a correct summarization if it always generates the same knowledge structures that the original, deliberate reasoning would have in the same situation. In other words, if the learned rule were not available, the agent would fall back to reasoning deliberately and produce the same exact knowledge structures that the learned rule would create.

Our definition of correctness is completely unrelated to whether the original reasoning produces optimal or even beneficial behavior in a particular task or domain. Correctness in this work only refers to whether the learned rules accurately summarize the inferences that the original reasoning would perform in the same situation.

In an ideal algorithm, learned rules are always correct. Ensuring correctness is critically important because, when an incorrect rule is used, the agent may make unjustified inferences, which could lead to behavior not intended by the agent engineer. This may disrupt or prevent successful task execution.

2.2 Necessary Conditions for Correct Behavior Summarization

One of insights that we have gained in this work is that correctness issues occur when an algorithm attempts to summarize a trace of behavior that is either deficient on some dimension or describes behavior that cannot be recreated by a single rule firing. Specifically, we have found four architecture-agnostic types of violations that lead to incorrect rules, and we use as them as the basis of the following *four necessary conditions that reasoning must meet for it to be correctly summarized*:

1. **Feasible: Reasoning Must be Logically Consistent and Contemporaneously Possible**

   Unless the architecture provides a mechanism to prevent it, a behavior trace could include a rule whose conditions logically conflict with those of another rule in the trace. For example, a behavior trace may include both a rule that matches when a feature has a particular value and another rule that matches after the feature changes and has a different value. If an online procedural learning algorithm summarizes a trace that includes both of these rules, it will learn a rule that has two conditions that can never be simultaneously satisfied. Similarly, a trace can have a rule whose conditions can never co-occur with the conditions of another rule because of the nature of the domain. For example, consider a trace that includes both a rule that tests the agent's location before the agent issues a command to move and a rule that tests the agent's location after the move. If an online procedural learning algorithm summarizes a trace with those two rules, it will learn a rule that requires the agent to be at both the starting and ending location at the same time. In summary, any explanation-based approach that attempts to learn from a trace whose reasoning is not logically consistent and contemporaneously possible can learn rules that will never match and are incorrect (Wray and Laird, 2003).

2. **Complete: Reasoning Must Be Described By a Behavior Trace That Explains All Dependencies**

   An online procedural learning algorithm can only learn a correct rule if it analyzes a behavior trace that describes all knowledge structures that the reasoning is dependent on. As previously discussed, dependencies may be missing from a behavior trace because the trace is only an approximation of the processing that underlies an agent's reasoning in a cognitive architecture. Describing all causal dependencies is a basic requirement of any explanation-based approach. If an online procedural learning algorithm does not include an aspect of the problem solving context that was required for the problem solving to occur, it learns a rule that may match in situations where that necessary aspect does not exist. Such rules are overgeneral and hence incorrect.

3. **Expressible: Reasoning Must be Expressible in a Single Rule**

   An online procedural learning system that summarizes reasoning into a single rule can only summarize *reasoning that can be expressed in a single rule using the architecture's rule syntax*. This is a tautological requirement that is a result of the fact that limitations in an architecture's rule syntax can make it possible for multiple rules to describe reasoning that a single rule cannot. In short, an online procedural learning system cannot summarize something that it does not have the language to describe.

4. **Repeatable: Reasoning Must be Reliably Repeatable**

   The reasoning that an online procedural learning algorithm summarizes must be reliably repeatable. For example, in Soar, declarative knowledge is propositional and rules fire deterministically, so a chunk will always fire if all of its conditions match against working memory. This means that the behavior that the rule summarized must be equally reliable. This may not be the case if subgoal reasoning involved uncertain knowledge or probabilistic decision-making, both of which can exist in Soar and are common capabilities in cognitive architectures.

2.3 Capturing Generality of Summarized Reasoning

Generality describes when a learned rule can be used.

We are **not** using a definition of generality that implies that a general rule applies in a broad range of situations, another common notion of generality. Because an online procedural learning algorithm attempts to summarize a historical trace of an agent's reasoning, its goal is to learn a rule that applies to the same space of situations that the agent's existing procedural knowledge applies to. So, the *optimal* level of generality is that space of situations. Consequently, our work uses a *relative* definition of generality that compares the space of situations a learned rule applies to, to the space of situations the original reasoning applies to.

Pane A of Figure 2.1 depicts this space of situations graphically. The point in the middle represents the situation in which learning occurred, while the dotted circle represents all similar situations in which the same original reasoning would also apply.

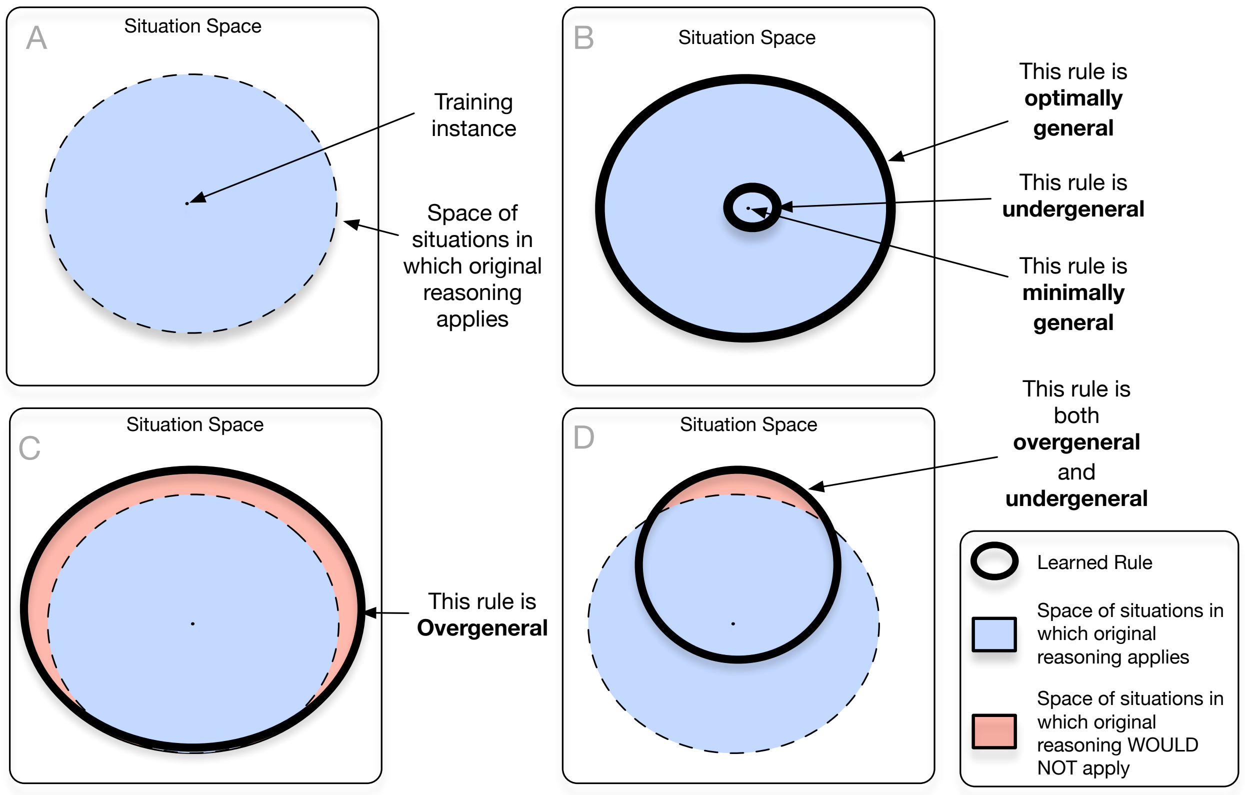

A
Situation Space
Training instance
Space of situations in which original reasoning applies
B
Situation Space
This rule is optimally general
This rule is undergeneral
This rule is minimally general
C
Situation Space
This rule is Overgeneral
D
Situation Space
This rule is both overgeneral and undergeneral
Learned Rule
Space of situations in which original reasoning applies
Space of situations in which original reasoning WOULD NOT apply

Figure 2.1: A graphical depiction of the generality of rules.

- A learned rule is *optimally general* if it can apply in all situations that the original domain knowledge would have applied in and no others. In pane B of Figure 2.1, the optimally general rule is the region with the thick border that covers the same space of situations as the original reasoning did in pane A.

- A learned rule is *minimally general* if it can only apply in the exact same situation that occurred when learning. In pane B of Figure 2.1, the minimally general rule covers one situation, which is the same one as the training instance in the left pane.

- A learned rule is *undergeneral* if it can only apply in a subset of the situations that the original domain knowledge would have applied in and no others. In pane B of Figure 2.1, the undergeneral rule is the smaller, circular region that covers a subset of the situations that the original reasoning can apply to.

- A learned rule is *overgeneral* if it can apply in situations beyond those that are supported by the reasoning in the original domain knowledge that applied. In pane C of Figure 2.1, the overgeneral rule is the region with the thick border that covers a larger space of situations than the original reasoning does.

In an ideal algorithm, all learned rules are optimally general. In practice, though, online procedural learning algorithms often learn rules that apply only in a subset of the situations that the original domain knowledge would have applied in.

2.4 Generality's Relationship to Correctness

We classify overgeneral rules as incorrect rules because they can make unjustified inferences when they are applied in the situations denoted by the region colored red in Figure 2.1. They are considered incorrect because they produce inferences in situations in which the original reasoning would produce no inferences at all. Similarly, an undergeneral rule is considered incorrect because, in some situations, it is not able to generate the same inferences that the original reasoning would.

In terms of task performance, an overgeneral rule is typically a more important problem for an agent than an undergeneral one because an overgeneral rule can change an agent's behavior adversely by producing inferences that may not be justified. In contrast, an undergeneral rule would simply not fire in some situations, which would lead the agent to fall back to using its original, deliberate reasons.

Note that a rule can be both overgeneral and undergeneral, as shown in pane D of Figure 2.1. Because overgenerality is more problematic for agents, we treat and refer to such rules simply as overgeneral.

2.5 Computational Benefits of Optimal Generality

As previously discussed, our definition of generality does not imply that a general rule applies in a broad range of situations. The size of the space of applicable situations that a rule can apply in, i.e. the absolute size of the region in the dotted circle, could be small or large depending on how abstract the reasoning being summarized. As a result, we use a limited, *relative* definition of generality that is relative to the space of situations that the original reasoning applies to. This allows us to assert that an online procedural learning system should always attempt to learn optimally general rules. This is a fairly trivial statement and is equivalent to other explanation-based learning systems saying that unification algorithms should always return the most general unification of variables (MGU) that an explanation supports. And, that is indeed the case; the only example of an EBL unification algorithm that does not return the MGU was one that was considered buggy (Dejong and Mooney, 1986).

We can visualize why we can always prefer optimal generality by considering Figure 2.2, which depicts four undergeneral rules that came out of the same reasoning. But first, we must define what we mean when we say that a learned rule provides computational benefits to an agent.

There are several ways that a rule can provide a computational benefit to an agent. One way is when it allows the agent to replace multiple rule firings with a single rule firing. Assuming the cost of learning, matching and executing a single rule is less than the cost of matching and executing the set of rules being summarized,[1] the computational benefit gained is faster execution time. Learned rules can also improve execution time by eliminating the cost of creating the subgoal and any rules that fired in the subgoal that did not participate in the chunk's summarized reasoning. Moreover, if the chunk summarizes reasoning that retrieves knowledge from a long-term memory store, it provides the additional computational benefit of eliminating the cost of the memory retrieval. Finally, if the learned rule summarizes reasoning that spanned multiple primitive cognitive decision cycles, it also eliminates other types of processing that cognitive architectures often perform every decision cycle, for example perception and output, which can require significant computational processing.

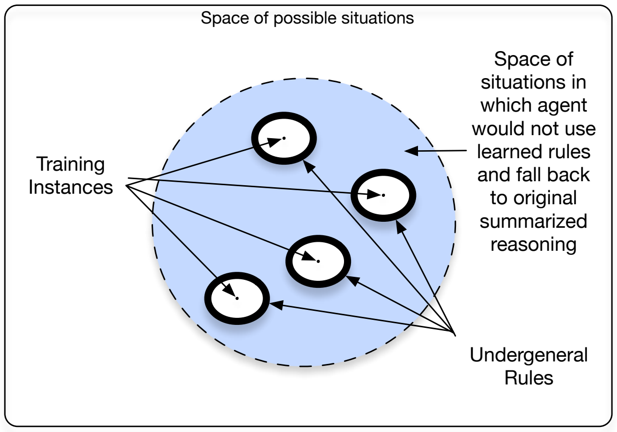

Space of possible situations
Space of situations in which agent would not use learned rules and fall back to original summarized reasoning
Training Instances
Undergeneral Rules

Figure 2.2: Situations when undergeneral rules don't offer computational benefits.

Figure 2.2 illustrates why optimally general rules provide computational benefits and are desirable. The dotted line again indicates the space of all situations that the original reasoning can apply in. The key thing to notice is the colored region, which denotes all situations in which a rule from the set of undergeneral learned rules cannot apply. Whenever the agent is in a situation in the colored region, the rules in the original deliberate reasoning would fire and produce the appropriate inferences. Note that an optimally general rule could also make the same inferences in the situations denoted by the colored regions. So, an optimally general rule could produce the same inferences as a combination of the less general rule and original reasoning would have. The difference is that, when in situations from the colored region, the optimally general rule generates the correct results in a single rule firing. Thus, in this analysis, the optimally-general rule is computationally more desirable than an undergeneral rule.

[1]For the moment, we ignore the utility problem (Mooney, 1989), i.e. situations in which our assumption is violated and the persistent cost of matching a learned rule outweighs the periodic performance improvements that occur when the learned rule fires and deliberate reasoning is avoided. Whether these "expensive chunks" arise and how overall match cost changes with an improved online procedural learning algorithm is discussed when we evaluate our system in Chapter 7. We also ignore the average growth issue (Doorenbos, 1995), i.e. the performance decrease that occurs as procedural memory grows. Not only is it not currently a problem — modern rule matchers like RETE can efficiently detect matches in procedural memories with millions of rules — we expect that this issue should be less problematic with an improved online procedural learning system that learns fewer, more general rules. We also evaluate whether this is true in Chapter 7.

2.6 Desiderata for An Online Procedural Learning Algorithm

Before we explore the specifics of online procedural learning in a particular cognitive architecture, we first establish what qualities we want an ideal algorithm to have. We posit the following desiderata, which we use to compare online procedural learning algorithms and evaluate improvements:

D1. Learns Only Correct Rules

Algorithm learns rules that produce the same knowledge structures that the original, deliberate reasoning would have in the same situations, ***from all types of reasoning*** *supported by the architecture and* ***from all sources of knowledge*** *available to the agent*

In other words, if the agent inhibits the firing of a learned, correct rule, the agent would reason deliberately and produce the same exact knowledge structures that the inhibited learned rule would have added. This criterion is critically important because an incorrect rule can produce effects that were not intended by the agent engineer and may hinder or prevent task execution. Moreover, an online procedural learning system in a cognitive architecture must support all of the cognitive capabilities and knowledge sources that the architecture makes available to the agent. Learning correct rules should not be limited to easily summarized reasoning or a subset of available knowledge sources.

D2. Learns Only Optimally General Rules

The algorithm learns optimally general rules ***regardless of how an agent represents or processes knowledge****.*

This desideratum favors algorithms that always learn optimally general rules. This is important because an optimally general rule is more likely to match future situations than an undergeneral one, and hence more likely to replace slower, deliberate reasoning with the immediate execution of a chunk. Moreover, an ideal online procedural learning system automatically learns optimally general rules from any agent, regardless of its design; it should not be limited to knowledge representations and reasoning that can accommodate the limitations of the learning algorithm.

D3. Learns Rules that Improve Future Performance

In future situations, learned rules replace deliberate reasoning over multiple rule firings with immediate inferences from a single rule firing.

This desideratum favors algorithms that learn rules which produce these replacements earlier and/or more often. It also favors algorithms that can provide computational benefits for agents designs and learning strategies that cannot use existing online procedural learning algorithms.

D4. Learns Without Negatively Affecting Reactivity

The algorithm must not incur computational costs that prevents the agent from responding with the reactivity that their embodiment or domain requires. This includes both the cost of initially learning rules and the cost of using that learned knowledge.

In Soar, we define *reactivity* as the duration of an agent's primitive decision cycle and *required reactivity* as the maximum amount of time that an agent's embodiment or domain allows between decisions. For example, cognitive architectures often strive to ensure that their decision cycles take less than approximately 50ms, the primitive cycle time that psychologists hypothesize occurs in the human brain (Newell, 1994). Similarly, agents embodied in robotics applications often have a requirement that the maximum duration of its primitive cycle time is less than 100ms (Laird et al., 2012b).

This desideratum favors algorithms that can learn procedural knowledge from task execution while maintaining the reactivity that their agent's embodiment or domain requires. These criteria are important because we are interested in improving the performance of real-world agents that operate in environments with real-time constraints. The computational cost of the learning algorithm includes both the overhead that occurs when the algorithm analyzes behavior and learn rules and the increased match cost that occurs as a result of the additional learned rules being added to production memory. These costs must not hinder the agent's required reactivity.[2]

There are trade-offs between these desiderata – techniques that improve how well the algorithm does on one desideratum may hurt how well the algorithm does on another desideratum – so we do not expect our proposed system to achieve complete success on every dimension. An obvious example is that adding new mechanisms to improve D1 and D2 will almost certainly add computational overhead and degrade D3. Similarly, we have found that we can improve D2 by slightly degrading D1. Specifically, we have found that we can sometimes repair an incorrect rule by specializing problematic aspects of the reasoning, which decreases the generality of the rule but ensures correctness. How well a particular agent does against the desiderata will be highly dependent on an agent's knowledge representation, problem-solving strategies and the nature of its environment. For some representations and types of reasoning, improving the generality of the

[2]Note that previous work has examined the algorithmic complexity of both chunking and the mechanisms utilized by our proposed algorithm (Kim and Rosenbloom, 2000; Mooney et al., 1986). In Chapter 7, we discuss the applicability of their work but do not provide a similar worst-case complexity analysis.

rules that an online procedural learning algorithm learns might not improve performance; a simpler mechanism may suffice.

CHAPTER 3

Soar's Online Procedural Knowledge Learning Algorithm

To facilitate a discussion of the issues that have hampered the effectiveness and adoption of chunking, this section provides a quick overview of the philosophy behind Soar and how it learns procedural knowledge.[1] While the following discussion is obviously Soar-centric, it focuses specifically on the aspects of Soar's philosophy and implementation that are necessary for chunking to work, most notably universal subgoaling and its goal-centric working memory design. The fact that there are 96 actively maintained cognitive architectures currently available (Kotseruba and Tsotsos, 2016), many of which share similar principles, suggests that other architectures may be able to use the lessons we learned in this project to build or at least inform their own online procedural learning systems.

3.1 Soar's Problem Space Computational Model

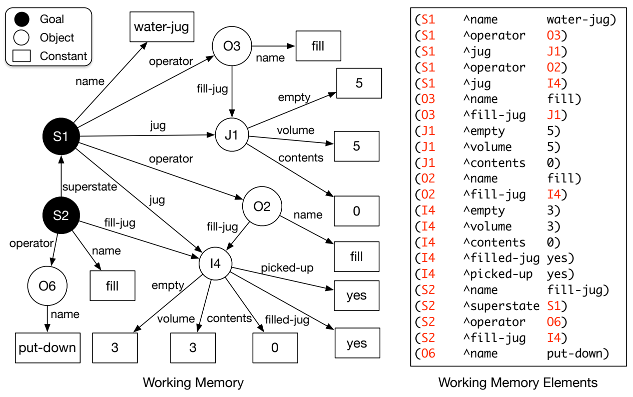
Soar is predicated on Allen Newell's problem space hypothesis, which states that any goal-oriented problem can be formulated as a search in a problem space, and, thus, problem spaces are a critical component of reasoning, problem solving and decision-making (Newell, 1980). A problem space is the set of all possible states along with a set of operators that transform a particular state within the problem space to another state. Given an initial state and a desired state, operators are iteratively selected and applied in an attempt to reach the goal state. The series of steps from the initial state to a desired state forms the solution or behavior path.

Soar incorporates a problem-space computational model (PSCM) that was implemented specifically to explore and refine Newell's problem-state hypothesis (Newell, 1980). A PSCM is any computational model designed around the concept of problem spaces. A spectrum of architectures

[1]An optional primer on Soar terminology is also provided in Appendix A. For a thorough treatment of the philosophy behind Soar, please see chapters 2-4 of *The Soar Cognitive Architecture* (Laird, 2012).

exists that can be described as PSCMs; they range in complexity from early production systems, like Ops5 and STRIPS, to modern cognitive architectures like Soar (Laird, 2012).

Soar's working memory (WM) is a declarative representation of all knowledge that is immediately available for an agent's problem solving and can be represented as a directed graph as shown in Figure 3.1. WM is composed of primitive representational units of knowledge called working memory elements (WMEs). Each WME consists of three symbols: an *identifier*, which references a node in the graph by using a special symbol called a "Soar identifier" (denoted as circles in the figure), an *attribute*, which specifies a feature label (denoted as edges), and a *value*, which specifies either a constant value (denoted as rectangles) or a Soar identifier (SI), i.e. another node in the graph (denoted as circles).



```
(S1    ^name        water-jug)
(S1    ^operator    O3)
(S1    ^jug         J1)
(S1    ^operator    O2)
(S1    ^jug         I4)
(O3    ^name        fill)
(O3    ^fill-jug    J1)
(J1    ^empty       5)
(J1    ^volume      5)
(J1    ^contents    0)
(O2    ^name        fill)
(O2    ^fill-jug    I4)
(I4    ^empty       3)
(I4    ^volume      3)
(I4    ^contents    0)
(I4    ^filled-jug  yes)
(I4    ^picked-up   yes)
(S2    ^name        fill-jug)
(S2    ^superstate  S1)
(S2    ^operator    O6)
(S2    ^fill-jug    I4)
(O6    ^name        put-down)
```

Figure 3.1: Working memory contents of an agent solving the waterjug problem after a subgoal is created.

Soar identifiers serve to organize a set of features into an object. So, the set of WMEs that share the same identifier constitute a description of an object, with each WME specifying either a feature or a link to another object. Soar's working memory is rooted in a Soar identifier that represents the agent's top-level goal (denoted as the filled circle S1).

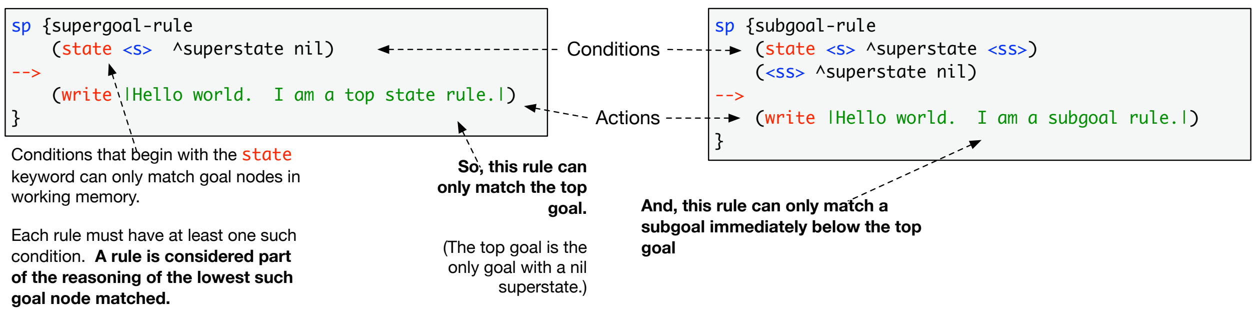
For clarity, please be aware that goals are also referred to as states in Soar. Given the emphasis on learning from subgoaling, we will use the term goal in our discussion. We point this out because some examples that use real Soar syntax will still use the term state and superstate instead of goal and supergoal.

Chunking is designed around one of the key features of Soar's PSCM: universal subgoaling (Laird, 1983). Whenever knowledge is not sufficient to make a decision for the current goal, the agent reaches an "impasse." Soar's response to an impasse is to create a new subgoal, which is subordinate to the first goal, to remedy the lack of knowledge that instigated the impasse, for example if the agent did not know how to implement an operator. Specifically, when a new subgoal is created, Soar adds a new node to working memory to represent that subgoal. Soar then adds features to the subgoal that provide meta-information to describe the impasse.[2] Finally, Soar connects the subgoal

[2]For clarity, Figure 3.1 does not include any of this meta-information except for the superstate link.

to its parent by adding a `superstate` feature to the subgoal. In our example, the superstate link that connects S2 to S1 is provided by the WME `(S2 ^superstate S1)`.

Soar requires that all rules specify which goal or class of goals that its reasoning applies to. Specifically, Soar requires that a rule provides a specification of the subgoal it can apply in by including at least one condition that tests a goal node. Figure 3.2 shows how a rule specifies which subgoal it is reasoning within.



```
sp {supergoal-rule
    (state <s>  ^superstate nil)
-->
    (write |Hello world.  I am a top state rule.|)
}
```

```
sp {subgoal-rule
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil)
-->
    (write |Hello world.  I am a subgoal rule.|)
}
```

Conditions that begin with the state keyword can only match goal nodes in working memory.

Each rule must have at least one such condition. **A rule is considered part of the reasoning of the lowest such goal node matched.**

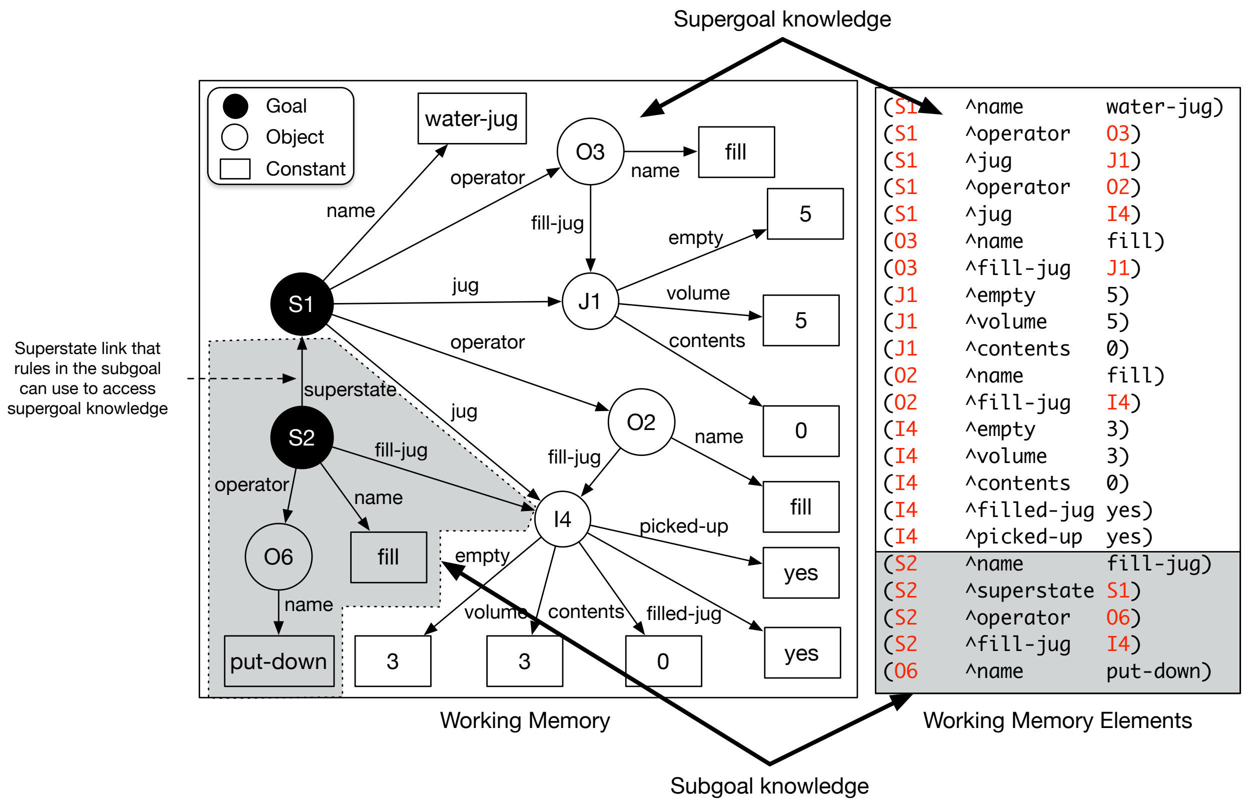
So, this rule can only match the top goal.

(The top goal is the only goal with a nil superstate.)

And, this rule can only match a subgoal immediately below the top goal

Figure 3.2: How rules specify which subgoals their reasoning applies to.

The goal-centric design of Soar's procedural memory, specifically how the conditions of all rules are rooted in a test of a goal node, allows Soar to demarcate reasoning by subgoals and create separate behavior traces for each subgoal. It does this by defining reasoning in a subgoal as all rule firings that have a condition that matched against that subgoal and no subgoal below that subgoal. Using this definition, Soar incrementally generates behavior traces. So, whenever a rule matches, Soar determines the lowest goal node tested by a condition and records the rule firing in the behavior trace of that subgoal. Figure 3.5 shows a simple behavior trace that was created from four rules that all tested the same subgoal identifier S2.



```
(S1   ^name        water-jug)
(S1   ^operator    O3)
(S1   ^jug         J1)
(S1   ^operator    O2)
(S1   ^jug         I4)
(O3   ^name        fill)
(O3   ^fill-jug    J1)
(J1   ^empty       5)
(J1   ^volume      5)
(J1   ^contents    0)
(O2   ^name        fill)
(O2   ^fill-jug    I4)
(I4   ^empty       3)
(I4   ^volume      3)
(I4   ^contents    0)
(I4   ^filled-jug  yes)
(I4   ^picked-up   yes)
(S2   ^name        fill-jug)
(S2   ^superstate  S1)
(S2   ^operator    O6)
(S2   ^fill-jug    I4)
(O6   ^name        put-down)
```

Figure 3.3: Goal nodes in WM allows chunking to demarcate the knowledge accessible to each goal.

Similarly, the goal-centric design of Soar's working memory, specifically how all knowledge is rooted in a stack of goal nodes, allows Soar to demarcate exactly which knowledge structures are available to each goal. Soar defines a knowledge structure's subgoal as the highest goal for which there is a path from its goal node to the knowledge structure. For example, in Figure 3.3, Soar would consider O6 "subgoal knowledge" with respect to goal S2 because a path of WMEs exists from S2 to O6 but no path exists from any higher-level goal to O6. In contrast, Soar would consider I4 "supergoal knowledge" (in respect to S2) because a path exists from higher-level goal, S1, to I4. Note that it does not matter that a path also exists from S2 directly to I4.

For clarity, we use the following terms interchangeably to refer to knowledge structures that are accessible to a subgoal: subgoal knowledge, local knowledge or, idiomatically, "knowledge in a subgoal." Similarly, we use the following terms interchangeably to refer to reasoning that matched a subgoal and no lower subgoal: subgoal reasoning, local reasoning or, idiomatically, "reasoning in a subgoal."

Note that subgoal rules are not limited to testing subgoal knowledge. They can still access knowledge in higher goal states by referencing a special architectural feature of every subgoal, the superstate link, which provides a way to reference the supergoal node. Because supergoal knowledge is all linked to the supergoal node, this link allows subgoal rules to access supergoal knowledge.[3]

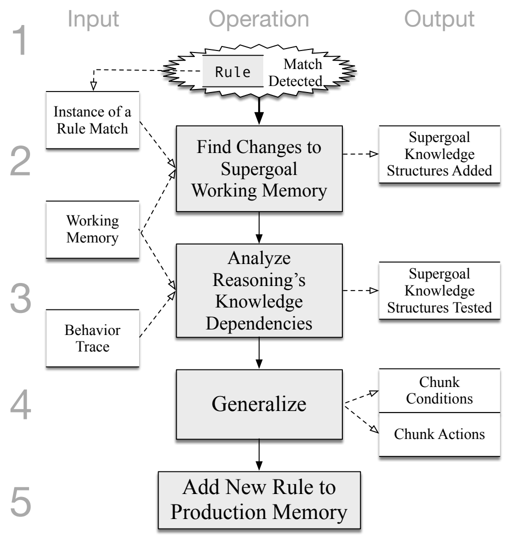
In the next section, we show how chunking builds the framework for a new rule by using these demarcations. Specifically, we describe how chunking builds the framework for the rule by determining how *knowledge in a supergoal* relates to *reasoning in a subgoal*.

3.2 Chunking: Soar's Online Procedural Learning Algorithm

To learn new procedural knowledge, Soar performs the following five basics steps, each of which is labeled in Figure 3.4. A full-page version of the same figure that includes the data structures tested and generated by each step is provided immediately after in Figure 3.6.

1. Rule match detected: The learning process begins whenever a rule matches in a subgoal. Soar creates an instantiation of the rule and adds it to a behavior trace for that subgoal. *The instantiation is a record of the rule match and contains all of the working memory elements that matched the rule along with the new working memory elements that the rule adds.*

2. Find changes to supergoal knowledge: The next thing chunking must do is detect whether there is an opportunity to learn at all, i.e. it must determine whether problem solving in the subgoal has led to new knowledge structures accessible to the supergoal. To use Soar terminology, chunking detects when a subgoal returns a *result* to a supergoal. To do this, chunking uses the goal demarcation provided by Soar's working memory design to collect all working memory elements of the rule's actions that will be accessible to a supergoal. (If the rule only adds subgoal knowledge, chunking takes no further action.)



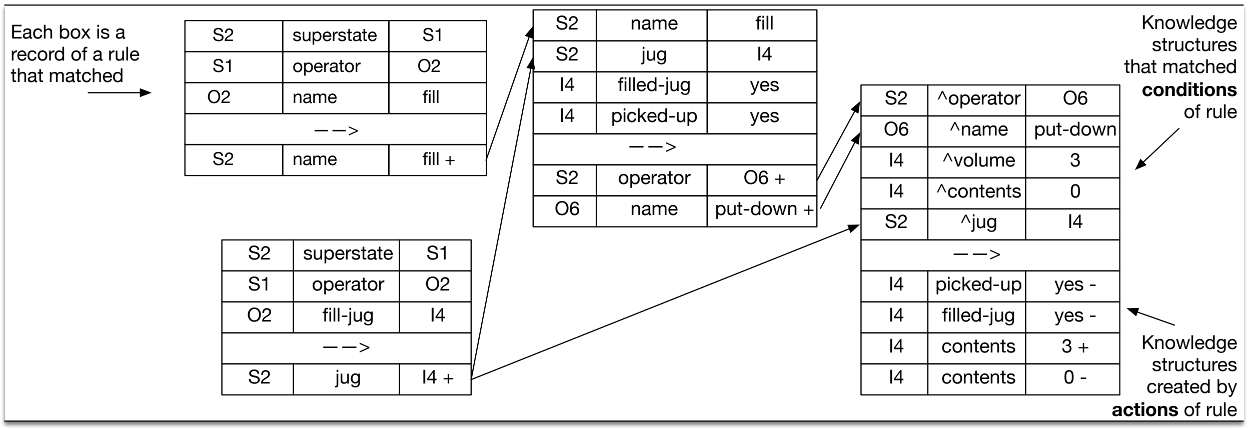
Input
Operation
Output
1
Rule
Match Detected
Instance of a Rule Match
2
Find Changes to Supergoal Working Memory
Supergoal Knowledge Structures Added
Working Memory
3
Analyze Reasoning's Knowledge Dependencies
Supergoal Knowledge Structures Tested
Behavior Trace
4
Generalize
Chunk Conditions
Chunk Actions
5
Add New Rule to Production Memory

Figure 3.4: A simplified overview of chunking.

3. Analyze reasoning's knowledge dependencies: If chunking determines that a result has been generated, it analyzes the training instance in the form of a behavior trace of all rules that fired in the subgoal and played a role in the supergoal knowledge structure that was added to the supergoal. Figure 3.5 shows a simple behavior trace of reasoning that consists of four rules that fired in a

[3]Note that Soar does not add a similar `substate` feature to a goal when it spawns a subgoal for it. Rules that fire in a goal never directly access knowledge in their subgoals.

subgoal for an agent solving the waterjug problem.


Each box is a record of a rule that matched
Knowledge structures that matched **conditions** of rule
Knowledge structures created by **actions** of rule

S2	superstate	S1
S1	operator	O2
O2	name	fill
——>		
S2	name	fill +

S2	superstate	S1
S1	operator	O2
O2	fill-jug	I4
——>		
S2	jug	I4 +

S2	name	fill
S2	jug	I4
I4	filled-jug	yes
I4	picked-up	yes
——>		
S2	operator	O6 +
O6	name	put-down +

S2	^operator	O6
O6	^name	put-down
I4	^volume	3
I4	^contents	0
S2	^jug	I4
——>		
I4	picked-up	yes -
I4	filled-jug	yes -
I4	contents	3 +
I4	contents	0 -

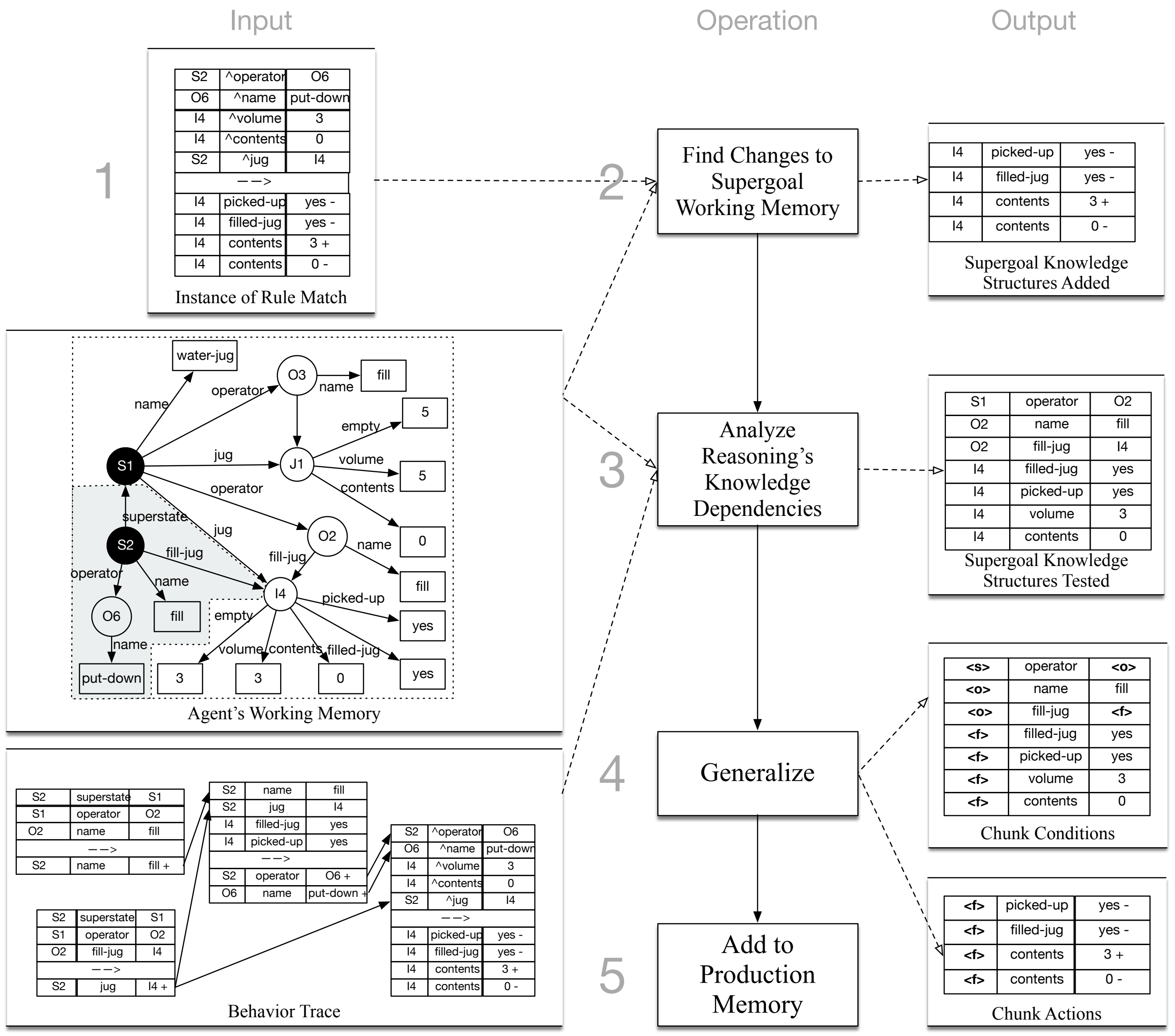
Figure 3.5: Simple behavior trace from an agent that solves the waterjug problem.

An instantiation describes the exact knowledge structure that matched each condition of its corresponding rule in procedural memory. In other words, each box in Figure 3.5 is an instance of a rule firing, not the original rule itself.

To determine which aspects of the supergoal were necessary for the reasoning to occur, chunking uses the goal demarcation provided by working memory to collect all working memory elements in the trace that are supergoal knowledge structures. This portion of the algorithm, which we call dependency analysis, takes an instantiation that creates a result and examines every WME that matched each of its condition. If a WME is supergoal knowledge, the algorithm adds it to the chunk's condition list. If the WME is subgoal knowledge, chunking does not add it but instead examines the rule that created that WME in the subgoal. By recursively backtracing through all such rules, the dependency analysis component produces a description of the features of the supergoal that were necessary for the problem solving to occur.

4. Generalize: At this point, chunking possesses both a list of the *specific* working memory elements that were tested from the supergoal, which will ultimately be transformed into the conditions of the rule, and a list of the specific working memory elements that will be added to the supergoal, which will be transformed into the actions of the rule. To generalize the rule and allow it to apply to a broader range of situations, the fourth step of chunking replaces some elements in both of the WME lists with variables. We will skip how the generalization step works for now and revisit this example in Section 3.6, which discussing issues with generality.

5. Add new rule to production memory: Chunking adds a new rule to long-term procedural memory that uses the generalized description of the problem-solving context as the conditions and the generalized description of the knowledge structures added to the supergoal as the actions.



Instance of Rule Match

S2	^operator	O6
O6	^name	put-down
I4	^volume	3
I4	^contents	0
S2	^jug	I4
— —>		
I4	picked-up	yes -
I4	filled-jug	yes -
I4	contents	3 +
I4	contents	0 -

Supergoal Knowledge Structures Added

I4	picked-up	yes -
I4	filled-jug	yes -
I4	contents	3 +
I4	contents	0 -

Supergoal Knowledge Structures Tested

S1	operator	O2
O2	name	fill
O2	fill-jug	I4
I4	filled-jug	yes
I4	picked-up	yes
I4	volume	3
I4	contents	0

Chunk Conditions

<s>	operator	<o>
<o>	name	fill
<o>	fill-jug	<f>
<f>	filled-jug	yes
<f>	picked-up	yes
<f>	volume	3
<f>	contents	0

Chunk Actions

<f>	picked-up	yes -
<f>	filled-jug	yes -
<f>	contents	3 +
<f>	contents	0 -

Behavior Trace

S2	superstate	S1
S1	operator	O2
O2	name	fill
— —>		
S2	name	fill +

S2	superstate	S1
S1	operator	O2
O2	fill-jug	I4
— —>		
S2	jug	I4 +

S2	name	fill
S2	jug	I4
I4	filled-jug	yes
I4	picked-up	yes
— —>		
S2	operator	O6 +
O6	name	put-down +

S2	^operator	O6
O6	^name	put-down
I4	^volume	3
I4	^contents	0
S2	^jug	I4
— —>		
I4	picked-up	yes -
I4	filled-jug	yes -
I4	contents	3 +
I4	contents	0 -

Figure 3.6: Data structures used and created when chunking learns a rule in a waterjug agent.

3.3 Chunking's Strengths

Because the majority of this work will naturally focus on the deficiencies of chunking that need to be improved, we must point out that the chunking has many strengths.

Chunking is a general mechanism that is task-independent and can be used for a wide variety of problem-solving strategies (Rosenbloom et al., 1993). It can learn knowledge from all PSCM functions: state elaborations, operator proposal, operator evaluation, and operator application. It can learn from any type of reasoning that can be encoded in Soar and processed in a subgoal, for example analogy, look-ahead search, task decomposition or planning (Steier et al., 1987). Chunking can transform any of these methods from deliberate reasoning into an efficient, compiled rule.

Chunking can also be combined with other learning mechanisms. For example, chunking can convert deliberate retrievals from long-term memory stores into automatic retrieval from procedural memory. Chunking can even be used to inductively learn new categories (Miller and Laird, 1996).

By replacing a set of rule firings with a single rule firing, chunking allows an agent to dramatically decrease the amount of time it takes to solve problems that are similar to ones that it has seen in the past. This can produce an exponential collapse in the reasoning required to solve a problem, which, for some agents, can be necessary to achieve the reactivity required by an environment. Figure 3.7 shows how chunking decreases the number of processing cycles required by an agent that uses depth-first look-ahead search with two different heuristics to solve a blocks-world problem(Laird, 2012).

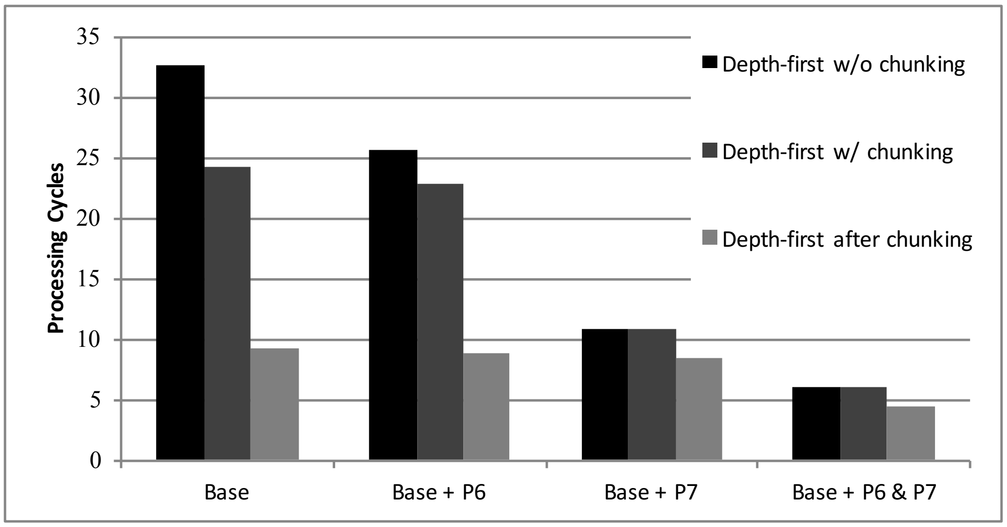


Figure 3.7: Computational benefits of chunking in agent solving blocks world problems.

Figure 3.8 shows an even more dramatic decrease in an agent that performs a graph search using an A* heuristic.(Laird, 2012) These results suggest that learned procedural knowledge is re-used and provides benefits.

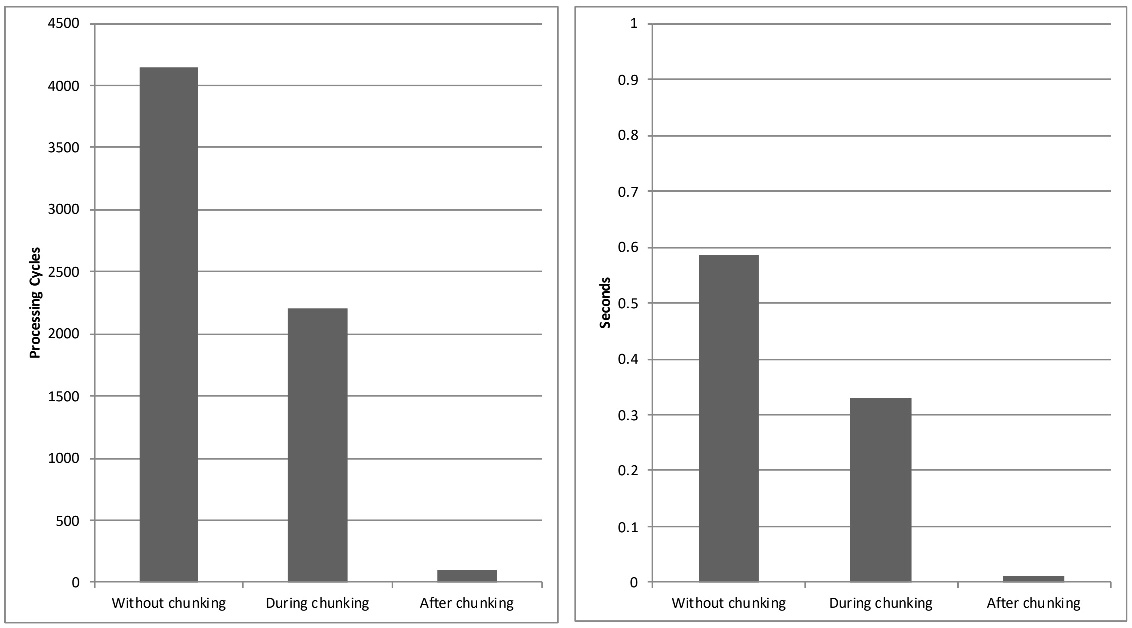


Figure 3.8: Computational benefits of chunking in agent performing graph search.

Another strength of chunking is that it compiles knowledge into a form that other learning algorithms can use. For example, in 2011, Laird et al. (2012a) used chunking as a component of a Soar agent that performs extensive probabilistic reasoning to play the game Liar's Dice.[4] What is interesting about this agent is that it used chunking to learn value functions that incorporated domain knowledge that were then tuned by reinforcement learning. Two examples of these rules are shown in Figure 3.9. This learning strategy was effective and resulted in a novel contribution that was not possible without chunking.

```
Rule 1: If the operator is to bid five 2's with no push and
           there are zero 1's and one 2 dice and
           there are four unknown dice
        then create an operator preference of -0.8754

Rule 2: If the operator is to bid five 2's pushing one 1 and two 2's and
           the previous bid was two 1's and
           there are five dice under my cup, with one 1 and two 2's and
           the other player has only a single unknown die
        then create an operator preference of 0.12
```

Figure 3.9: Chunking learns reinforcement learning value functions that incorporate domain knowledge.

In another recent project that shows how chunking can compile knowledge into a form that another

[4]The game, which includes the aforementioned Soar agent, is available as a multi-player game on the App Store.

learning algorithm can use, Zhong et al. (2012) used Soar agents to populate a city traffic simulator to explore how variations in driver traffic law compliance affect traffic patterns. Each agent modeled a driver's decision-making identically, differing only by how they evaluate the expected utility of a route. As the agents explore the city, they learn chunks that evaluate the desirability of a route, which vary based on the agent's compliance level; these chunks allow each agent to choose the route with the highest expected utility, which would then be tuned by reinforcement learning using rewards that were based on how fast the route turned out to be. The resulting simulation was then analyzed to explore traffic phenomenon that the researchers were interested in.

Chunking achieves these results with little computational overhead. Because of its commitment to learning from an instantiated trace of behavior using a simple generalization mechanism that does not require analysis of the underlying reasoning, chunking learns rules quite efficiently. Previous evaluations have shown that the computational cost of enabling chunking in every subgoal increases execution time by 20-30 percent in agents that use subgoals to decompose problems. And after an agent has built chunks, reactivity can even improve because the agent no longer performs as much deliberate reasoning. For example, Figure 3.10 shows that the reactivity of the Dice agent is minimally affected during learning and improved after learning.

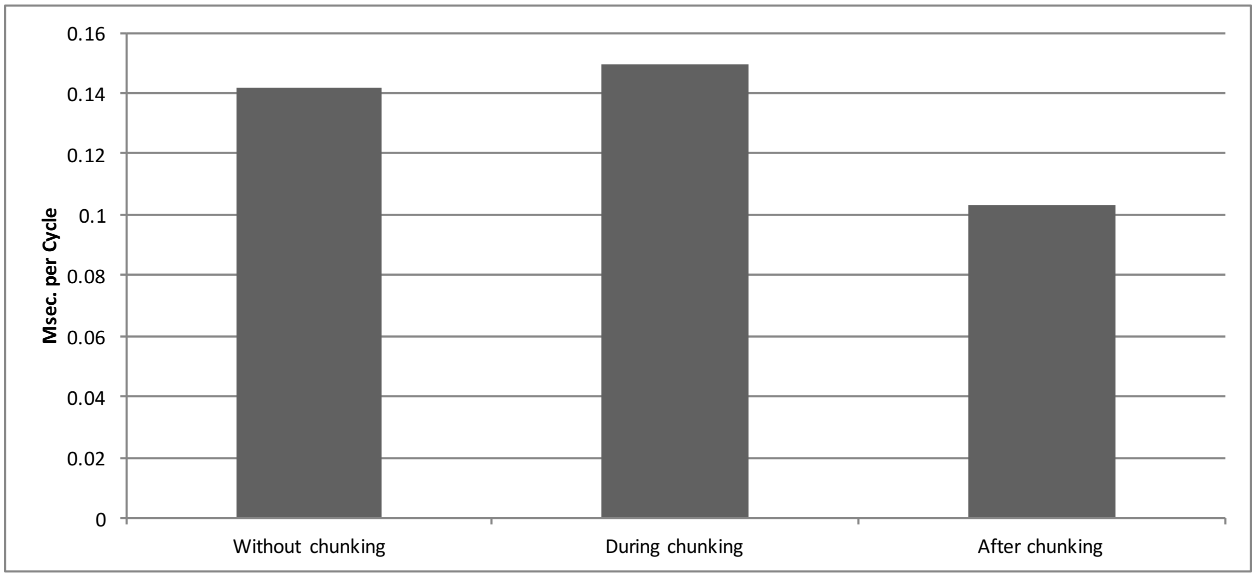


Figure 3.10: Reactivity of agent performing graph search is not hindered by chunking.

Despite its efficiency and potential to provide computational benefits, chunking is nonetheless seldom used outside of research on procedural learning. To understand the reasons why, we must first understand some key limitations of chunking and their real-world implications. Section 3.5 discusses situations where chunking can learn incorrect rules and presents a taxonomy of 12 types of reasoning that can lead to incorrect chunks. Section 3.6.1 explores why chunking has problems learning optimally general rules from some types of reasoning. Finally, Section 3.7 describes why chunking cannot be used by current research projects that are building agents who learn using higher-level, intentional strategies, like learning by instruction.

3.4 Relationship to Explanation-Based Learning

We postpone exploring chunking's limitations to briefly discuss some key similarities and differences between a behavioral summarization algorithm like chunking and classic explanation-based learning (EBL) techniques, since the two have often been compared in past literature.

Classic explanation-based learning algorithms use a single training example of a goal concept to create a rule that can be applied in one step to determine if another instance matches the goal concept (Mitchell, 1983). They do this by using domain knowledge to generate an explanation of why that instance matches the goal concept and then generalizing that explanation to learn a rule. Most EBL systems learn by performing the following four steps:

1. **Explanation Generation**: EBL systems typically have a proof generation or planning component that uses domain knowledge to create a causal model, or explanation, for why an instance is an example of a goal concept.

2. **Pruning**: EBL traverses the explanation and removes non-operational aspects of the explanation, which are concepts in trace that cannot be efficiently used to recognize instances of the concept it denotes (Keller, 1988). One challenge EBL systems face is that definitions of operationality can vary from system to system and may be expensive to determine. How operationality is defined in an EBL system is a critical question that has significant implications in terms of whether the rule will improve the agent's performance (Gratch and Gerald, 1991). For example, some of the challenges of operational pruning can be seen in Mooney's GENESIS system (Mooney, 1988), which used EBL to learn schemata for achieving goals in a narrative. Mooney's system, which learned from a training example that describes a specific example story of people carrying out plans, performed three different forms of non-operational pruning:

    a Remove parts of the network that do not causally support the achievement of the main thematic goal;

    b Remove nominal instantiations of known schemata;

    c Remove actions and states that only support inferences to more abstract actions or states.

3. **Generalization**: EBL creates a generalized explanation by determining the weakest set of preconditions that supports the pruned explanation. To do this, EBL analyzing the structure of the rules in the explanation and how they interconnect using variable unification algorithms, for example variable regression (Waldinger, 1977) or EGGS (Mooney et al., 1986).

4. **Rule Formation**: EBL uses the generalized explanation to create a rule that can identify instances of the concept.

Chunking is also an analytical, one-shot, deductive learning algorithm that learns by generalizing a causal explanation. Clearly, there are some key similarities. But, while chunking is an explanation-based approach, it was developed independently of EBL algorithms and has some significant differences. The original motivation for chunking was to model the power law of learning (Newell and Rosenbloom, 1981) and was inspired by a psychological theory of memory in which a chunk is a symbol that designates a pattern of other symbols (Miller, 1956). If we compare the four steps of EBL, we find that there are some similarities but many differences; only step 4, rule formation, is the same:

1. **Explanation Generation**: This does not exist as an explicit step in chunking. Because chunking's goal is fundamentally different – it is attempting to summarize behavior generated in a subgoal – it does not need to generate explanations and simply analyzes the historical record of all rules that fired in a subgoal. With chunking, the explanation generation step is one of the least expensive steps. This contrasts with typical EBL systems, where explanation generation is the most expensive step of the learning process because it typically performs some form of search through the entire space of possible explanations to generate one or more from which to learn rules from. Moreover, the explanations that the two systems generate, a behavior trace and a logical proof, have differences that lead to important implications which are discussed below.

2. **Pruning**: While chunking does traverse the explanation, it does not try to determine anything akin to operationality and instead performs the straightforward task of analyzing knowledge dependencies. In contrast to the types of operationality pruning that an EBL system performs, chunking's dependency analysis chooses the aspects of an explanation to include in the learned rule using a simple, task-agnostic, unambiguous mechanism: it simply collects all nodes in the explanation that matched supergoal knowledge.

3. **Generalization**: Chunking does not determine the weakest set of preconditions like EBL does; instead, it uses a Soar-specific, heuristic-based generalization mechanism.

4. **Rule Formation**: This step is roughly equivalent.

We have found that learning from a behavior trace introduces challenges not found when learning from an explanation generated by the types of planners or theorem-provers that EBL systems typically employ. The most problematic aspect is that a behavior trace is only an approximation of all of the processing that occurred in a cognitive architecture and is not guaranteed to be a complete explanation of the agents reasoning. In contrast, the trace used in an EBL system is built specifically to explain the training instance and always justifies all knowledge structures. In a cognitive architecture, there are other processes that occur during agent reasoning that may not be reflected in a trace of how rules changed working memory. For example, a behavior trace does not explain what knowledge was used to choose amongst candidate operators or whether a probabilistic decision was made when choosing an operator. If a behavior trace does not provide a complete explanation, an online procedural learning algorithm will be unaware of necessary dependencies, which can then lead to overgeneral, incorrect rules. In fact, half of chunking's correctness issues are the result of such missing dependencies. To make things more complicated, it's also possible for a knowledge structure to have multiple rules that support it. This too can lead to learning incorrect rules or at least rules that are not optimally general. A complete list of the sources of incomplete explanations are discussed in Section 3.5.

Another problematic difference is that a cognitive architecture often provides rules with powerful, dynamic capabilities that are not found in the rules that EBL uses to generate its proofs. For example, rules can contain actions which perform arbitrary functions (math calculations, string manipulation, random number generation, etc.), which are effectively black boxes to the online procedural learning algorithm. An online procedural learning algorithm that attempts to summarize a trace with reasoning that includes such rules can learn an incorrect rule or at least have difficulty learning an optimally general rule.

There are also several differences between a behavior trace and a proof that impact efficiency. For example, because there is no limit on the amount of processing that can occur in a subgoal or the amount of time that a subgoal can exist, a behavior trace can be arbitrarily large. The behavior trace previously used as an example is intentionally simple. Figure 5.7 shows a more complex trace from an agent that solves the Mouse and Cats problem. Despite being a far more complex trace than the water jug example from Section 3.2, the behavior described is still from an agent reasoning about an easily-solvable toy problem. Real-world agents can have far more complex traces; to give a feel for the amount of reasoning involved in such agents, a trace of a recent natural language understanding agent (Lindes et al., 2017) is so large that a visualization of one particular subgoal covered 16 pages and was barely readable. We leave the efficiency-related differences for Section 5.3, where we present a novel unification algorithm that can handle the challenges of those differences.

3.5 A Taxonomy of Chunking's Correctness Issues

One significant limitation of Soar's chunking algorithm is that it cannot correctly summarize certain types of problem solving. In other words, chunking can learn incorrect rules that do not always produce the same inferences that the original subgoal would have generated in the same situation. Correctness issues are numerous and particularly difficult to explicate because they have disparate underlying issues, many of which require a deep, technical understanding of how Soar works to understand. To date, no systematic exploration of correctness issues has been performed. We present a first attempt in this work.

We use the four necessary conditions for correct behavior summary to organize Soar's correctness issues into a taxonomy. Figure 3.11 shows 12 types of reasoning that can lead to incorrect rules in Soar's chunking algorithm. A category for reasoning that is not logically consistent or not contemporaneously possible is not included because no correctness issues currently exist for that requirement. Previous research work eliminated that problem in Soar by adding a consistency enforcement mechanism to universal subgoaling (Wray and Laird, 2003).

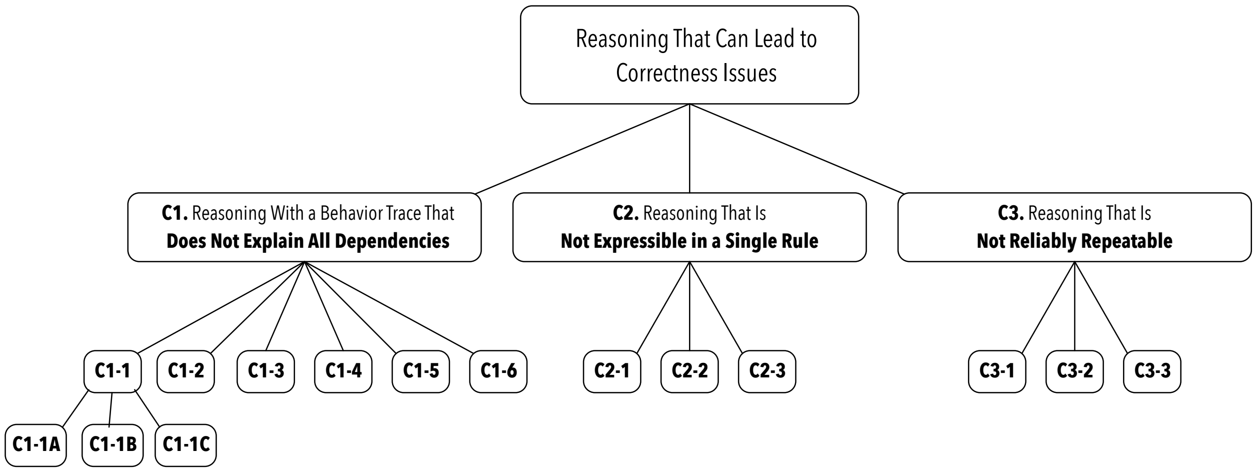

Reasoning That Can Lead to Correctness Issues
C1. Reasoning With a Behavior Trace That Does Not Explain All Dependencies
C2. Reasoning That Is Not Expressible in a Single Rule
C3. Reasoning That Is Not Reliably Repeatable
C1-1
C1-2
C1-3
C1-4
C1-5
C1-6
C2-1
C2-2
C2-3
C3-1
C3-2
C3-3
C1-1A
C1-1B
C1-1C

Figure 3.11: Overview of taxonomy of reasoning that chunking cannot always correctly summarize.

While some issues are specific to the Soar architecture – we will discuss which ones are and which ones are not – the requirements and categories are architecture agnostic and can help other architectures that attempt online procedural learning determine why their problems are occurring. Moreover, how we approach the individual problems in each category offers lessons on how this work can be translated to other cognitive architectures. Specifically, these categories share commonality in their mitigation strategies. Figure 3.12 provides a preview of how the 12 correctness and 2 generality issues map to the five mitigation strategies that our approach uses, which are described in Chapter 4.

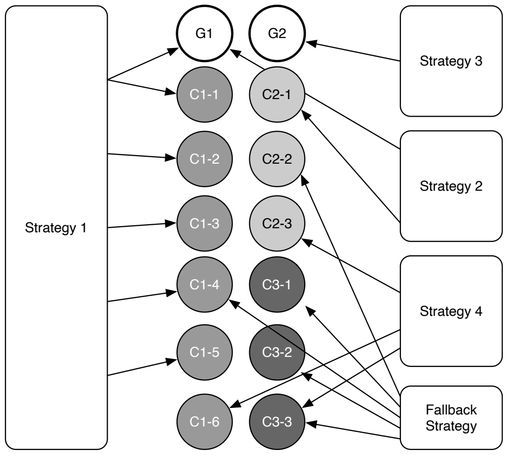
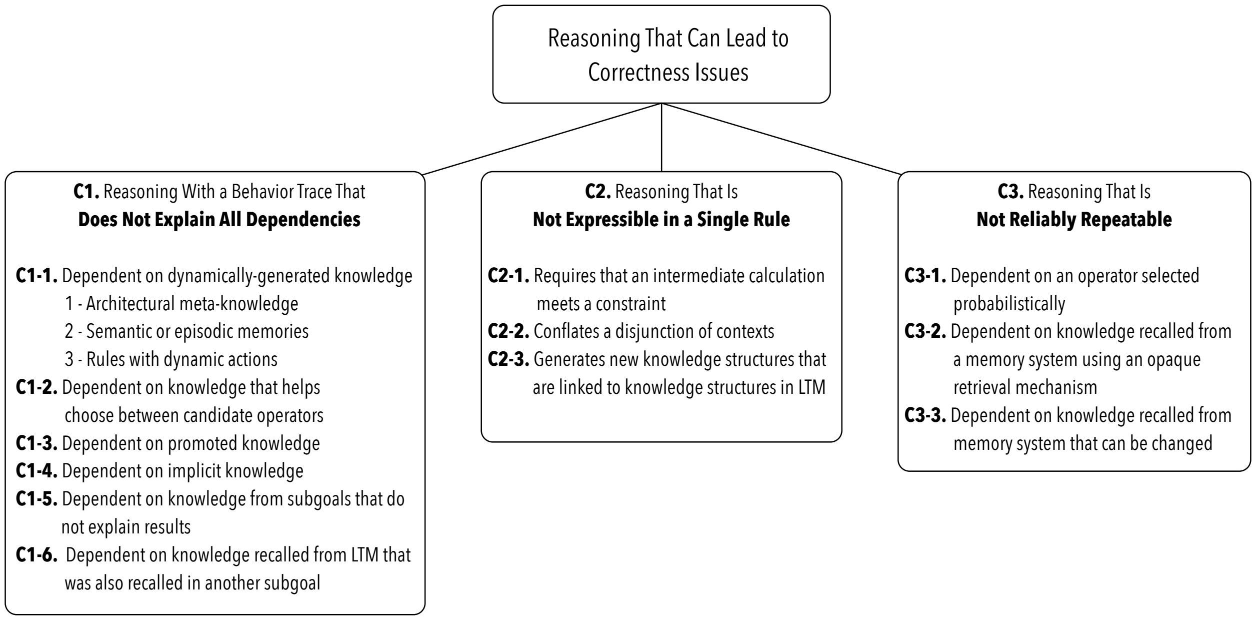


Figure 3.12: A preview of how various correctness issues share common mitigation strategies.



Reasoning That Can Lead to Correctness Issues

C1. Reasoning With a Behavior Trace That **Does Not Explain All Dependencies**

C1-1. Dependent on dynamically-generated knowledge
1 - Architectural meta-knowledge
2 - Semantic or episodic memories
3 - Rules with dynamic actions
C1-2. Dependent on knowledge that helps choose between candidate operators
C1-3. Dependent on promoted knowledge
C1-4. Dependent on implicit knowledge
C1-5. Dependent on knowledge from subgoals that do not explain results
C1-6. Dependent on knowledge recalled from LTM that was also recalled in another subgoal

C2. Reasoning That Is **Not Expressible in a Single Rule**

C2-1. Requires that an intermediate calculation meets a constraint
C2-2. Conflates a disjunction of contexts
C2-3. Generates new knowledge structures that are linked to knowledge structures in LTM

C3. Reasoning That Is **Not Reliably Repeatable**

C3-1. Dependent on an operator selected probabilistically
C3-2. Dependent on knowledge recalled from a memory system using an opaque retrieval mechanism
C3-3. Dependent on knowledge recalled from memory system that can be changed

Figure 3.13: A taxonomy of reasoning that chunking cannot always correctly summarize.

Figure 3.13 shows a more complete picture of our correctness issue taxonomy. While we believe this list is exhaustive, please note that historically many of these correctness issues arose when Soar gained additional capabilities; in earlier, simpler versions of Soar, many of the specific issues did not exist. While we currently cannot conceive of other correctness issues, we also cannot rule out them out.

Although the twelve correctness issues described may seem like a debilitating number of problems, it should be noted that, in practice, the problems can often be managed. In some cases, correctness issues can be avoided by altering the agent's reasoning. For example, for issue C2-1, summarizing reasoning that requires that an intermediate calculation meets a constraint, the problematic rules that involve mathematical calculations can be modified so that the intermediate result is stored in the supergoal rather than the subgoal. With this alteration, the agent learns two chunks, both of which are correct. Moreover, some correctness issues do not actually produce incorrect behavior in practice. For example, knowledge sources that are considered unreliable, like semantic memory, can safely be used if the designer knows that the queries the agent performs will always retrieve the same knowledge structures in the future.

Despite the existence of work-arounds, agent engineers cannot be confident that their agents will learn correct rules unless they have a high-degree of Soar expertise; they must know when each correctness issue can occur, how each issue would manifest itself during execution and how to modify their agent's reasoning to work around the issues. The chance for correctness-related task failure coupled with the level of expertise needed to avoid the underlying issues has caused Soar's online procedural learning capabilities to seem risky and not worth the effort for many projects.

In any case, many of the correctness issues in Figure 3.13 require a deeper understanding of Soar and all of its various components than we can assume the reader has, so we expect that they may be challenging to unpack at this point. To avoid overwhelming the reader, we move on to generality. We return to describe Soar's twelve correctness issues and the mechanisms used to mitigate in Chapter 6.

3.6 Problems with Generality

Chunking's second problem is that it only learns optimally general rules for some types of agent reasoning. When chunking was first developed, the hypothesis was that using an instantiated trace of agent behavior was sufficient to create a general learning algorithm that was both highly efficient and able to provide computational benefits to an intelligent agent. To some degree, it proved true. The computational overhead of building new rules is low and chunking does indeed learn rules that improve performance for some agents. But for many agents that utilize a fairly common type of reasoning, chunking's early commitments prevent it from learning general rules that provide computational benefits.

The underlying reason for this Soar-specific limitation is that chunking cannot capture the generality of rules that reason about the specific values of features and/or the relationships between the values of features (Rosenbloom and Laird, 1986; Zerr and Ganascia, 1991). For example, consider the trivially simple example depicted in Figure 3.14, in which chunking must summarize a single rule which tests whether the value of an object's `size` feature in working memory is greater than five.

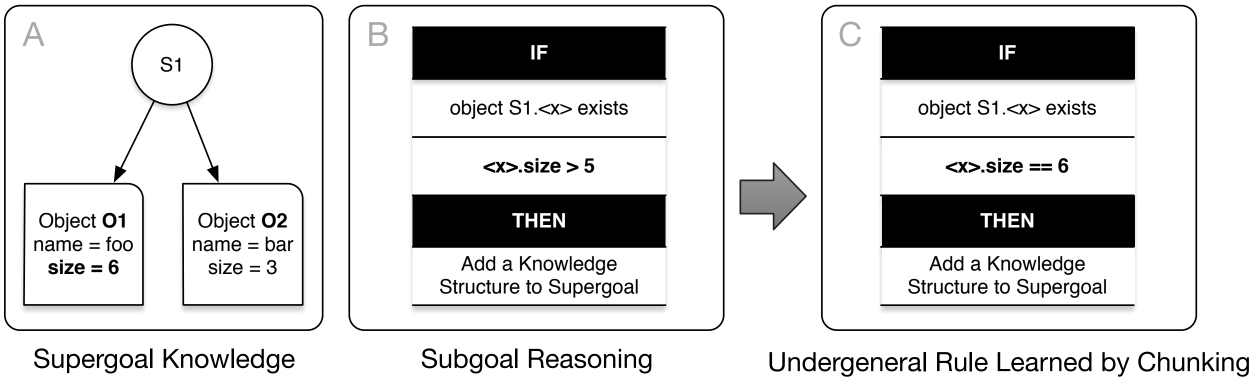


Figure 3.14: Trivial example in which chunking should learn the same rule as it is summarizing.

If chunking is summarizing a single rule firing that only tests supergoal knowledge, for example the rule in pane B of Figure 3.14, it should learn the exact same rule that it was given to analyze. In this case, it does not because it cannot capture the generality in the original rule; instead, as shown in pane C, it learns an undergeneral rule that only applies when that feature has the specific value that existed when the rule was learned, which, in this case, is six. (If the agent should later encounter other objects with different sized that are also greater than 5, it will learn additional undergeneral chunks for each new value.)

In practice, the degree of undergenerality varies depending on how the agent represents its knowledge and the type of reasoning the agent performs. To provide a real-world example, we can examine a classic Soar demo agent that solves arithmetic problems. This is an agent that reasons about feature values and was not specially engineered to learn optimally general rules with chunking, so it should show some signs of undergenerality. If we task this agent with solving 1000 multi-column addition and subtraction problems using both chunking and the new online procedural learning algorithm described later in this paper, we see a stark difference in the generality of rules learned. In Figure 3.15, pane A represents the space of situations in which a set of `process-column` rules can perform subtraction on the digits in one particular column. Since these rules match in a subgoal and return the calculation to the supergoal, chunking attempts to learn a new rule every time the `process-column` rules perform their calculations. After each agent solves the same 1000 problems, we examine how many chunks each agent learned using the `process-column` rules. As shown in pane B, our new system learns 8 optimally general rules, after which all deliberate `process-column` reasoning is replaced by chunks firing. In contrast, as shown in pane C, chunking learns 1263 undergeneral rules based on the `process-column` reasoning before it is able to cover the same space of situations. (Samples of the actual rules learned by the two systems can be compared in Figure 7.18 of Chapter 7.)

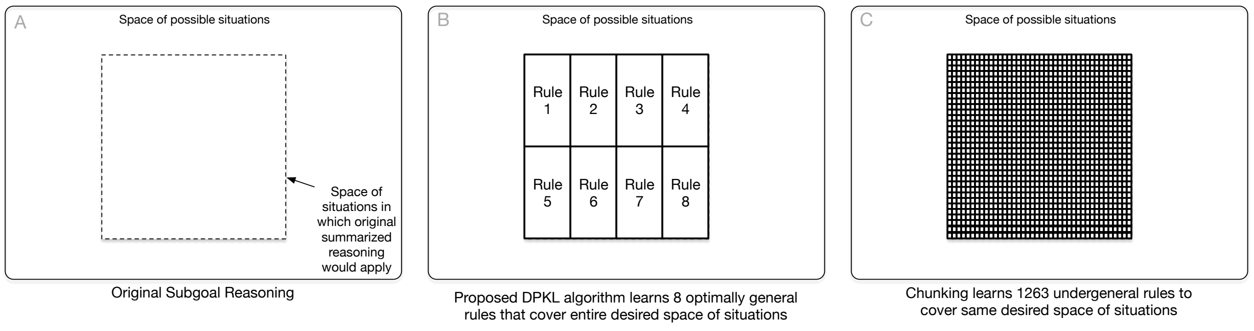
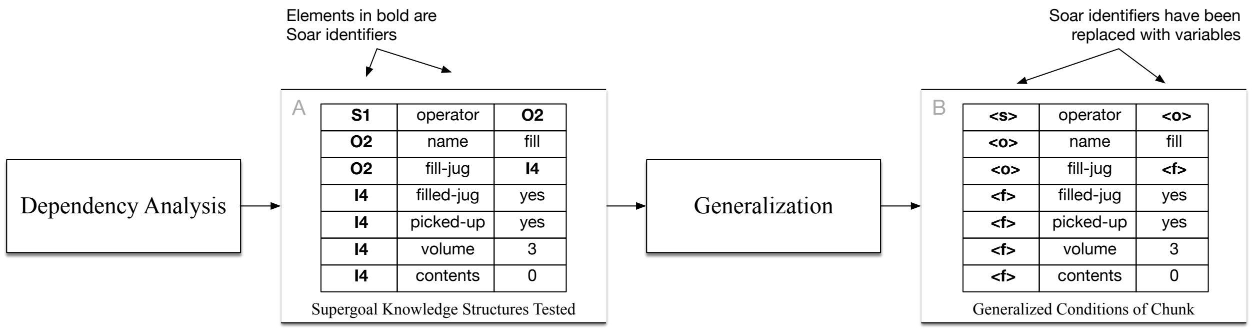

A
Space of possible situations
Space of situations in which original summarized reasoning would apply
Original Subgoal Reasoning
B
Space of possible situations
Rule 1
Rule 2
Rule 3
Rule 4
Rule 5
Rule 6
Rule 7
Rule 8
Proposed DPKL algorithm learns 8 optimally general rules that cover entire desired space of situations
C
Space of possible situations
Chunking learns 1263 undergeneral rules to cover same desired space of situations

Figure 3.15: Undergenerality in an agent that performs arithmetic.

Reasoning about feature values and/or the relationships between them is common in many domains and agent designs, which means that it is also common that chunking learns undergeneral rules.

3.6.1 Problems Capturing Generality of Underlying Reasoning

Chunking cannot capture all of the generality underlying an agent's reasoning because, frankly, it doesn't try to analyze the underlying reasoning. Instead, it uses a heuristic that only applies to aspects represented that it knows are always reasoned about generally: Soar identifiers.[5] To explain how it works, we briefly revisit the example from Section 3.2, in which we described how a chunk is learned in an agent that solves the water jug problem. Specifically, we return to the point immediately after the dependency analysis component has traversed the behavior trace and collected all matched knowledge structures that were accessible to the supergoal. The list of WMEs collected, as shown in pane A of Figure 3.16, is a description of the context that was necessary for that problem solving to occur.


Elements in bold are Soar identifiers
Soar identifiers have been replaced with variables
Dependency Analysis
A
Generalization
B
Supergoal Knowledge Structures Tested
Generalized Conditions of Chunk

S1	operator	**O2**
O2	name	fill
O2	fill-jug	**I4**
I4	filled-jug	yes
I4	picked-up	yes
I4	volume	3
I4	contents	0

<s>	operator	**<o>**
<o>	name	fill
<o>	fill-jug	**<f>**
<f>	filled-jug	yes
<f>	picked-up	yes
<f>	volume	3
<f>	contents	0

Figure 3.16: An example of heuristic-based variablization.

[5]Soar identifiers are transient symbols generated at run-time, so rules cannot reference them directly and must use variables to reason about the objects they represent. This means chunking can always safely assume that elements that match Soar identifiers were reasoned about generally.

At this point, the context description is a list of specific working memory values and can only match the same situation that was learned from. To allow the context described by the rule to apply to more than the same situation, chunking generalizes the conditions by replacing some of the matched values with variables, a process in Soar called variablization. To do this, it uses a heuristic that does not require any analysis of the original, underlying rules that matched. This heuristic replaces all Soar identifiers (SI), which again are special architecturally-assigned symbol that are used to organize sets of features into objects, with corresponding variables.[6] In pane A of Figure 3.16, all of the Soar identifiers, denoted in bold, are replaced with variables, as shown in pane B.

Since the heuristic-based variablization mechanism cannot be used on any aspect of the problem-solving context that didn't match a SI, Soar cannot capture the generality of any reasoning that matches features of the context with specific values. As a result, chunking can learn undergeneral rules that require the exact features that existed when the original learning occurred.

3.6.2 Mitigating Problems with Generality

While problems from lack of generality may seem widespread as described, one of the reasons that this issue has not been addressed is that undergenerality can sometimes be mitigated by modifying an agent's representation and reasoning so that chunking can learn optimally general rules. In this section, we will demonstrate how this can be done by revisiting the example from Figure 3.14 and re-engineering its reasoning so that chunking can learn an optimally general rule. We will then discuss when such mitigation strategies are impossible or become impractical.

In our original example, as shown again in panes A and B of Figure 3.17, chunking is summarizing a single rule firing and should ideally learn an identical rule that applies whenever an object has a size feature greater than 5. As previously discussed, chunking instead learns a rule that applies only when objects have a size of 6, the specific value encountered when learning.

To allow it to apply whenever the original reasoning would have, one could "move" that feature test *out of an element that chunking can't capture the generality of*, namely a numerical feature, *into something it can capture the generality of*, namely a Soar identifier. Pane C of the figure shows how two modifications can be made to our original example so that chunking does that and

[6]At this point in the discussion, it may not be easy to understand why this heuristic variablization mechanism works, but for those who are curious, there are two reasons. Because SIs are transient and not allowed in any rules, we know that any rule that matched an SI did so by using a variable, so chunking knows any SI element in the context description must also be replaced with a variable. And because all references to an SI are guaranteed to point to the same structure in working memory, chunking knows that systematically replacing all instances of an SI with the same corresponding variable cannot introduce overgenerality.

is indeed able to summarize the reasoning at the same level of generality as the original reasoning. The first modification is the addition of a second rule that automatically annotates every object with a size greater than 5 with a new feature whose label is "MoreThanFive" and whose value is *a newly created Soar identifier*. The second modification alters the conditions of the original rule to test for this new Soar identifier feature rather than the original numeric one; this "moves the reasoning" into an aspect of the representation that chunking can generalize. With these two changes, chunking learns the rule described in pane D, which applies in all of the same situations as the original rule.

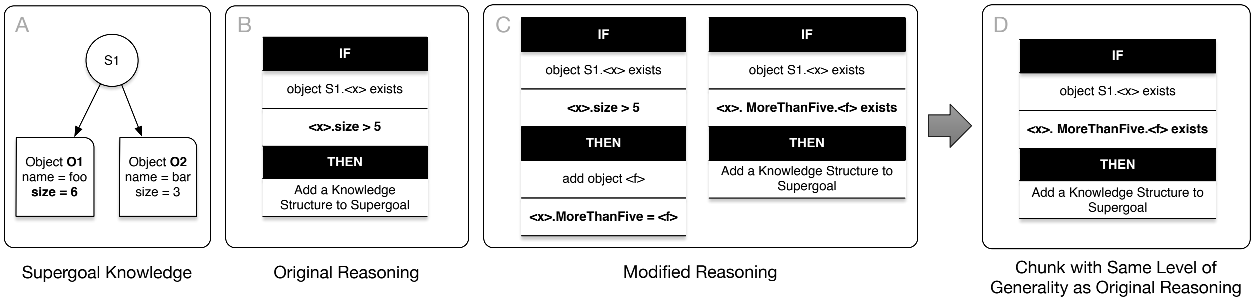


Figure 3.17: Previous example re-engineered so that chunking can learn an optimally general rule.

For some agent reasoning, tricks like this can work. For many agents, though, this approach is impractical, since it doesn't scale well if the agent needs to reason about a large set of features or complex combinations of features. And for some agents, this workaround cannot be used at all, such as in the case where an agent that does not know a priori what combination of features it will need to reason about.

Most importantly, though, this workaround cannot be used to capture the generality of reasoning that actually does something meaningful with the feature values, for example performing a calculation or simply adding it to another knowledge structure. To demonstrate this, Figure 3.18 describes how the workaround cannot be used if the agent attempts to return one of the tested values as a result. In the original example, the agent creates a WME in the supergoal that consisted completely of literal elements and was not dependent on any of the WMEs tested in the supergoal; in the new example, the agent instead creates a result based on the specific value of the feature that matched. Because chunking cannot generalize elements of conditions that test feature values, it cannot capture the generality of the agent's reasoning in this case. Consequently, chunking learns the undergeneral rule described in pane D.

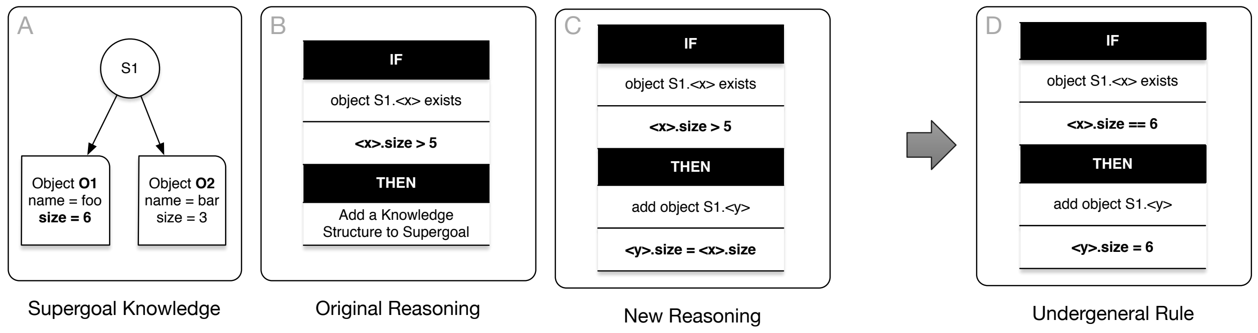


Figure 3.18: Reformulated example that cannot be re-engineered easily because a rule action copies a feature value.

3.7 Problems Supporting Higher-Level, Deliberate Learning Strategies

The deficiencies described above place limits on which agents can use chunking to learn new procedural knowledge. Obviously, any agent that cannot change its reasoning to avoid correctness issues would not be able to use chunking. Similarly, any agent that reasons about values of features but does not benefit from undergeneral rules would not use chunking. Until the generality and robustness of Soar's online procedural learning algorithm is improved, these agents must always re-deliberate.

An interesting class of learning agents that face significant difficulties using chunking's capabilities are agents that learn by instruction. There are currently several projects that are pursuing this avenue of research. For example, the ROSIE group at the University of Michigan is exploring interactive task learning (ITL) by building robots that learn procedural knowledge based on natural language interaction with a human (Mininger and Laird, 2016; Kirk et al., 2016; Lindes et al., 2017; Kirk and Laird, 2016). (You may remember that ITL and ROSIE were used as an early example in Section 1.1.) In some sense, these agents are attempting to use online procedural learning within ITL to achieve what DeJong theorized could improve a similar problem in classical explanation-based learning systems:

> "Another interesting direction is to explore a human in the learning loop. In this regard, a human teacher might collaboratively help to build or to choose explanations. Human interaction has proven to be effective in theorem proving, which is similar to explanation building. Alternatively, the human might select or suggest elements for (the explanation)." (Dejong, 2014)

ITL agents have not been able to fully utilize chunking as the basis for interactive task learning for three main reasons:

- ITL agents cannot tolerate undergenerality. Other agents can compensate for undergeneral rules with extra training, i.e. they learn multiple rules that are equivalent to an optimally general rule. Agents that learn by instruction cannot repeatedly ask a human the same question. So, these agents need to learn optimally general rules from a single instruction (Laird et al., 2017), a capability chunking cannot currently provide.

- ITL agents utilize many of the types of reasoning that leads chunking to learn incorrect rules. In terms of our correctness taxonomy, our current ITL agents utilize nearly every type of problematic reasoning we have described: C1-1 (1-3), C1-3, C1-4, C1-5, C1-6 C2-1, C2-2, C3-2 and C3-3. The most problematic issue for these agents is C1-6, which occurs as result of unexpected interactions between different subgoals that retrieve the same long-term semantic memories. For these agents to use chunking as the basis for their higher-level, deliberate learning strategies, many of chunking's correctness issues would need to be ameliorated.

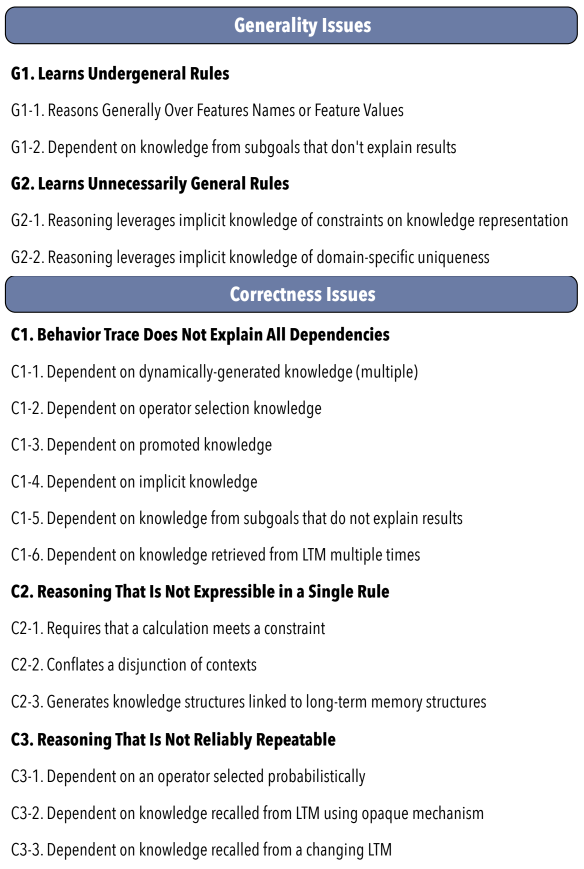
- ITL agents must learn rules that encode the knowledge acquired from the instruction without being directly dependent on a declarative representation of the instructor's utterance. (A rule that was dependent on the words of the instructor would have little utility since those instructions are only available during the interaction with the instructor.) Previous Soar systems that learned by instructions would force learned knowledge to be independent of declarative instructions by using a technique that takes advantage of one of chunking's correctness issues that leads to overgeneral rules: C1-2, reasoning that is dependent on knowledge that helps choose between candidate operators (Huffman and Laird, 1995). Specifically, those systems would put tests of declarative instructions in operator selection rules that chunking ignores, thus producing rules that do not rely on the original instructions. Regardless of whether our proposed system successfully mitigates C1-2, which it does do, ITL agents should have a formal mechanism that allows them to make their subgoal reasoning cleanly independent of the instruction.

Until the generality and robustness of Soar's online procedural learning algorithm is improved, ITL agents will have difficulty incorporating what they learn from human instructions into their procedural knowledge in a way that is general and will transfer to similar problems.

3.8 Conclusions

In summary, compiling an agent's deliberate problem solving into efficient, reactive, task-specific knowledge is not simply a matter of turning on chunking. For the agents that can use chunking, a multitude of reasons, summarized in Figure 3.19, exist that can result in undergeneral or incorrect rules.

Because Soar's ability to achieve general and correct online procedural learning is dependent on having representations and processing that accommodate its limitations, agent engineers need to either carefully design their agents to mitigate potential problems or be willing to accept the risk that their agents learn undergeneral rules that do not provide computational benefits or incorrect rules that can hinder/prevent task execution. We argue that this requirement coupled with the fact that chunking could not be used at all for some important types of reasoning is why online procedural learning systems have never reached their potential and are seldom utilized, as reflected in the following:


Generality Issues

G1. Learns Undergeneral Rules

G1-1. Reasons Generally Over Features Names or Feature Values

G1-2. Dependent on knowledge from subgoals that don't explain results

G2. Learns Unnecessarily General Rules

G2-1. Reasoning leverages implicit knowledge of constraints on knowledge representation

G2-2. Reasoning leverages implicit knowledge of domain-specific uniqueness

Correctness Issues

C1. Behavior Trace Does Not Explain All Dependencies

C1-1. Dependent on dynamically-generated knowledge (multiple)

C1-2. Dependent on operator selection knowledge

C1-3. Dependent on promoted knowledge

C1-4. Dependent on implicit knowledge

C1-5. Dependent on knowledge from subgoals that do not explain results

C1-6. Dependent on knowledge retrieved from LTM multiple times

C2. Reasoning That Is Not Expressible in a Single Rule

C2-1. Requires that a calculation meets a constraint

C2-2. Conflates a disjunction of contexts

C2-3. Generates knowledge structures linked to long-term memory structures

C3. Reasoning That Is Not Reliably Repeatable

C3-1. Dependent on an operator selected probabilistically

C3-2. Dependent on knowledge recalled from LTM using opaque mechanism

C3-3. Dependent on knowledge recalled from a changing LTM

Figure 3.19: Situations in which chunking can learn undergeneral, unnecessarily general or incorrect rules.

> "In order to prevent these problems, the authors of SOAR propose that the use of restriction tests on variables and the use of evaluation instructions be avoided. This restriction is considerable, since it reduces the programming language to a much impoverished minimal subset, which is unlikely to be used in industry. The generality of rules is controlled by means of the data representation. [...] Chunking is correct only if considered as a process for memorizing results. When asked to learn more general rules, the loss of variable bindings or restriction tests leads to inconsistencies in the knowledge base. We may prevent these weaknesses by creating intermediate subgoals or using a heavier data representation. Unfortunately, in using a heavier data representation, more conditions are included in production rules, whose pattern matching time will increase immediately. This affects system performance and knowledge base

legibility.” (Zerr and Ganascia, 1991)

Now that the nature of Soar’s online procedural learning deficiencies have been discussed, Chapter 4 presents a comprehensive strategy to address those deficiencies and achieve the qualities described by our desiderata in Section 2.6.

CHAPTER 4

Our Approach to Online Procedural Knowledge Learning

In this chapter, we discuss our approach to mitigating the deficiencies we've described to provide the capabilities necessary for an efficient online procedural learning algorithm that is robust to correctness issues and can learn optimally general rules. We name this approach *explanation-based behavior summarization* (EBBS).[1]

As we alluded to in Section 3.5, EBBS uses five strategies to mitigate correctness and generality issues.

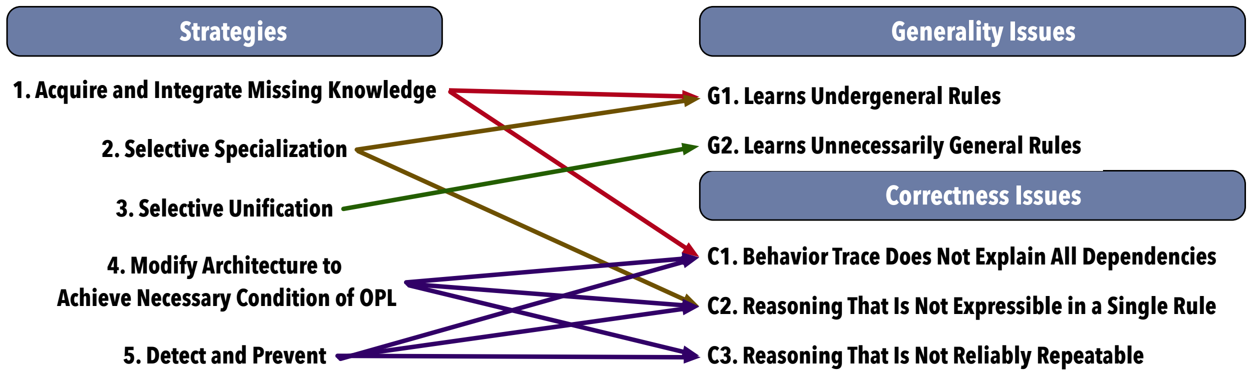

Strategies
1. Acquire and Integrate Missing Knowledge
2. Selective Specialization
3. Selective Unification
4. Modify Architecture to Achieve Necessary Condition of OPL
5. Detect and Prevent
Generality Issues
G1. Learns Undergeneral Rules
G2. Learns Unnecessarily General Rules
Correctness Issues
C1. Behavior Trace Does Not Explain All Dependencies
C2. Reasoning That Is Not Expressible in a Single Rule
C3. Reasoning That Is Not Reliably Repeatable

Figure 4.1: The Five Strategies of EBBS.

The five strategies, which will be explained in the next few sections, are as follows:[2]

1. Acquire and integrate missing knowledge into the online procedural learning algorithm (7)

[1]We include explanation-based because our new generalization algorithm performs analysis similar to that found in the generalization algorithms of classic EBL systems. We use the term behavior summarization to highlight how learning correct rules from an imperfect approximation of an agent's reasoning over time poses challenges that learning a rule from a proof does not.

[2]The number in parentheses indicate how many correctness and generality issues a strategy helps mitigate.

2. Selectively specialize some elements of learned rules (2)

3. Selectively unify some elements of learned rules (2)

4. Modify the architecture to guarantee that the assumptions the online procedural learning algorithm makes are met (3)

5. Detect learning scenarios that can result in incorrect rules and avoid learning rules in those scenarios (5)

4.1 Strategy 1: Acquire and Integrate Missing Knowledge

Our diagnosis is that the main reason for many of the problems that chunking has summarizing problem-solving is that it does not capture and utilize all of the knowledge available about the problem-solving that occurs in the subgoal. The importance of implementing this strategy is shown in Figure 4.2, which lists the 7 issues it mitigates from our 14.

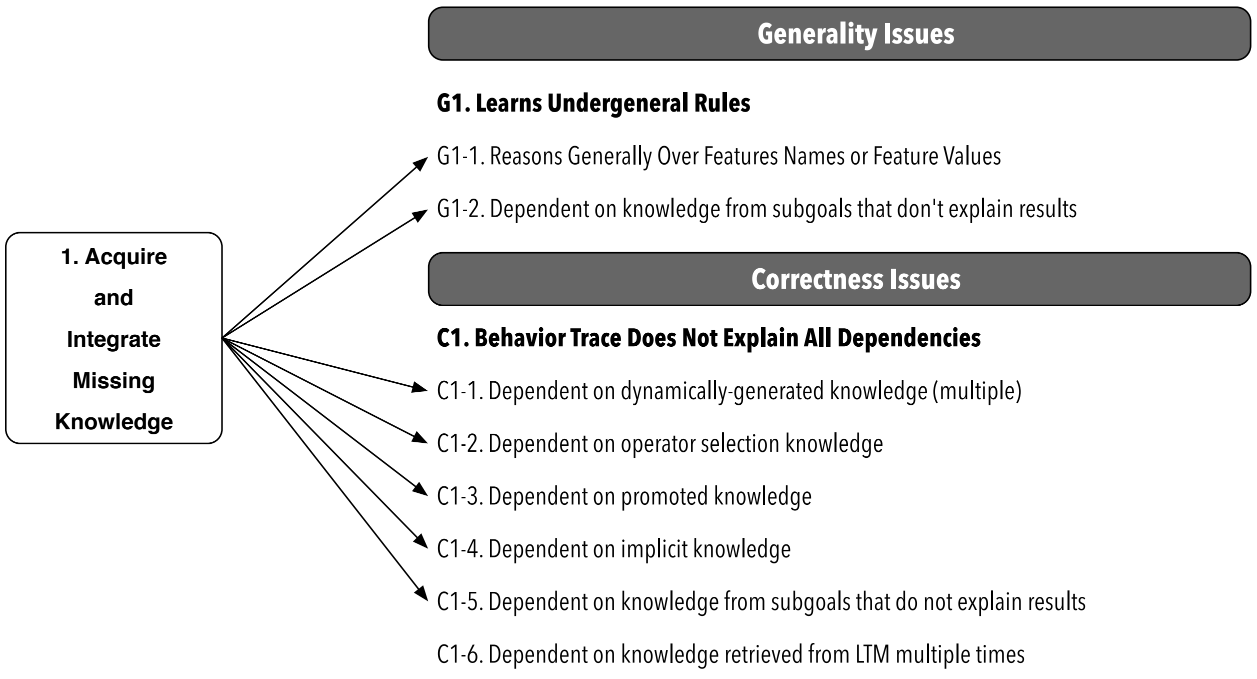
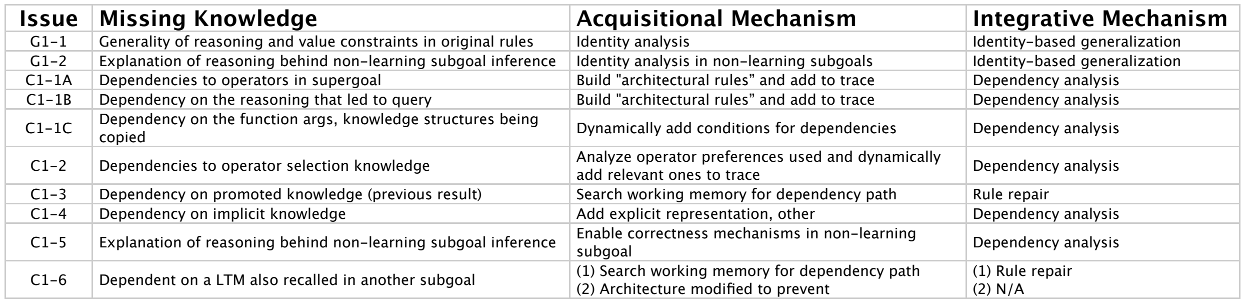

Generality Issues
G1. Learns Undergeneral Rules
G1-1. Reasons Generally Over Features Names or Feature Values
G1-2. Dependent on knowledge from subgoals that don't explain results
1. Acquire and Integrate Missing Knowledge
Correctness Issues
C1. Behavior Trace Does Not Explain All Dependencies
C1-1. Dependent on dynamically-generated knowledge (multiple)
C1-2. Dependent on operator selection knowledge
C1-3. Dependent on promoted knowledge
C1-4. Dependent on implicit knowledge
C1-5. Dependent on knowledge from subgoals that do not explain results
C1-6. Dependent on knowledge retrieved from LTM multiple times

Figure 4.2: Correctness issues mitigated by strategy 1.

The obvious general solution to this problem is to find a way to integrate that missing knowledge into the online procedural learning algorithm. To this end, this strategy provides both *acquisitional mechanisms* to describe missing knowledge and *integrative mechanisms* to incorporate it into the algorithm's analysis. Specifically, the acquisitional mechanisms build data structures that describe the missing knowledge. For some issues, this is straightforward and simply requires that the information is encoded in a way that the online procedural learning algorithm can use. For other issues, though, the acquisitional mechanism performs specialized analysis of the agent's problem solving, for example searching through working memory or analyzing reasoning in other subgoals. After the missing knowledge is acquired, the integrative mechanisms provide ways to incorporate it into the online procedural learning process. Figure 4.3, provides a preview of the acquisitional and integrative mechanisms that our new online procedural learning algorithm uses to address undergenerality and six different correctness issues. Each of these mechanisms is described in detail in Chapters 5 and 6.



Issue	Missing Knowledge	Acquisitional Mechanism	Integrative Mechanism
G1-1	Generality of reasoning and value constraints in original rules	Identity analysis	Identity-based generalization
G1-2	Explanation of reasoning behind non-learning subgoal inference	Identity analysis in non-learning subgoals	Identity-based generalization
C1-1A	Dependencies to operators in supergoal	Build "architectural rules" and add to trace	Dependency analysis
C1-1B	Dependency on the reasoning that led to query	Build "architectural rules" and add to trace	Dependency analysis
C1-1C	Dependency on the function args, knowledge structures being copied	Dynamically add conditions for dependencies	Dependency analysis
C1-2	Dependencies to operator selection knowledge	Analyze operator preferences used and dynamically add relevant ones to trace	Dependency analysis
C1-3	Dependency on promoted knowledge (previous result)	Search working memory for dependency path	Rule repair
C1-4	Dependency on implicit knowledge	Add explicit representation, other	Dependency analysis
C1-5	Explanation of reasoning behind non-learning subgoal inference	Enable correctness mechanisms in non-learning subgoal	Dependency analysis
C1-6	Dependent on a LTM also recalled in another subgoal	(1) Search working memory for dependency path (2) Architecture modified to prevent	(1) Rule repair (2) N/A

Figure 4.3: Mechanisms to acquire missing knowledge and integrate into learning process.

For example, consider G1, the issue that can lead to undergeneral rules. As previously discussed, Soar's early commitment to learning from an instantiated trace led to a heuristic-based variablization mechanism that can only generalize those aspects of the reasoning that are encoded in Soar identifiers. For the most part, chunking ignores the underlying rules that describe why each condition of a rule matched a WME. This is valuable and easily accessible missing information that can allow chunking to determine what is generalizable, what limits there are on those generalizations, and which variables in a behavior trace refer to the same concept or object in the underlying reasoning. For example, an agent that uses a variable with a constraint in a rule is reasoning generally about the class of objects that has that same feature with a value that meets that constraint. This is generality that the online procedural learning algorithm should capture. Similarly, when one rule matches knowledge structures that another rule creates, any corresponding variables then share common semantics within the chain of reasoning. This information is also missing from the behavior trace that chunking analyzes. A trace of the original rules that matched, on the other hand, reveals the underlying semantics of the agent's reasoning by describing exactly which elements of a rule match involve generalizable reasoning and how elements in different rule firings are related. An online procedural learning algorithm can use this information to learn more general rules.

4.2 Strategy 2: Selective Specialization

We have found that correctness issues can sometimes be remedied by decreasing the generality of the rule learned in a specific, controlled way. Specifically, an incorrect rule can sometimes be fixed by selectively specializing only the problematic elements that were reasoned about. While the rule loses some generality, it gains correctness. Figure 4.4 shows the two issues that selective specialization is applied to.

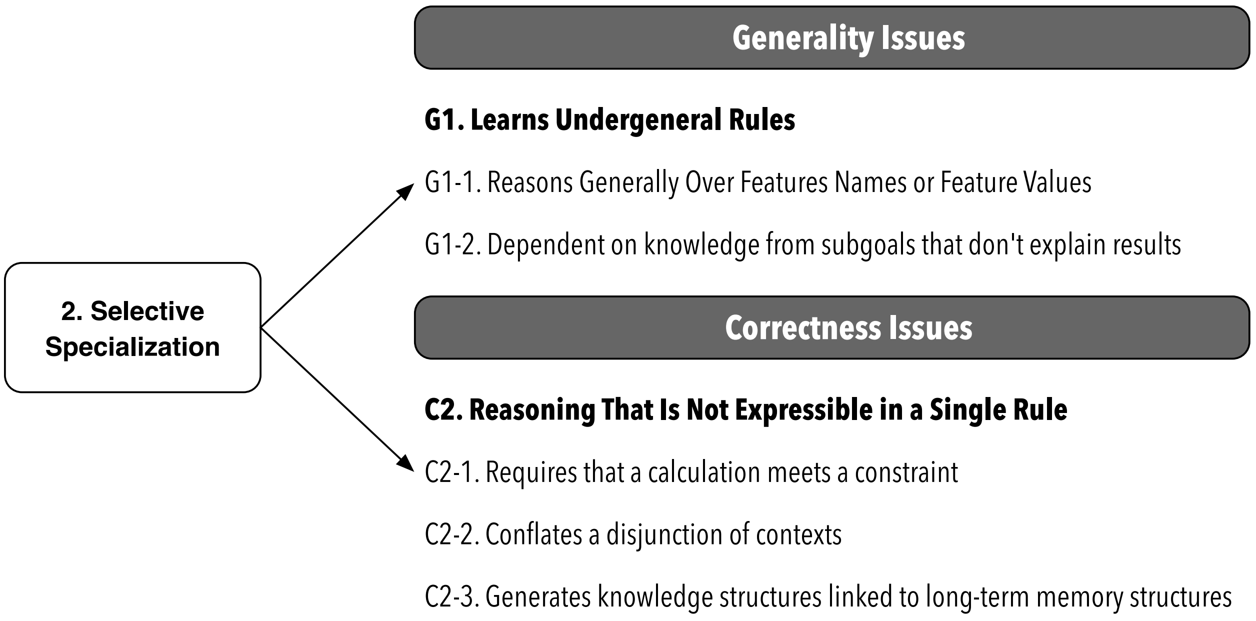

2. Selective Specialization
Generality Issues
G1. Learns Undergeneral Rules
G1-1. Reasons Generally Over Features Names or Feature Values
G1-2. Dependent on knowledge from subgoals that don't explain results
Correctness Issues
C2. Reasoning That Is Not Expressible in a Single Rule
C2-1. Requires that a calculation meets a constraint
C2-2. Conflates a disjunction of contexts
C2-3. Generates knowledge structures linked to long-term memory structures

Figure 4.4: Issues mitigated by strategy 2.

One situation in which this strategy is used is when summarizing reasoning that requires that an intermediate calculation satisfies a constraint. For the correctness issue to occur, one rule fires and performs a function calculation and another rule later fires that tests whether the first rule's function output meets a constraint. Because the second rule placed a constraint on the intermediate results, any learned rule must only match when the calculation returns a value that meets the constraint. Unfortunately, Soar's production syntax and matcher cannot currently perform an arbitrary calculation and test that it matches a constraint within a single rule. *EBBS handles this situation by selectively specializing the identities associated with both the elements that appear as arguments to the intermediate function and the elements in any conditions that test the output of the function*. This fixes the value of the intermediate calculation and any other numbers based on it and guarantees that any constraints in conditions that test the output of that RHS function will be met. While this will make the learned rule more specific, it will also ensure that the rule is correct.

4.3 Strategy 3: Selective Unification

If constraints on the structure of working memory exist and the agent's procedural knowledge implicitly assumes those constraints, an online procedural learning algorithm can learn rules with *unnecessary generality* that are computationally expensive to match. For example, our initial experimental version of EBBS, which used only EBL-style generalization of the reasoning without any selective unification, learned chunks that were so general that they could handle the situation where every reference to a supergoal in the trace could refer to an independent supergoal; technically there is nothing in the logic of the rules that says otherwise. These rules had an exorbitant number of similar conditions and slowed Soar's production matcher significantly. A rule that abstract is not only expensive to match but also completely unnecessary, because, as discussed in Section 3.2, Soar constrains working memory so that structures are rooted in goal nodes connected by unique superstate links, with each subgoal having exactly one parent supergoal.

To avoid building rules with unnecessary generality, EBBS uses a technique called selective unification. We will explain how the technique works after we explain how EBBS analyzes the generality of the rules that led to a behavior trace in Section 5.3. For now, we can say that EBBS uses selective unification to produce rules that also assume those implicit, architecturally-enforced representational constraints.

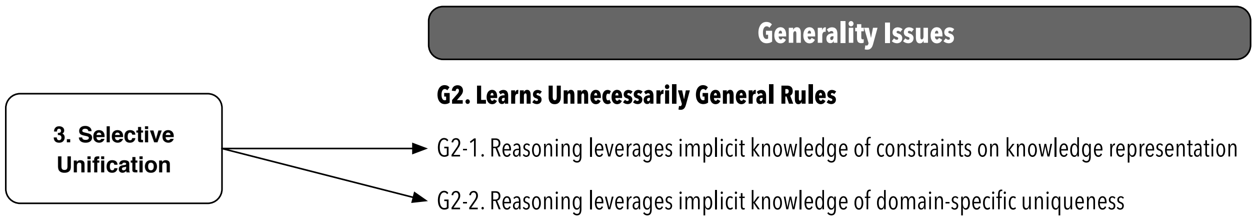

Generality Issues
G2. Learns Unnecessarily General Rules
3. Selective Unification
G2-1. Reasoning leverages implicit knowledge of constraints on knowledge representation
G2-2. Reasoning leverages implicit knowledge of domain-specific uniqueness

Figure 4.5: Issues mitigated by strategy 3.

4.4 Strategy 4: Modify Architecture

In Section 2.2, we described four necessary conditions for an online procedural learning algorithm to learn correct rules.

1. Reasoning must be logically consistent and contemporaneously possible.
2. Reasoning must be described by a behavior trace that explains all dependencies.
3. Reasoning must be expressible in a single rule.
4. Reasoning must be repeatable.

All four conditions involve a violation of one of the basic assumptions underlying an explanation-based online procedural learning algorithm. Our fourth strategy is to address such violations by changing the basic mechanisms of the cognitive architecture to ensure that they do not occur and that the online procedural learning algorithm's assumptions are always met. If the change succeeds in making the processing of the cognitive architecture parsimonious with the online procedural learning algorithm, the correctness issue can be avoided entirely.

Figure 4.6 shows the three correctness issues that this strategy is applied to.

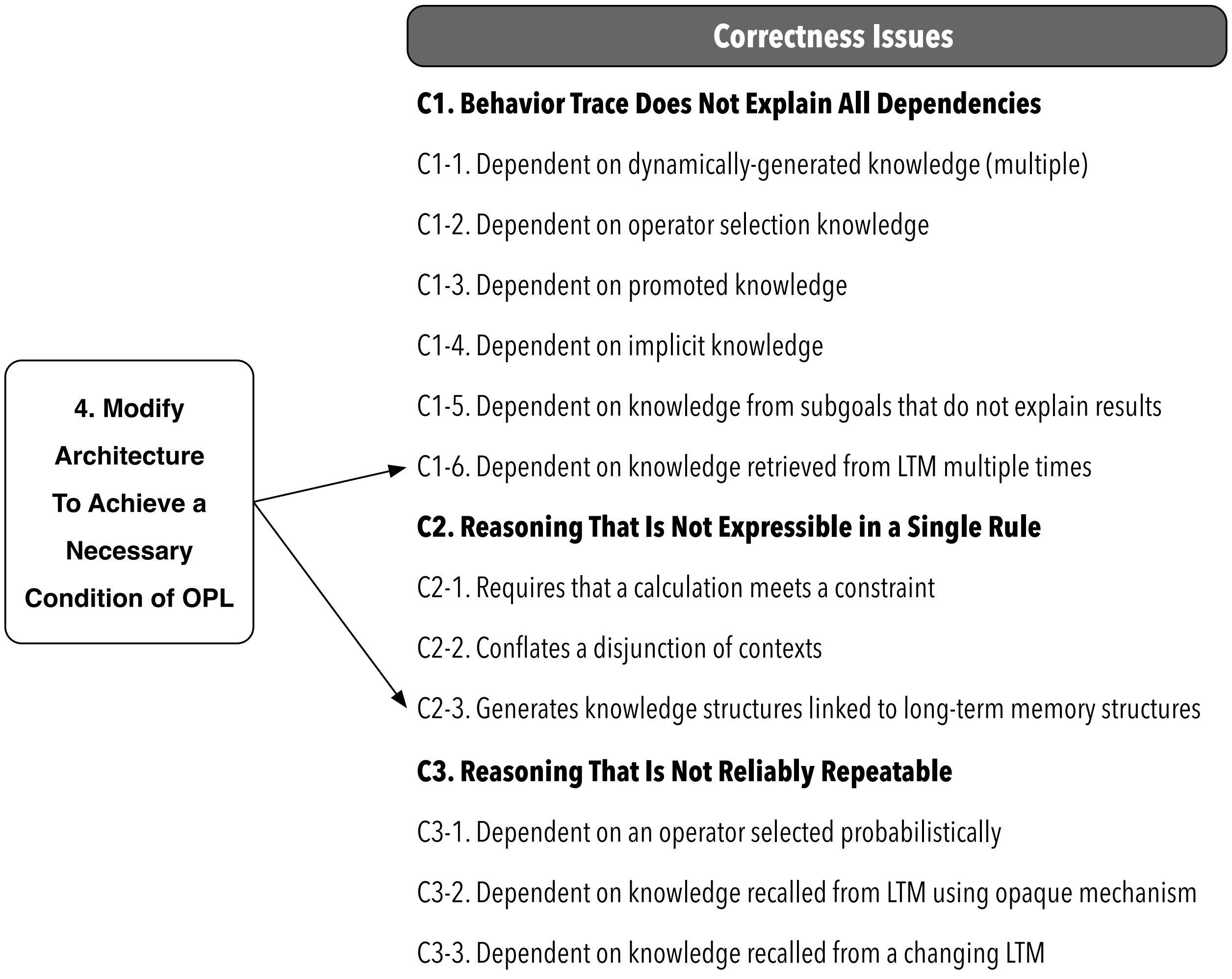

4. Modify Architecture To Achieve a Necessary Condition of OPL
Correctness Issues
C1. Behavior Trace Does Not Explain All Dependencies
C1-1. Dependent on dynamically-generated knowledge (multiple)
C1-2. Dependent on operator selection knowledge
C1-3. Dependent on promoted knowledge
C1-4. Dependent on implicit knowledge
C1-5. Dependent on knowledge from subgoals that do not explain results
C1-6. Dependent on knowledge retrieved from LTM multiple times
C2. Reasoning That Is Not Expressible in a Single Rule
C2-1. Requires that a calculation meets a constraint
C2-2. Conflates a disjunction of contexts
C2-3. Generates knowledge structures linked to long-term memory structures
C3. Reasoning That Is Not Reliably Repeatable
C3-1. Dependent on an operator selected probabilistically
C3-2. Dependent on knowledge recalled from LTM using opaque mechanism
C3-3. Dependent on knowledge recalled from a changing LTM

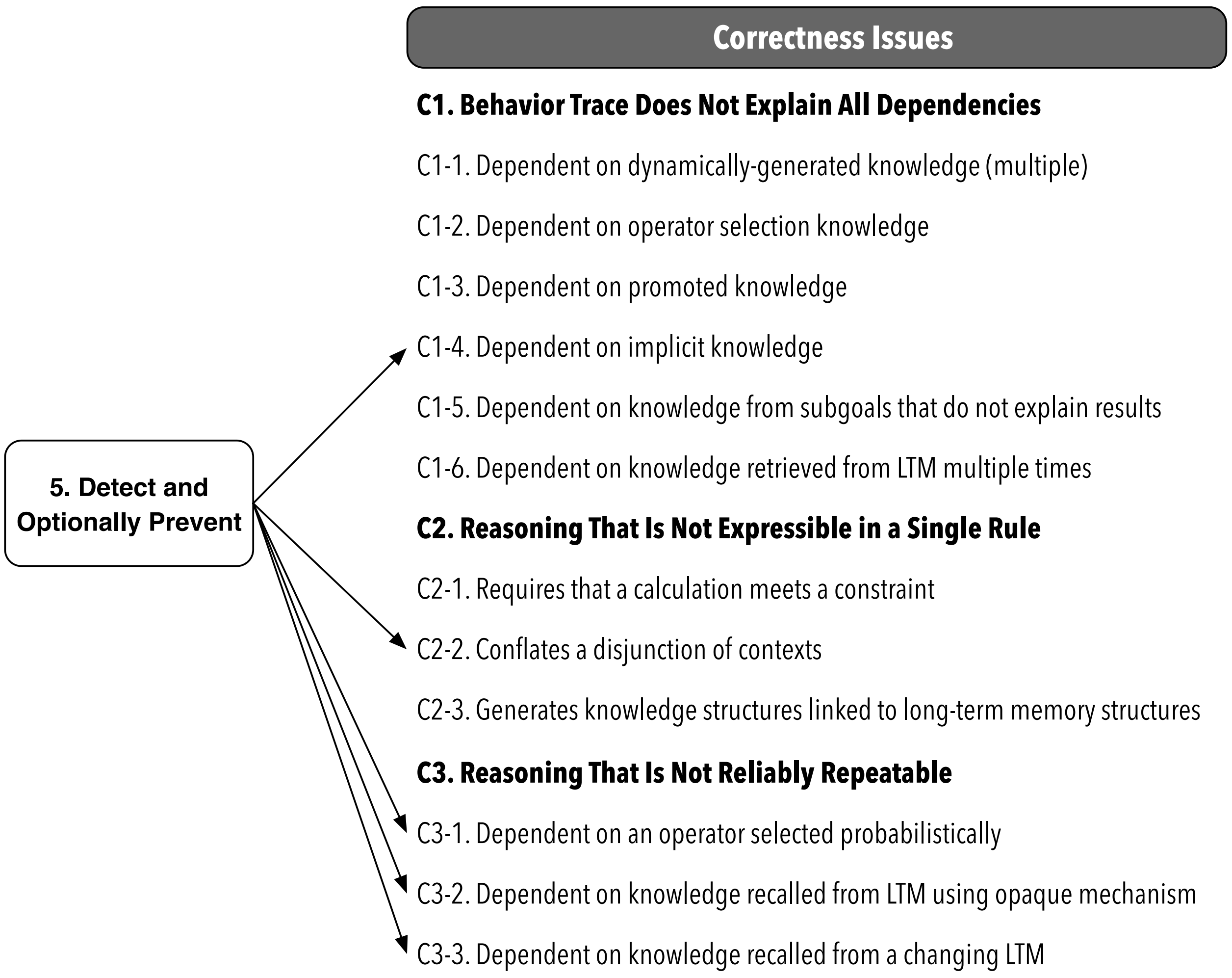
Figure 4.6: Correctness issues mitigated by strategy 4.

An excellent example of how correctness issues can be remedied by a major architectural change is Robert Wray's 1998 thesis on ensuring consistency in hierarchical architectures (Wray et al., 1996). In his thesis, Wray identified a significant correctness issue that chunking had when it was first applied to agents that operate in changing domains. Specifically, he described how changes to the environment that occurred during problem-solving could lead to learning incorrect, logically inconsistent rules, which he called non-contemporaneous chunks. In these cases, chunking attempted to summarize reasoning that involved two rules that could never match at the same time as each other. For example, consider an agent that *moves locations while in a subgoal*. Before the agent moves, a rule that tests the agent's location fires in the subgoal and creates a persistent local knowledge structure. The agent moves and then another rule matches, which also tests the agent's location as well as the structure created by the first rule. This second rule returns a result to the supergoal, causing a chunk to be learned. The version of chunking from that era would try to summarize this chain of reasoning and learn an impossible chunk that required that the agent is at both the starting and ending locations at the same time. Wray realized that the underlying issue was not a consequence of the online procedural learning algorithm, but, rather, a consequence of the Soar's model of subgoaling. To eliminate this source of incorrect rules, Wray improved one of the core aspects of Soar PSCM by adding a new architectural mechanism called the Goal Dependency Set (GDS) (Wray and Laird, 2003). The GDS monitors knowledge dependence in every subgoal to ensure that the reasoning in each is always consistent with higher level knowledge structures. If a change occurs in a knowledge structure that have persistent working memory elements in a lower subgoal dependent on it, the GDS removes the lower subgoal. Because the impasse was never resolved, a new subgoal is immediately created to resolve it again. This forces some reasoning to be repeated but ensures that reasoning in a subgoal is always consistent with the current state of the world. Consequently, any rules learned from that reasoning is temporally consistent. In summary, the GDS was able to eliminate one of chunking's correctness issues by ensuring that one of the online procedural learning algorithm's assumptions were met, namely that all reasoning in a subgoal is contemporaneously possible.

In this thesis, we eliminate two additional correctness issues. One of these problems, C1-6, was a particularly difficult and widespread problem that arose when retrievals of the same knowledge structures from semantic memory were retrieved in multiple subgoals. In our case, our solution was to fundamentally change how Soar represents semantic memories in working memory, which required a significant re-formulation of Soar's semantic memory system and how agents interact with it. This change was successful and allowed us to completely eliminate all unexpected interactions between memory retrievals and the incorrect rules they gave rise to. We revisit this problem in Section 6.3.2, after Section 6.3.1 provides the necessary background of how Soar represents long-term semantic memories using special *long-term Soar identifiers*, that have qualities of both a Soar identifier and a constant.

4.5 Strategy 5: Detect and Optionally Prevent

If the issues leading to potential incorrectness cannot be mitigated, an online procedural learning algorithm should not learn an incorrect rule that could produce unjustified inferences that interfere with task execution. To avoid this, EBBS detects reasoning that it cannot summarize correctly and aborts learning a rule in those situations. EBBS uses this detect-and-prevent strategy to handle four issues, as shown in Figure 4.7.[3]



Correctness Issues
C1. Behavior Trace Does Not Explain All Dependencies
C1-1. Dependent on dynamically-generated knowledge (multiple)
C1-2. Dependent on operator selection knowledge
C1-3. Dependent on promoted knowledge
C1-4. Dependent on implicit knowledge
C1-5. Dependent on knowledge from subgoals that do not explain results
C1-6. Dependent on knowledge retrieved from LTM multiple times
5. Detect and Optionally Prevent
C2. Reasoning That Is Not Expressible in a Single Rule
C2-1. Requires that a calculation meets a constraint
C2-2. Conflates a disjunction of contexts
C2-3. Generates knowledge structures linked to long-term memory structures
C3. Reasoning That Is Not Reliably Repeatable
C3-1. Dependent on an operator selected probabilistically
C3-2. Dependent on knowledge recalled from LTM using opaque mechanism
C3-3. Dependent on knowledge recalled from a changing LTM

Figure 4.7: Issues mitigated by strategy 5.

While this fallback strategy may seem superficial, it does make an important contribution. As described in Section 3.5, one of the reasons that classical chunking was not used frequently was that it was difficult to determine which situations an agent may learn incorrect knowledge until you

[3]We did not count its use in the implicit knowledge correctness issue because that particular mechanism is limited and is not used for all forms of implicit knowledge.

actually saw problematic behavior. While classical chunking did have a limited implementation of detect and prevent – it was able to detect local negations and disconnected conditions – it generally relied on the agent engineer to design the agent's reasoning in a way that incorrect rules would not be learned. And to do that, the engineer would need a deep understanding of many of the issues discussed in this chapter. So, while the detect and prevent strategy may seem superficial, adding algorithmic detection of the problematic situations eliminates an important reason behind why procedural learning is not often used.

EBBS also includes the option to "detect but not prevent." This may seem odd given how we describe the problems in this thesis. While our focus on where procedural learning can go wrong may make it seem like an incorrect rule will always result in a catastrophe, in practice, some correctness issues are not as problematic as they may seem. For example, the problem may involve an edge case that the designer is not worried about. Or, the designer may know that the knowledge source that the agent is using will return the same knowledge structures every time it is used, so it can be considered reliable. In cases like those, the agent designer may want to allow the system to learn rules despite the theoretically possibility that it may learn an incorrect rule. If that is the case, the designer can instead tell EBBS to flag the rules as potentially problematic but still add it to procedural memory, i.e. "detect but not prevent".

4.6 Applying the Five Strategies of EBBS

Now that we have described the five strategies at a high level, we can see how they map to every one of our correctness and generality issues, as shown in Figure 4.8. Chapter 5 will explain how the core EBBS algorithm uses the first three strategies to learn rules that are optimally generality, while Chapter 6 will discuss how EBBS uses the strategies to mitigate each of the 12 correctness issue.

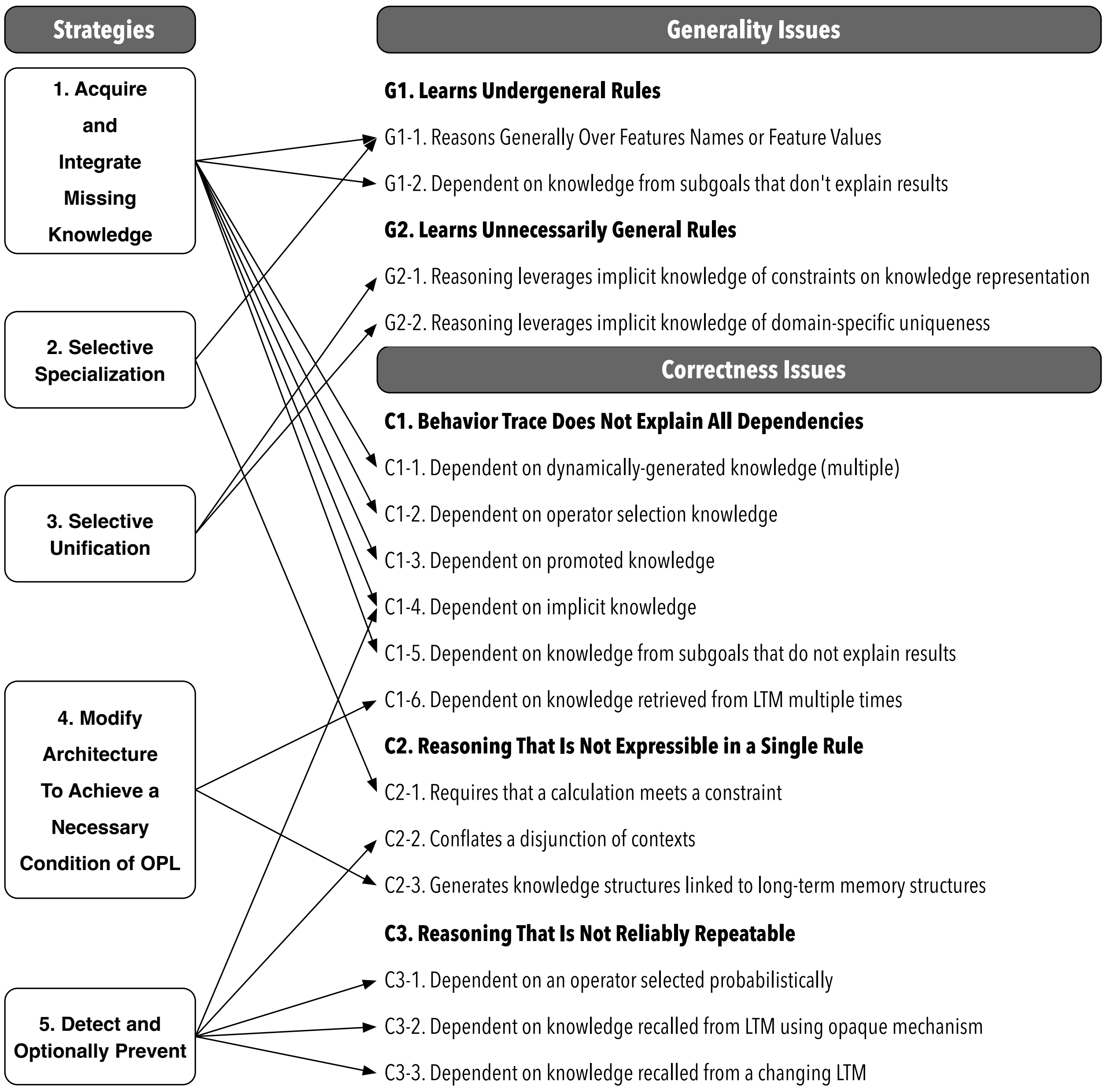

Strategies
1. Acquire and Integrate Missing Knowledge
2. Selective Specialization
3. Selective Unification
4. Modify Architecture To Achieve a Necessary Condition of OPL
5. Detect and Optionally Prevent
Generality Issues
G1. Learns Undergeneral Rules
G1-1. Reasons Generally Over Features Names or Feature Values
G1-2. Dependent on knowledge from subgoals that don't explain results
G2. Learns Unnecessarily General Rules
G2-1. Reasoning leverages implicit knowledge of constraints on knowledge representation
G2-2. Reasoning leverages implicit knowledge of domain-specific uniqueness
Correctness Issues
C1. Behavior Trace Does Not Explain All Dependencies
C1-1. Dependent on dynamically-generated knowledge (multiple)
C1-2. Dependent on operator selection knowledge
C1-3. Dependent on promoted knowledge
C1-4. Dependent on implicit knowledge
C1-5. Dependent on knowledge from subgoals that do not explain results
C1-6. Dependent on knowledge retrieved from LTM multiple times
C2. Reasoning That Is Not Expressible in a Single Rule
C2-1. Requires that a calculation meets a constraint
C2-2. Conflates a disjunction of contexts
C2-3. Generates knowledge structures linked to long-term memory structures
C3. Reasoning That Is Not Reliably Repeatable
C3-1. Dependent on an operator selected probabilistically
C3-2. Dependent on knowledge recalled from LTM using opaque mechanism
C3-3. Dependent on knowledge recalled from a changing LTM

Figure 4.8: Strategies used to mitigate generality and correctness issues.

CHAPTER 5

Learning Optimally General Procedural Knowledge

In this chapter, we provide an explanation of how EBBS learns rules that capture the underlying generality of an agent's reasoning by altering the core components of classical chunking and coupling it with a novel, two-stage unification algorithm that we call the "Distributed Identity Graph Unification" (DIGU).

5.1 Capturing the Generality of a Reasoning Trace

To understand how a rule captures the generality of an agent's original reasoning, we first examine how a rule's structure influences its generality. Even though it may seem extremely basic, consider the three rules in Figure 5.1, each of which compares a feature representing the agent's size with a feature representing an object's size and then prints out a message if the object is bigger than the agent. Production-1 can only match in a very specific situation, namely one where the agent's size feature is *exactly* 5 and the object's size feature is *exactly* 6. Production-2, on the other hand, can match a broader range of situations because it uses a variable and a constraint for the object size, which allows it to match for any object whose size feature is greater than 5. So, by using variables and constraints instead of literals, a rule can include representation of a class of objects, which allows it to apply to a broader range of situations. This is generality that an online procedural learning algorithm can capture. Similarly, the third rule applies to an even broader range of situations because it uses a variable for the agent's size, which allows it to apply to agents of all sizes. So, production-3 has even more generalizable aspects that can be captured.

```
sp {production-1
    (state <s> ^my-size 5
               ^object-detected <obj>)
    (<obj> ^size 6)
    -->
    (write (crlf) | I have detected an object bigger than me!|}

sp {production-2
    (state <s> ^my-size 5
               ^object-detected <obj>)
    (<obj> ^size {> 5 <object-size>})
    -->
    (write (crlf) | I have detected an object bigger than me!|}


sp {production-3
    (state <s> ^my-size <my-size>
               ^object-detected <obj>)
    (<obj> ^size {> <my-size> <object-size>})
    -->
    (write (crlf) | I have detected an object bigger than me!|}
```

Figure 5.1: Three similar rules, each with increasingly more generality that can be captured.

Now that we have established how generality is encoded in a single rule, we examine how the generality of classes of objects described by two rules interact when one rule matches based on the output of the other rule. For example, consider the relationships between the variables in the two rule matches of Figure 5.2:

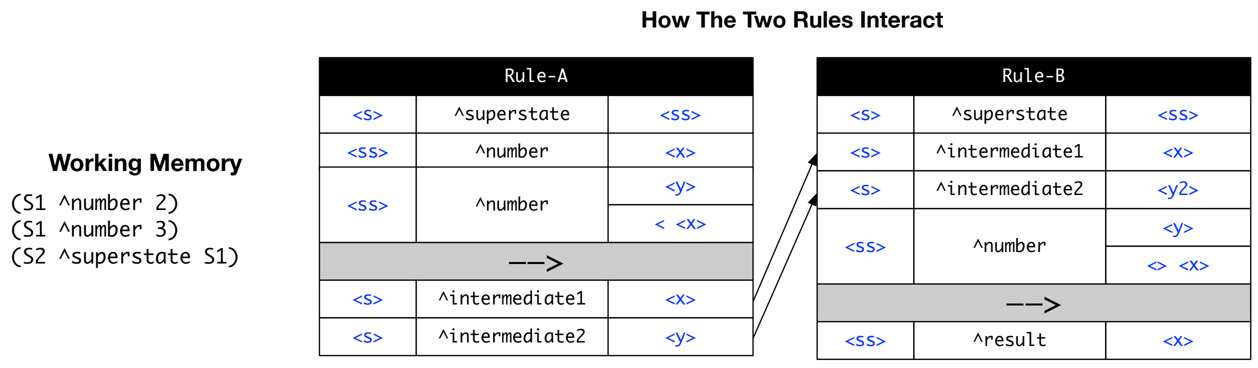


Figure 5.2: Two simple rules that matched in a substate.

The left pane of Figure 5.2 shows two rules and a set of working memory elements that would cause them to match, while the right pane shows the relationship between the two rules after they match. The arrows indicate how WMEs that are created by the first matched rule are then tested by the second matched rule.

Note that the actual symbol used for a variable isn't meaningful across rules. It doesn't matter that the <y> in rule-a uses the same variable name as the <y> in rule-b. It also doesn't matter that both conditions with <y> happen to match the same working memory element, `(S1 ^number 3)`. In terms of sharing an identity, the only thing that matters is how the rules interact, namely whether there's a connection between elements in the condition of one rule and elements in the actions of another rule.

To facilitate describing *elements in a behavior trace that refer to the same underlying object or feature*, we say that such elements have a *shared semantic identity*. The most trivial example of a shared identities are how all instances of a variable within a rule, by definition, must refer to the same underlying object or concept. For example, we know all of the <y>'s in rule-a must refer to the same number. A more interesting example is how, when two rules have a connection like the one described in 5.2, the elements in the condition of the second rule that matched the elements of the actions of the first rule represent the same underlying object. For example, the connection between rule-a and rule-b implies that the elements that matched variables on the right-hand side of rule-a, <s>, <x> and <y>, refer to the same objects/features as the elements that matched the corresponding conditions in rule-b, <s>, <x> and <y2>. So, the <x> in rule-b shares the same identity as the <x> in rule-a. Similarly, the <y2> in rule-b shares the same identity as <y> in rule-a. In contrast, the <y> in rule-b does *not* share the same identity as the <y> in rule-a.

In the example shown in Figure 5.2, neither rule placed any constraints on the values that could match, so they both specified the underlying objects at the same level of generality. But that is not always the case. Rules that are connected in this way often reason about the underlying classes of objects at different levels of generality. For example, consider what happens if the second rule adds a constraint on <x>, as shown in red in Figure 5.3. While the rule-a will create an `intermediate1` feature based on any two `number` features that its conditions matched, rule-b will only match and fire if the `intermediate1` feature the first rule created is less than 4. So, the combined reasoning from these rules only applies if two numbers exist and the larger `number` one is less than 4. Since working memory contained the numbers 2 and 3, all of the constraints of the two rules were met. But, if working memory contained the numbers 3 and 4 instead, the first rules would still match but *the sequence of two rules would no longer match.*

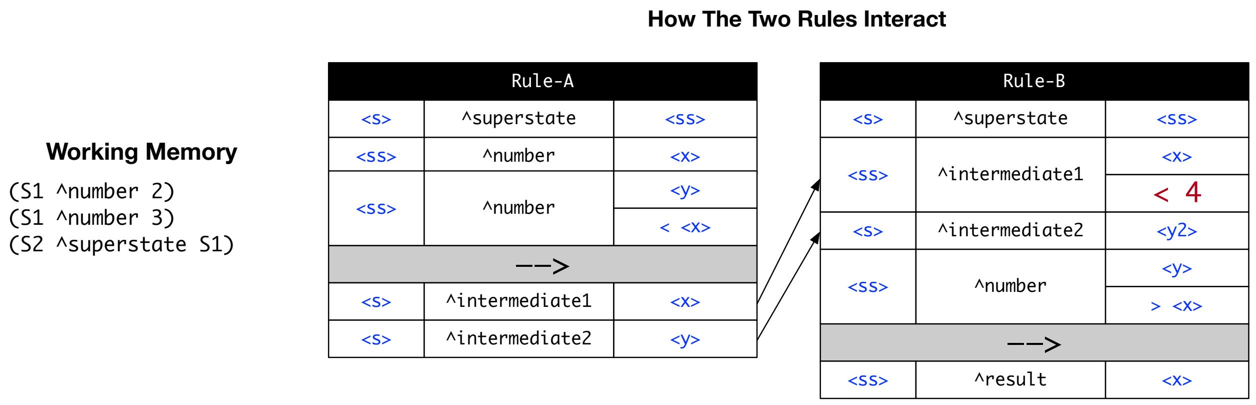


Figure 5.3: Two simple rules that matched in a substate where second rule adds a constraint.

Consequently, to build a rule that summarizes multiple connected rule firings and applies only in the same contexts, an online procedural learning algorithm must describe any elements with a shared identity in a way that the constraints of each of the individual rules of the trace placed upon that shared identity are also satisfied. To do that, an online procedural learning algorithm must extend this analysis to all connections in a behavior trace to determine which elements in the overall trace share an identity and what constraints must be met to support the entire trace of reasoning.

5.2 Challenges of Using Classical EBL Unification

The identity formalism that we have begun to describe is related to the analysis that classic EBL algorithms perform to generalize rules. As described in Section 3.4, EBL algorithms are able to learn general rules by using unification algorithms such as variable regression (Waldinger, 1977) or EGGS (Mooney et al., 1986) to determine the most general unification of variables that an explanation can support. To do that, those unification algorithms analyze the generality of an explanation trace by traversing it and examining exactly what we described in the last section: how the variables in the actions of one rule match up with the variables in the conditions of other rules. They use these pair-wise comparisons in different ways. For example, when regression analyzes two connected rules, it directly replaces the variables in the rule that created a working memory element with the variables names used in the rule that used the working memory element.[1] The EGGS algorithm, on the other hand, builds a table of variable pairs that indicate when two variables in a trace were connected by two rule firings. Note that all unifications algorithms have been shown to produce the same most general unification (Mooney et al., 1986).

Early versions of EBBS implemented versions of both of these classic unification algorithms. The more successful of the two versions implemented a particular extension of Mooney's EGGS algorithm as described in (Mooney, 1989). While that system did provide the generality that our online procedural learning algorithm needs, some agent designs had debilitating issues with efficiency. We argue that the reason for this inefficiency is that summarizing a behavior trace in a cognitive architecture introduces computational challenges more demanding than those faced in the EBL systems for which EGGS was developed.

One reason for this computational challenge is that a behavior trace can describe algorithmic-like processing. For example, it is not uncommon for an agent to use a few rules to maintain a counter in a subgoal, i.e. a working memory element with a numeric value element that is repeatedly replaced with another WME with an incremented value. Every time the counter is incremented, at least one new instantiation is created that is dependent on the instantiation that created the previous value. So, a simple counter that did nothing but increment a value every decision cycle would quickly produce a trace with thousands of chained instantiations. Analyzing such traces requires a prohibitively expensive number of variable mappings in the lookup table that EGGS uses, so much so that, in our initial EGGS-based implementation of EBBS, agents would lose reactivity when they tried to learn new rules from such algorithmic-style reasoning.

Another reason for the computational challenges of analyzing a behavior trace is that it is a historical account of processing, which can be arbitrarily long. In some sense, there is no *learning cost* to building large traces since it is built *prior* to the learning episode. In contrast, classic EBL systems would require computation *during the learning episode* to build similarly complex explanations; this is because those systems built the explanation on demand using a means-ends search through

[1]To do this, both regression and EGGS must "uniquify" the variables in the trace to ensure that the same variable name is not used in two different rules.

a knowledge base of domain rules. Because brute-force search is inherently expensive and many (possibly unbounded) explanations for a training example may exist, EBL systems tend to favor simpler explanations, which are easier for a search algorithm to find.

Finally, these problems are compounded by the fact that this learning algorithm exists in a cognitive architecture as an automatic mechanism that is continually occurring each time knowledge is added to a supergoal. Such integrated, automatic online procedural learning algorithms may be tasked with analyzing dozens or even hundreds of these large traces every decision cycle. Since each decision cycle must be under 50 ms to meet our performance desideratum, an algorithm that is reliably efficient is essential.

5.3 Distributed Identity Graph Unification

Because the challenges described in the last section made it difficult to ensure the reactivity required by desideratum D4 and facilitate the type of operations needed to mitigate correctness issues, we developed a new unification algorithm that we call the *"Distributed Identity Graph Unification" (DIGU) algorithm*, which successfully eliminates the inefficiencies we have encountered with our initial versions of EBBS and provides the framework for our other mechanisms.

5.3.1 Identity: A Common Formalism Across EBBS Strategies

One of the key insights that we have had in this work is that there is a tight relationship between generality and correctness: adding mechanics that capture more of the generality of the underlying reasoning revealed new correctness issues that could often be mitigated by selectively decreasing generality. This suggests that the mechanisms that capture generality must work closely with the mechanisms that prevent correctness issues. To allow the various mechanisms of EBBS to contribute to the analysis of the learning episode, our approach adopts a common formalism that all mechanisms use, which is based on the concept of "shared semantic identity" that we previously discussed.

While classic EBL unification algorithms perform a syntactic analysis of a behavior trace to determine which variables to substitute, what they are really doing is determining semantic identity. As we previously described, different algorithms build the mappings in different ways – some build tables, some propagate variables substitutions in place – but they all use them in the same way: they use the mappings to replace original elements in the explanation trace with the unified variables that they are mapped to. To translate to our formalism, all variables in the EGGS unification table that map to the same variable refer to the same underlying object or feature and, hence, share the same semantic identity.[2]

Readers may wonder whether identity is just a convenient term to facilitate discussion or just another way to describe a substituted variable or set of them, but that is not the case. In our approach, *identities are explicit first-class objects that the EBBS algorithm creates and manipulates*. Each identity has its own identity data structure, which is created to explicitly represent that EBBS has detected a potential new underlying object or feature that is referred to by one or more variables in the trace. Identities are composed into what we call *the identity graph*, which essentially describes what EBBS knows about the overall generality of reasoning in a subgoal. While the EBBS algorithm does use the variables in the original rules to inform how it creates identities, the identity structures themselves are an independent system and do not contain any references to variables or any other elements in the behavior trace. They simply describe what EBBS has learned about the underlying object or feature. So, when EBBS map another variable to an existing identity, because it is not making any new claims about the underlying object or concept, it does not need to alter the identity data structure at all. The only thing EBBS is asserting when it maps a variable to an existing identity is that the variable in question is guaranteed to refer to the same underlying object as all other variables mapped to that identity.

One of the benefits of this design is that it allows the various mechanisms of EBBS to alter the identity graph as needed to influence how EBBS generalizes the rule in the final step. For example, if one of EBBS's mechanisms determines that two different variables refer to the same underlying object or feature, it can temporarily join the two identities by forcing one to refer to the other. Similarly, if one of the mechanisms needs to specialize an element to ensure correctness, it can map that identity to a special non-generalizing identity, which we call the null identity. In short, identity is not simply syntactic sugar; it is the central currency of our approach.

5.3.2 The Identity Graph

In this section, we provide an overview of how the identity graph works. The particulars of how identities are created, manipulated and used will be discussed in the remainder of Chapters 5 and 6, alongside the various mechanisms of EBBS in which they happen.

An identity represents an underlying object or feature that is being reasoned about in a subgoal. Each identity is represented as a node in an identity graph. Whenever an instantiation is created for a rule firing, EBBS examines the elements in the underlying rules and, as it detects potentially new underlying objects or features, adds new identity nodes as needed to the identity graph. How

[2]This argument can also be extended to variable regression; one could simply collect all of the variable substitutions that a regression algorithm eventually makes into a unification table that would similarly show which concepts and objects shared the same semantic identity.

EBBS creates the mapping between elements in a rule and identities is explained in greater detail in Section 5.6.1. Note that EBBS does not create a new identity for every single element in a rule that matched a variable. Multiple elements in the behavior trace, even those from different rule instantiations, can be initially mapped to the same identity. Identities do not contain links back to the variables in the behavior trace.

To provide a mechanism to merge identities, each identity has one outgoing edge, which is called its super-join. Initially, though, the graph is unconnected, i.e. each identity's super-join edge points back to itself. Note that each subgoal has its own identity graph that exists as long as the subgoal exists.

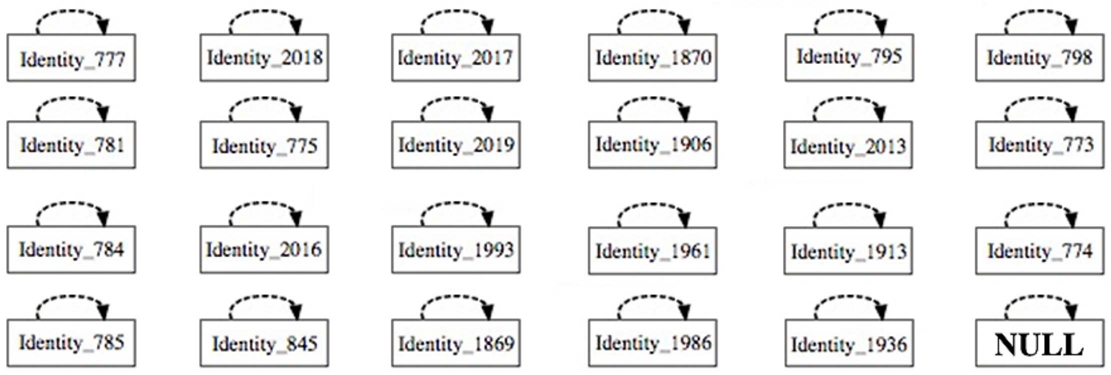


Figure 5.4: An identity graph before a learning episode.

During a learning episode, EBBS may determine that two identities refer to the same underlying object or feature. If so, it sets the edge of one identity to point to the other. Figure 5.4 shows a visualization of an identity graph after 10 rules have fired in a blocks-world subgoal. Each block in the figure represents an identity.[3] While all of the constituent identities are the same as those in Figure 5.4, the super-join edges are different and indicate which identities are shared. For example, the edges of Figure 5.5 indicate that all of the elements in a particular behavior trace that were mapped to `Identity_774`, `Identity_2018` and `Identity_2017` referenced the same object or feature.

[3]The text within each block, for example `Identity_777`, reflects is an identification number that EBBS assigns to each identity to facilitate debugging.

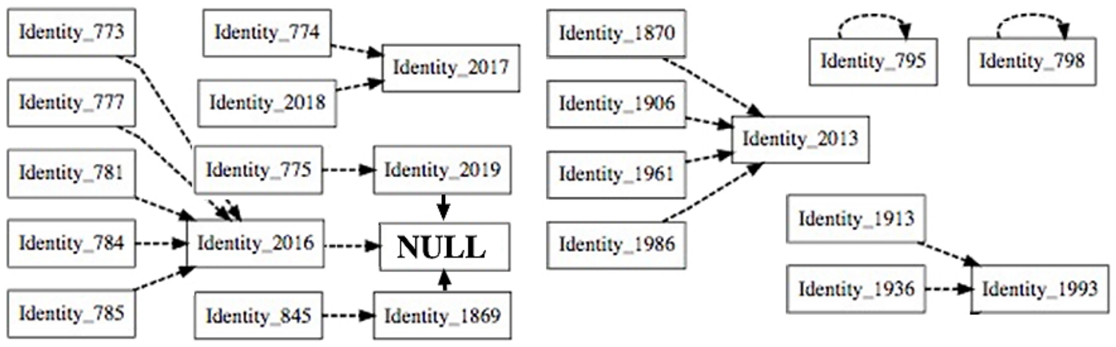


Figure 5.5: An identity graph after learning a rule.

Every identity graph also includes the **NULL identity**, which is a special identity that indicates that an element cannot be generalized and must contain a specific value. All elements in the original rule that reference specific constant values are trivially mapped to the NULL identity, but elements in a trace that match a variable can also be mapped to the null identity. When an identity is mapped to the null identity, we say the identity has been *literalized*. (This is because such elements will produce a literal value in the final learned rule.) The NULL identity is a special node in the graph and has no outgoing edge.

While not visually depicted in the figure, an identity can also be temporarily annotated with any constraints that a behavior trace imposes on the values that the underlying object/concept can have. These constraints may be relational and may even reference other identities. For example, a constraint may indicate that the value of one identity must be less than the value of another.

After a learning episode is complete, all super-join links and constraint annotations are reset back to the state shown in Figure 5.4. This allows additional learning episodes in that subgoal to re-use the identity graph.

5.3.3 Summary of DIGU Algorithm

There are four novel aspects to our DIGU algorithm:[4]

1. Instead of framing the problem syntactically in terms of variable substitutions and mappings, EBBS frames the problem around the underlying objects and features by adopting the formalism of object and feature *identity*. Two *variables share an identity* if they both refer to the same underlying object or feature, and an identity represents that underlying object or feature. Treating "sameness" as a first-order object facilitates performing operations on and

[4]The first item has been discussed extensively, but we include it in this list for completeness.

storing data related to a group of generalizable elements in a trace at any point during the learning episode, even before the set is fully determined.

2. DIGU takes advantage of the fact that a cognitive architecture generates its behavior trace differently than other explanation-based systems. In contrast to EBL systems which typically have a proof-generation component that uses domain knowledge to build explanations on demand for a particular training example (Dejong and Mooney, 1986), in a cognitive architecture, a history of task execution is incrementally created and *can be re-used repeatedly by multiple learning episodes*. For example, consider a single subgoal that has multiple rules fire in it, each of which creates supergoal knowledge. In such a case, multiple chunks are learned from the same set of subgoal instantiations, albeit different subsets of them. DIGU leverages the persistence of a behavior trace across learning episodes by splitting its *identity analysis* component into two distinct mechanisms that occur at different times:
    (a) The first mechanism, forward identity propagation, performs partial (and potentially incomplete) unification every time a rule matches and is added to a behavior trace. This amortizes the cost of the majority of EBBS's unifications and, more importantly, allows the computation performed by this mechanism to be re-used across multiple learning episodes.
    (b) The second mechanism, identity resolution, refines the initial identity mappings based on the subset of rule firings that played a role in a result. This phase refines the unifications provided by forward identity propagation by doing things that can only happen after the full trace is determined (joining identities, specializing identities, determining overall constraints) to determine the most general unification that a behavior trace will support.

3. To avoid problems with expensive look-ups and redundancies, DIGU builds an explicit, *identity graph* for each subgoal incrementally as it builds the behavior trace. Like the instantiation graph, the identity graph is a persistent data structure that exists as long as the subgoal exists. During instantiation creation, when the forward identity propagation phase occurs, DIGU adds new identities to the identity graph for any new generalizable elements it encounters in the instantiation being added. Prior to a learning episode, the identities in the graph are completely unconnected as shown earlier in Figure 5.4. During a learning episode, i.e. when EBBS analyzes the behavior trace, the identity resolution mechanism manipulates the identity graph by joining identities and adding constraints. Figure 5.5 shows an identity graph after EBBS has analyzed a trace and performed identity resolution. This representation allows EBBS to perform efficient operations on identities.

4. To allow unifications to be reused across multiple learning episodes in a subgoal, the identity

graph operations performed are reset so that it can be re-used in future learning episodes. In other words, the identity graph is reset back to the unconnected state shown in Figure 5.4.

5.4 EBBS Processing Overview

Now that we have discussed identity and defined our core terminology, we show an overview of the processing EBBS performs.

While most of the work that EBBS performs occurs after a rule in a subgoal fires and adds knowledge to the superstate, some critical aspects of the analysis it performs occur during the agent's task execution prior to learning. In Figure 5.6, the mechanisms of EBBS are split up into two groups. The first group encompasses all mechanisms that occur during the routine decision-making processes of the cognitive architecture. The second group of mechanisms is those involved specifically in *the learning episode*, which occurs immediately after a subgoal rule fires and creates knowledge in the supergoal. This is when the EBBS algorithm analyzes the behavior trace, modifies the identity graph and builds a new rule to add to an agent's production memory.

To highlight the differences between EBBS and Soar's existing online procedural learning system, the items in bold indicate new mechanisms that EBBS introduces.[5]

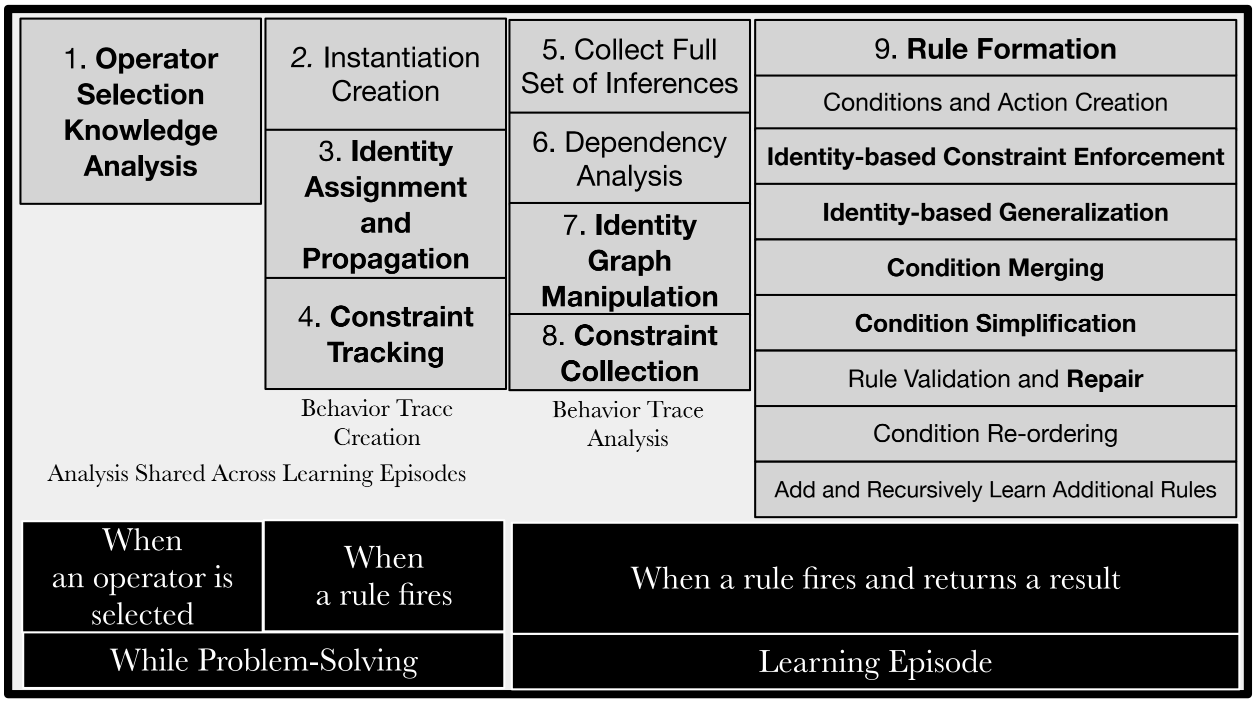

1. **Operator Selection Knowledge Analysis**
2. Instantiation Creation
3. **Identity Assignment and Propagation**
4. **Constraint Tracking**
Behavior Trace Creation
Analysis Shared Across Learning Episodes
5. Collect Full Set of Inferences
6. Dependency Analysis
7. **Identity Graph Manipulation**
8. **Constraint Collection**
Behavior Trace Analysis
9. **Rule Formation**
Conditions and Action Creation
Identity-based Constraint Enforcement
Identity-based Generalization
Condition Merging
Condition Simplification
Rule Validation and **Repair**
Condition Re-ordering
Add and Recursively Learn Additional Rules
When an operator is selected
When a rule fires
When a rule fires and returns a result
While Problem-Solving
Learning Episode

Figure 5.6: Overview of processing that EBBS performs to learn a rule.

[5]The other mechanisms all have some modifications, mostly related to supporting identity-based analysis, but they generally work the same as they did under classical chunking.

1. **Operator selection knowledge analysis**
   Analysis of the knowledge that contributed to operator selection occurs immediately after a new operator is selected in any subgoal. In Soar, operators are selected based on the content of preference memory, which is separate from working memory and only contains operator preferences. There are several types of preferences; some examples are preferences that indicate that an operator is required, desirable, better or worse than another operator. As Soar decides which operator to select, EBBS monitors Soar's preference resolution algorithm to determine which operator selection preferences were *necessary* for that particular operator to be selected. Preferences, like working memory elements, are created by rules, so, any time an operator is selected, EBBS stores references to the instantiations that created the necessary preferences for it. If another instantiation is later created *that tests that selected operator*, EBBS creates a link between that new instantiation and the set of instantiations that contributed preferences for the selected operator. This process effectively expands the behavior trace to also include operator selection reasoning. This mechanism has some complexity and is discussed in greater detail in Section 6.1.2.

2. **Instantiation Creation**
   Whenever a rule matches during agent execution, EBBS creates an internal record of the rule that fired that contains instance information about what matched (the specific working memory elements) along with some extra explanatory information about how the underlying rule matched. This is discussed in Section 5.5.

3. **Identity Assignment and Propagation**
   Whenever a rule matches during execution, this component creates the data structures needed to determine which elements in a behavior trace are generalizable and which ones share the same identity. To make this determination, this component does two things: (1) it annotates elements of the behavior trace with initial identity mappings and (2) it builds a persistent identity graph for the subgoal that will later be manipulated in individual learning episodes. This component will be discussed in Section 5.6.1.

4. **Constraint tracking**
   Whenever a rule fires, this component stores within the instantiation any value or relational constraint (e.g. `<> <x>`, `>= 3.14`, « `disjunction of constants` ») placed on the variables in the original rule. Constraint tracking is discussed in Section 5.6.2.

5. **Calculate Full Set of Inferences**
   When a rule creates results that includes a Soar identifier, it indirectly connects any another local working memory structures that are rooted at that identifier. In other words, when a

working memory structure becomes a result, all of its children also become results. Consequently, one of the first tasks that must be performed is determining exactly which working memory elements become accessible to the supergoal. Calculating the full set of inferences is discussed in Section 5.7.1.

6. **Dependency Analysis**
   This component is the first step of an actual learning episode, i.e. the processing that occurs after a subgoal produces a result for a supergoal. The dependency analysis component backtraces through the behavior trace to determine which conditions in matched rules test working memory elements in a supergoal. The rule formation mechanism will later use all the conditions that dependency analysis collects as the basis for the conditions of the learned rule. This is the step that is most similar to classic chunking, though it does have a couple key new differences, which are discussed in Section 5.7.3.

7. **Identity Graph Manipulation**
   As the dependency analysis component backtraces through the behavior trace, this component performs some basic graph operations on the identity graph. These operations allow EBBS to both mitigate correctness issues and build rules that are more optimally general. This is discussed in Section 5.7.4.

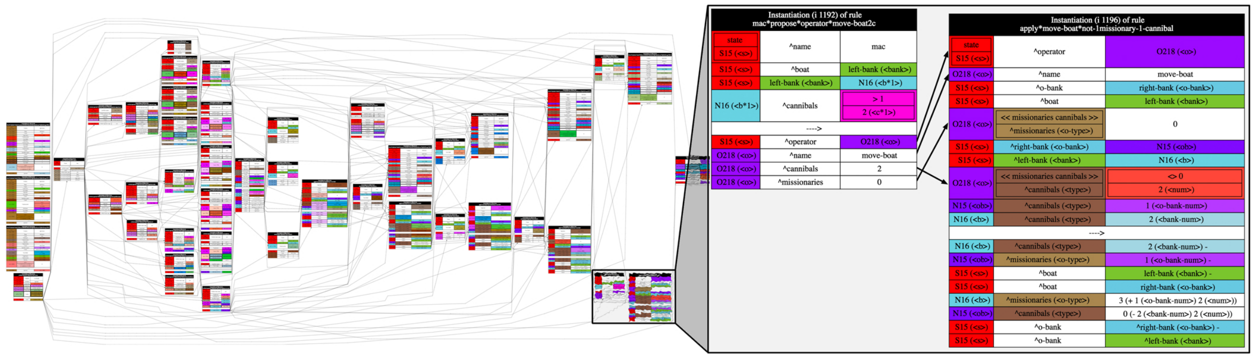
8. **Constraint Collection**
   Any constraints encountered during backtracing, which were previously recorded during constraint tracking, are copied to the relevant nodes of the identity graph. These constraints will be enforced during the rule formation phase to make sure that the learned rule only fires in situations that have the necessary qualities of the situation that was learned from. This is discussed in Section 5.7.5.

9. ***Rule Formation***
   Mechanisms 1-8 perform the analysis that EBBS needs to form a general and correct rule. The ninth mechanism, rule formation, uses the results of that analysis to build the new rule. Rule formation has eight different stages of its own, all of which are discussed in greater detail in Section 5.7.6. If a valid rule is created, Soar immediately adds the rule to production memory.

5.5 The Behavior Trace

Like classical chunking, EBBS *incrementally* builds a behavior trace that describes the agent's reasoning. Every time a rule matches during agent processing, EBBS stores a copy of the instantiated rule. As first described in Section 3.2, the instantiation of each rule consists of the set of working memory elements that matched each condition of the rule and the set of working memory elements created by the rule. The algorithm also stores a link from each matched WME in the new instantiation to a right-hand side action of the existing instantiation that created the matched WME. This allows it to later analyze the relationship *between elements in one matched rule with those in another*.

We call the set of instantiations that were created in a subgoal the *instantiation graph* of that subgoal. The behavior trace is a *subgraph* of the instantiation graph. To calculate the behavior trace, EBBS *backtraces* through the instantiation graph from the rule that creates a result. This allows it to determine which instantiations in the graph participated in the reasoning that produced the result. The behavior trace is that set of instantiations and the connections among them. Figure 5.7 shows an example of a behavior trace.



Instantiation (i 1192) of rule mac*propose*operator*move-boat2c
state
S15 (<s>) ^name mac
S15 (<s>) ^boat left-bank (<bank>)
S15 (<s>) left-bank (<bank>) N16 (<b*1>)
N16 (<b*1>) ^cannibals > 1 2 (<c*1>)
---->
S15 (<s>) ^operator O218 (<o>)
O218 (<o>) ^name move-boat
O218 (<o>) ^cannibals 2
O218 (<o>) ^missionaries 0
Instantiation (i 1196) of rule apply*move-boat*not-1missionary-1-cannibal
state
S15 (<s>) ^operator O218 (<o>)
O218 (<o>) ^name move-boat
S15 (<s>) ^o-bank right-bank (<o-bank>)
S15 (<s>) ^boat left-bank (<bank>)
O218 (<o>) << missionaries cannibals >> ^missionaries (<o-type>) 0
S15 (<s>) ^right-bank (<o-bank>) N15 (<ob>)
S15 (<s>) ^left-bank (<bank>) N16 (<b>)
O218 (<o>) << missionaries cannibals >> ^cannibals (<type>) <> 0 2 (<num>)
N15 (<ob>) ^cannibals (<type>) 1 (<o-bank-num>)
N16 (<b>) ^cannibals (<type>) 2 (<bank-num>)
---->
N16 (<b>) ^cannibals (<type>) 2 (<bank-num>) -
N15 (<ob>) ^missionaries (<o-type>) 1 (<o-bank-num>) -
S15 (<s>) ^boat left-bank (<bank>) -
S15 (<s>) ^boat right-bank (<o-bank>)
N16 (<b>) ^missionaries (<o-type>) 3 (+ 1 (<o-bank-num>) 2 (<num>))
N15 (<ob>) ^cannibals (<type>) 0 (- 2 (<bank-num>) 2 (<num>))
S15 (<s>) ^o-bank ^right-bank (<o-bank>) -
S15 (<s>) ^o-bank ^left-bank (<bank>)

Figure 5.7: A behavior trace of a subgoal in an agent that solves the Mouse and Cats puzzle. The rule that returned a result and gave rise to a chunk is magnified.

5.6 Integrating Identity into the Behavior Trace

If we stopped at this point in the description, EBBS's instantiation graph would be equivalent to the one created by classical chunking. Each instantiation describes the working memory elements that matched each condition and the working memory elements that are created by each action.

To create rules that summarize behavior at the same level of generality as the agent's underlying reasoning, DIGU needs to understand the generality of the rules that led to the behavior trace. Specifically, it needs to determine which elements in conditions are variables and which ones are literal constants, which variables are the same variables, what constraints must be met on the values of each variable and what relationships there are between variables within a rule and between rules.

While none of this information is contained in the behavior trace that chunking analyzes, it is available in the agent's production memory. So, EBBS applies strategy 1, acquire and integrate missing knowledge, to take advantage of this missing information. As described in Section 4.1, this strategy can be described in terms of an acquisitional mechanism to provide the missing knowledge and an integrative mechanism that incorporates that missing knowledge into the algorithm's analysis. For generality issues, DIGU acquires the needed missing information by examining the logic within each rule that led to a match in the behavior trace. Specifically, every time a rule's conditions match and EBBS creates an instantiation to document it, DIGU iterates through the conditions and actions of the matched rule and *annotates every element*, which is a literal matched value, *with a pointer that maps it to either a new or existing identity in the identity graph*. The following section describes this process in detail, including how DIGU chooses whether to create a new identity or use an existing identity for the mapping.

5.6.1 Identity Analysis Phase 1: Identity Creation and Propagation

EBBS uses the rules underlying a behavior trace to determine how objects and features are reasoned about during the problem solving. *When an instantiation is created, EBBS maps all elements of every condition and action to an identity, creating new identities as necessary.* The goal of this phase is neither to assign every variable with its own identity nor to perform every unification needed to learn a rule. Rather, the goal of this phase is to perform a set of common unifications that all learning episodes in this subgoal will share.

Identities are created and propagated using the following rules:

1. Any element in an instantiation that matched a literal value in the original rule is mapped to the NULL identity.

2. Any element in an instantiation that matched a variable must be mapped to the same identity as other elements in the same rule that use the same variable.

   This is the trivial case. The semantics of rules implies that two identical variables must match the same underlying object or feature. DIGU always maps these elements to the same identity.

3. If an element in an instantiation matches a variable, DIGU *creates* a new identity in two cases:

   (a) The element is a variable that is *in a condition* that matches a supergoal WME.[6]

[6]Note that if two conditions match the same supergoal WME, EBBS will consider them separately and give the elements of each condition their own identities. (In classic chunking, the second condition would be ignored since its generalization heuristic would always produce a duplicate condition.)

    Because Soar's online procedural learning approach is about determining how subgoal reasoning relates to supergoal knowledge, we assign new identities to references to supergoal knowledge.

   (b) The element is a variable that is used *in a RHS action* but does not appear in any condition of the rule.

       In these cases, the variable is being used to create a new Soar identifier, so a new identity is created for that new object. (These elements are also known as unbound RHS variables.)

4. If an element in an instantiation matches a variable, DIGU *propagates* an existing identity into it when both of the following two properties hold:

   (a) The element is in a condition *that matched a subgoal WME*, i.e. one created by an existing instantiation in the subgoal.

   (b) The corresponding element from the existing instantiation is also a variable.

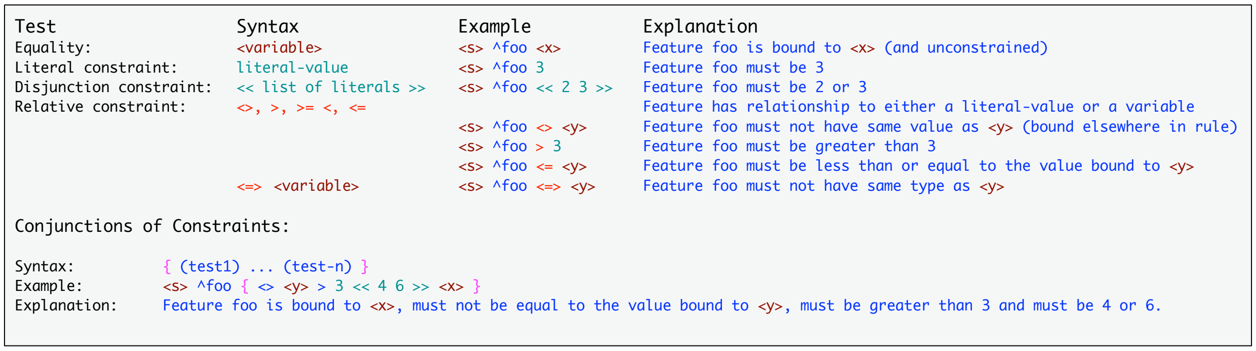
   This is one of the basic cases of shared identity described in Section 5.6. If a RHS action of one rule creates a subgoal WME and a LHS condition of another rule tests that same WME, then we know the variable in the condition refers to the same object as the variable in the action of the other rule. As a result, those elements will be mapped to the same identity. This effectively propagates *some* of the identities forward through the behavior trace.

The careful reader may notice that rule 2 may conflict with other rules. For example, consider the case when the same variable appears in two different conditions, but they both match different subgoal WMEs. While rule 2 insists that both variables must be mapped to the same identity, rule 4 says that each one should be mapped to the different propagated identities from the two different subgoal WMEs. In such conflicts, which are fairly common, rule 1 is always enforced and propagation is ignored. This is why we said that only *some* identities are propagated forward.

During the second phase of identity analysis, which is called *identity resolution* and occurs during the learning episode, EBBS performs a backward traversal of a behavior trace, re-examining all of the condition-action pairs. As it performs this traversal, ignored propagations will be enforced by manipulating the identity graph. (Note that an element's identity never changes after it is initially assigned. Its effective identity may be changed if the identity graph is manipulated, but it will always be mapped to the same identity node.)

5.6.2 Adding Identity-Based Constraints

The rules underlying a behavior trace may impose constraints on the values that the variable within the rules can hold. Figure 5.8 shows all legal constraints that can be specified in Soar and their syntax.



```
Test                     Syntax                 Example             Explanation
Equality:                <variable>             <s> ^foo <x>        Feature foo is bound to <x> (and unconstrained)
Literal constraint:      literal-value          <s> ^foo 3          Feature foo must be 3
Disjunction constraint:  << list of literals >> <s> ^foo << 2 3 >>  Feature foo must be 2 or 3
Relative constraint:     <>, >, >= <, <=                            Feature has relationship to either a literal-value or a variable
                                                <s> ^foo <> <y>     Feature foo must not have same value as <y> (bound elsewhere in rule)
                                                <s> ^foo > 3        Feature foo must be greater than 3
                                                <s> ^foo <= <y>     Feature foo must be less than or equal to the value bound to <y>
                         <=> <variable>         <s> ^foo <=> <y>    Feature foo must not have same type as <y>

Conjunctions of Constraints:

Syntax:         { (test1) ... (test-n) }
Example:        <s> ^foo { <> <y> > 3 << 4 6 >> <x> }
Explanation:    Feature foo is bound to <x>, must not be equal to the value bound to <y>, must be greater than 3 and must be 4 or 6.
```

Figure 5.8: All types of tests that can be specified for an element of a Soar condition.

Our learning algorithm must consider these constraints in its analysis of the trace. Otherwise, it may generate rules that can apply to situations where those constraints are not met. Since such rules could fire in situations in which previous sub-state reasoning could not have occurred, they are overgeneral. The obvious solution is to track those constraints and make sure they are enforced in the learned rule. To this end, EBBS stores within each instantiation all constraints that were required for the underlying rule to match.

These constraints are stored relative to the identities used within the instantiation. For example, if a variable <x>, which was mapped to identity node 1 in some condition, had the constraint < 7, then EBBS would record an annotation in that rule that says it requires that identity 1 must always have a value less than 7. Similarly, if a rule required that a variable <x> mapped to identity node 1 had the constraint <> <y>, which is mapped to identity node 2, EBBS would record that this instantiation requires that the identity node 1 cannot have the same value as identity node 2. Assuming that our generalization algorithm properly determines the most general unification of the trace, these constraints are guaranteed not to conflict. (If a logical conflict truly existed in the trace, then the WME that matched the rule would not have matched. Moreover, if EBBS did learn a rule with such a conflict because of some flaw in the algorithm, the RETE algorithm would immediately tell EBBS that it learned a rule that does not currently match despite being based on the current contents of working memory.)

This concludes our discussion of the two acquisitional components that arose from applying strategy 1 to the behavior trace. How EBBS integrates the identity and constraint information into its summary of the behavior trace is discussed in subsequent sections. Specifically, Sections 5.7.4 and 5.7.6 discuss identity analysis and identity-based variablization, the integrative components of identity generation and propagation, while Section 5.7.5 and 5.7.6 discuss constraint collection and constraint enforcement, the integrative components of identity-based constraints.

5.7 The Learning Episode

Learning opportunities occur immediately after a rule matches in a subgoal. So, any time EBBS records an instantiation of a rule match, it checks to see whether it should learn a new rule. If all the working memory structures created by the match are only accessible to the subgoal, then normal Soar processing continues and no rule is learned. But, if the match creates results – it makes working memory structures accessible to the supergoal – EBBS attempts to learn a rule, and the learning episode begins.

5.7.1 Calculating the Full Set of Inferences

Since the core goal of Soar's online procedural learning algorithm is to summarize how subgoal reasoning produces new inferences in the supergoal, it must first determine which inferences were created in the supergoal. In all the of the examples discussed so far, this determination is straightforward because all supergoal data structures were generated by a single rule. Consequently, the algorithm can simply collect any right-hand side actions of the rule that generate supergoal data structures.

While the data structures produced by a single rule firing are the *initial basis* for creating the actions of a chunk, other subgoal knowledge may also become accessible to the supergoal as a consequence of that rule firing. This occurs when subgoal reasoning incrementally builds a data structure in the subgoal. For example, consider a case where a set of subgoal rule firings create a complex working memory structure that is accessible only to its subgoal. Afterwards, another rule fires that links *only* the top element of that structure to another working memory element that is accessible to the supergoal. Obviously, all of its children also become accessible to the supergoal *even though they are not explicitly referenced in the rule that linked the knowledge to the supergoal*.

To handle this, chunking must first determine the entire structure that becomes accessible after the rule finishes firing. It does this by examining each of the working memory elements that will be created by the right-hand side of the rule. If any of them contain a reference to an existing object in the subgoal, i.e. their value element is a Soar identifier, the algorithm collects all children working memory elements rooted at that Soar identifier. This process is recursively repeated until a complete description of the knowledge structure that the matched rule makes accessible to the supergoal is built.

5.7.2 Backtracing and the Three Types of Analysis Performed

When learning a new rule, EBBS performs an analysis of a subgoal behavior trace using a process called *backtracing*. Backtracing works as follows. Starting with the instantiated rule that creates a subgoal result, backtracing iterates through each working memory element that matched a condition. If the working memory element is local to the subgoal, then backtracing recursively examines the instantiation that created that WME. If the working memory element is a supergoal WME, then that path of backtracing's exploration terminates. Thus, backtracing examines the behavior trace by traversing the instantiation graph backwards through all subgoal rules that contributed to a supergoal inference.

This description is a bit of a simplification. As described in Section 5.7.1, there may be additional structures created by previous rule firings that are also indirectly results. So, while it may seem that backtracing starts its traversal of the behavior trace only through the conditions of *the single rule* that created the initial result, it must also consider the rules that created the children results. In other words, backtracing may have multiple starting points: the conditions of the initial rule firing and the conditions of each rule that previously fired and created children substructures.

The following three sections will discuss the three core aspects of the EBBS's analysis that occur during backtracing: dependency analysis, identity resolution and constraint collection.

(Please note that an important mechanism to handle overgenerality, selective unification, also occurs during backtracing but will not be discussed until Section 5.8. Moreover, several of the mechanisms that mitigate correctness issues also occur during backtracing, but, for the sake of clarity, we will leave those aspects out until Chapter 6.)

5.7.3 Dependency Analysis

The goal of dependency analysis is to determine which conditions in the behavior trace tested working memory elements accessible to the supergoal. As EBBS is backtracing through each rule that fired in the subgoal, it collects all supergoal WMEs that the rule tested. Once the entire behavior trace is traversed, EBBS has determined exactly what supergoal knowledge was tested during the reasoning in the subgoal.

The manner in which EBBS backtraces through a trace to determine supergoal knowledge dependency has not changed much from classical chunking, but there is one key aspect that is different. In classical chunking, if the algorithm encounters more than one condition that matches a supergoal WME, it would only add the first one it encountered, ignoring all others. This made sense for classical chunking because it knew that its heuristic for generalization would always generate two identical, conditions based on the same WME. In contrast, EBBS generalizes by capturing the generality in the underlying reasoning, so, if the reasoning behind each of the two conditions is different, EBBS must consider them separately, despite the fact that they both matched the same WME. Consequently, EBBS adds an additional copy of the supergoal WME each additional time it is tested in the trace.

If the underlying rule has a negated condition that tests the absence of supergoal knowledge, they must also be collected in the dependency analysis. This can be problematic to determine because they do not have corresponding matching WMEs; they are obviously testing that a WME does not exist. To incorporate the constraints imposed by these conditions and avoid overgenerality, both classical chunking and EBBS incorporates them into the online procedural learning process in the following way. As the algorithm backtraces through the trace, it collects the set of all negated conditions seen in the underlying rules. It takes those negated conditions and replaces any variables that also appear in a positive condition with their matched values. In other words, it instantiates as much of the negative conditions as possible. This allows it to determine which negative conditions it encounters are testing lack of knowledge in the supergoal. If the negative condition tests lack of knowledge in the subgoal, it is not included in the chunk. If the negative condition tests lack of knowledge in the supergoal, EBBS records it and then adds it to the chunk during the rule formation step.

5.7.4 Identity Analysis Phase 2: Identity Resolution

To review, identity analysis, the process of determining which elements in a trace share the same identity, occurs in two phases. Section 5.6.1 describes how the first phase, identity creation and propagation, is performed when rules fire and instantiations are recorded. As discussed in that section, propagation provides only partial unification and may lead to overgeneral rules. One reason already mentioned is that conditions may have conflicting identities propagated forward, one of which was ignored. But there are other, more complicated interactions between multiple rules that forward propagation also cannot handle.

To refine the partial unification performed by the initial forward propagation phase, a second phase of identity analysis, which we call *identity resolution*, is performed during backtracing. To do this, EBBS traverses the behavior trace backwards and re-examines each condition-action pair between instantiations. Just like it did when creating the instantiation, it compares the elements of each condition with the corresponding element in the action that produced the WME that matched the condition. EBBS then manipulates the identity graph based on whether the corresponding elements are mapped to the same identity.

Figure 5.9 shows how the identity graph initially contains a node of each identity used in the behavior trace. Each node has a single directed *super-join edge* that initially points back to itself.

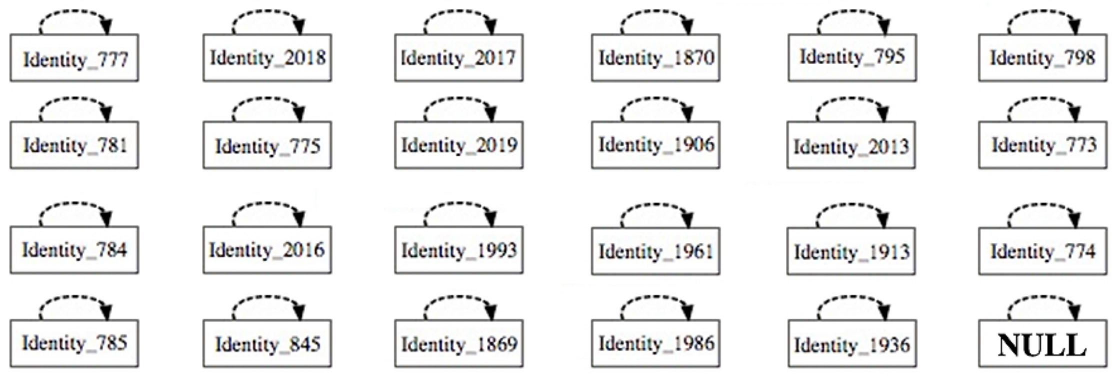


Figure 5.9: An identity graph before identity resolution.

EBBS chooses one of three options when it compares corresponding elements:

1. **Ignore**
   If the identities are already the same, propagation has already done the work and no identity graph manipulation is necessary. This is the most common case.

2. **Join**
   If the identities of corresponding elements in condition-action pair are both variables but are mapped to different identities, EBBS performs a join operation between the two identities. This assigns one identity as the *joined identity* and points the super-join edges of the other identity and any previously joined identities to the new *joined identity*.

3. **Specialize**
   If a behavior trace has a condition-action pair with one element that is a variable and one that is a literal, the reasoning described could not have occurred unless that variable matched that specific value. As a result, EBBS selectively specializes the identity associated with that variable element by pointing its super join edge to the NULL identity. This means that any conditions in the final chunk that have elements mapped to that identity will be treated like a constant and will not be variablized. Instead, the matched value will be used for that element. To differentiate this specific operation on an identity from a more general notion of specialization, we call this form of selective specialization *literalizing* an identity.

After the behavior trace is traversed, all necessary unification will have been performed. Even though the initial identity mappings in the instantiation do not change, the identity graph indicates exactly which identities are always reasoned about generally and which identities are shared for that behavior trace. Figure 5.10 shows a visualization of Figure 5.9's identity graph after a learning episode:

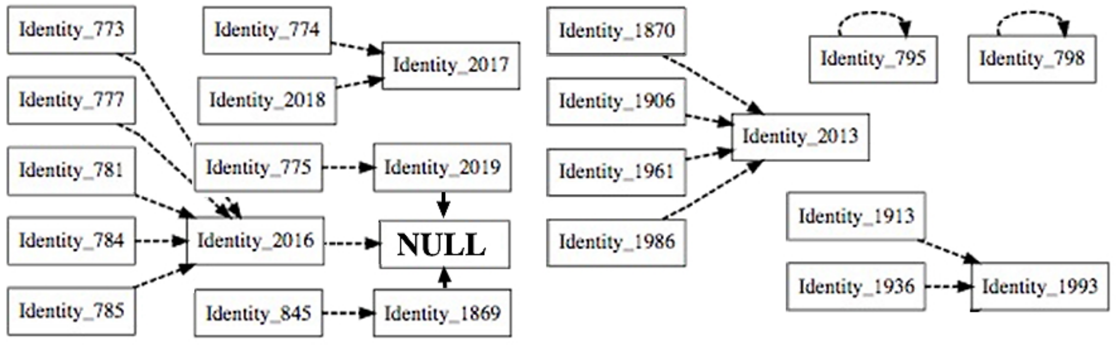


Figure 5.10: An identity graph after identity resolution.

5.7.5 Constraint Collection

Section 5.6.2 described how EBBS annotates instantiations with a list of all variables constraints that were met when the rule matched. While backtracing, EBBS collects all of the constraints it encounters in each rule.

Some of these constraints may not be necessary in the learned rule being built. EBBS only needs a constraint if it involves identities that appear in the learned rule. Unfortunately, EBBS cannot determine which identities are needed until backtracing is complete, so it must collect all of them. Once backtracing is complete, EBBS iterates through the constraint list and prunes any that do not involve identities that appear in the conditions of the new rule.

The identity-based nature of this system allows EBBS to record constraints without worrying about which identities will eventually be unified. It also allows it to handle tricky aspect like *conditions that constrain subgoal knowledge which then transitively place constraints on supergoal knowledge matched*. This occurs when the identity of a constrained element matching a subgoal WME is later joined with the identity of one matching a supergoal WME.

5.7.6 Rule Formation

At this point, DIGU has completed both phases of identity analysis and collected all constraints required by a behavior trace. This section provides a brief overview of how EBBS uses that information to build a rule that captures the generality of the underlying reasoning. Note that, in all steps of rule formation, EBBS takes advantage of identity resolution by always using an element's *effective identity*, which is quickly referenced through its super-join edge.

The rule formation component consists of eight distinct, sequential stages:

Step 1. Condition and Action Creation

This stage creates the initial basis for the condition and actions of the rule.

To form the conditions of the learned rule, EBBS creates copies of all supergoal WMEs that were collected during dependency analysis of the behavior trace. Each element in these copied WMEs is a literal value along and an annotation that map the element to an initial identity.

To form the actions of the learned rule, EBBS creates copies of the actions that produced each of the result and all children of those results that came along for the ride. Like conditions, all elements in an action are literal values along and an annotation that map the element to an initial identity.

Initially, all elements of both the conditions and actions are literal values.

Step 2. Identity-Based Constraint Enforcement

This stage adds all constraints that were collected during backtracing. As previously described, each constraint is indexed in terms of the identity it constrains. So, *if the identity being constrained exists in one of the conditions of the learned rule*, EBBS will enforce the constraint by adding a new constraint to that condition.

This component replaces chunking's limited constraint tracking mechanism. While classical chunking did not annotate the rule instantiations with any constraints, it did keep a log of all cases of one specific type of constraint it found while backtracing: inequalities between Soar identifiers. Because chunking only variablizes Soar identifiers, it only needs to enforce constraints that can be placed on Soar identifiers, which are inequalities. In contrast, EBBS's new constraint tracking system is much more comprehensive and tracks all types of constraints on all types of matched values. This includes literal constraints on the value a single identity, for example `>= 0`, `< 23`, relational constraints between two identities, for example `> <min>`, `< <max>` or `<> <other>` and conjunctions of constraints.

This does introduce some new challenges; for example, one situation in which attaching a constraint can be tricky occurs *when the constrained identity has been literalized but the constraint itself refers to an identity that has not been literalized*, for example `{ > <x> = 3 }`, which says that a feature must be greater than `<x>` and equal to 3. Even though the constraint references an element that can only match a value of `3`, EBBS cannot simply throw the constraint out. The relationship between `3` and the identity of `<x>` must still hold (assuming `<x>` appears in a different test somewhere else in the rule). Since this constraint needs to be enforced to ensure a correct rule, *EBBS inverts the constraint and attaches it to a variable in another condition*. In this example, it would add a `< 3` to some other condition with an element that had `<x>`'s identity.

Step 3. Identity-Based Generalization

To build rules that can match a range of situations, some of the literal elements in the conditions and actions must be replaced by variables; otherwise, the new rule will only ever fire when the exact same objects and features are matched.

Because both phases of identity analysis are complete at this point, all of the real work needed to determine the generality of the underlying reasoning has already been performed. So, this step replaces the elements in the rule that the agent always reasoned about generally with a variable. Specifically, this step iterates through each condition and action of the rule and examines the effective identity that each element is mapped to. Based on that identity mapping, EBBS does one of two actions:

- If the element is mapped to the NULL identity, its literal value is left intact.
- If the element is mapped to an identity, EBBS replaces the element with the variable associated with that identity. (It creates new variables and assigns them to identities as needed during rule formation.)

This process is also applied to both the negative conditions and the constraint tests that were attached to conditions in step 2.

Step 4. Condition Merging

If two conditions in the learned rule share the same identities in their identifier, attribute and value elements, they can be combined. In such cases, it is logically impossible for those two conditions to match two different WMEs and cause the same rules to match and same conditions to unify in the subgoal. As a result, EBBS can safely merge those two conditions without losing generality.

Note that the two conditions may have different constraints, in which case the constraints are merged. These constraints are guaranteed not to conflict for the same reasons cited in 5.6.2.

Step 5. Condition Simplification

EBBS refines the conditions of the learned rule by pruning unnecessary constraints on literalized elements and replacing multiple disjunction constraints with a single simplified disjunction.

1. **Merge disjunction constraints:** If an element in a condition has two disjunction tests, the constraints will be merged into a single disjunction that contains only the shared values. `{ «`

`a b c » « b c d » <x>}` becomes `{ « b c » <x> }`, because it is impossible for `<x>` to be either `a` or `d`. This will also eliminate any duplicate disjunctions.

2. **Eliminate unnecessary constraints on literals:** If an element in a condition has been literalized but also has a literal constraint on its value, then the constraint is unnecessary and will be discarded. For example, `<s> ^value { < 33 23 }` becomes `<s> ^value 23`.

   This step also eliminates duplicate constraints.

Step 6. Rule Validation and Repair

At this point, the rule is formed. EBBS must now make sure that the learned rule can be legally added to production memory. Specifically, this step looks for malformed rules that have one or more dangling conditions or actions, i.e. ones that are not linked to a goal state in the rule. Rule validation was first developed as a way to avoid a number of correctness issues that could result in malformed rules. Since those issues were not well-understood, the validation step was added as a failsafe to prevent bad rules from being added to production memory. It was also used as a way to alert the agent designer that they might be facing one of the chunking issues that were known about at the time.

While the rule validation algorithm is essentially unchanged from classic chunking, what happens as a result of that validation has indeed changed. Classical chunking throws out any learned rules that cannot be validated. In contrast, EBBS instead repairs the malformed rule by searching working memory for missing dependencies that could link the dangling conditions or actions. We describe the details of rule repair in Section 6.1.3.1.

While rule validation is now essentially a way to detect when rule repair is needed, we still call it rule validation. This is because online procedural learning problems often manifest as rules with one or more conditions or actions that are not properly connected to a goal. If new correctness issues are discovered or introduced in the future (we are wrong about existing correctness issues), this rule validation step is still the most likely way the problem will be detected. In that sense, it still serves as validation. If repair fails[7], EBBS does indeed discard the invalid rule just like classical chunking.

Step 7. Condition Re-ordering

Since the efficiency of Soar's RETE rule matching algorithm depends heavily upon the order of a production's conditions, both classical chunking and EBBS attempt to greedily sort the chunk's

[7]There are no known situations where repair currently fails.

conditions. At each stage, the condition-ordering algorithm tries to determine which eligible condition, if placed next, will lead to the fewest number of partial matches. For example, a condition that matches an object with a multi-valued attribute will lead to multiple partial matches, so it is generally more efficient to place these conditions later in the ordering.

Step 8. Add Rule to Agent's Procedural Knowledge and Recursively Learn Additional Rules

Now that the rule has been built, validated, and optimized, it is added to the agent's production memory. This only fails if the new rule is a duplicate of an existing rule.

It is quite common that a learned rule will immediately generate another learned rule. When a problem has been decomposed *into two or more subgoals*, a single result can produce a chunk for every intermediate subgoal. The reason for this process is a bit complex. Once the rule is in production memory, it is treated like any other. Because it was just learned and is based on the current state of working memory, its conditions are guaranteed to immediately match in the supergoal. Moreover, the chunk has made new data structures accessible to the goal it matches in, so it needs to also record which knowledge structures in that goal that those results are dependent on. Otherwise, future learning episodes will not be able to backtrace through the newly created structures. So, EBBS creates an instantiation for the chunk match that includes new identity mappings relative to the identity graph of the supergoal. This provides that explanatory knowledge necessary to learn new chunks based on the chunk just added.

This process occurs whenever the goal that the instantiation was added to is itself a subgoal. If the chunk has at least one action that makes WMEs accessible to an even higher-level goal, EBBS will backtrace through the chunk match and learn a second chunk based on the reasoning in the subgoal that the chunk matched in, which was the supergoal of the subgoal it was learned from. This process stops when a chunk's instantiation is only generating working memory elements in the same state that it matches in.

5.8 Unnecessary Generality

Our initial implementations of this algorithm generated surprisingly convoluted rules that were expensive to match. This even occurred when summarizing simple, almost trivial behavior traces. For example, the chunk in Figure 5.11 summarizes a behavior trace that consists of three rule firings, each of which tests a single feature in the supergoal. Instead of learning a rule that also tests three features in its supergoal, our system instead learned a rule that tests three features in three, possibly different, supergoals.

```
sp {chunk*substate-result-rule*t1-1
    (state <s1> ^superstate nil)
    (state <s2> ^foo <f1>)
    (state <s3> ^foo { <f2> <> <f1> })
    -->
    (<s1> ^result |Two foos!|)
}
```

Figure 5.11: A learned rule with unnecessary overgenerality.

These unexpected rules led to a surprising insight about the generality of an agent's underlying rules: *determining the most general unification of a trace always produced unnecessarily general rules*.[8]

This unnecessary generality arose from the fact that the initial EBBS implementation did not consider in its analysis an implicit constraint that the architecture imposes on working memory. As discussed in Section 3.1, one of the few constraints on Soar's working memory is that all knowledge is linked to goals and those goals are organized in a stack linked by a ^superstate feature. Consequently, every subgoal can only have one supergoal at a time in Soar; in other words, if we have (`S5 ^superstate S3`), we cannot also have (`S5 ^superstate S1`) at the same time. We call features that are guaranteed to always have a single value *singletons*. In contrast, non-singletons can have multiple values for a feature, for example (`S5 ^block A ^block B`).

This problem we describe in this section is the first example of a learning issue that arises from the use of implicit knowledge in the design of an agent's rules. The singleton nature of Soar's superstate link allows agent engineers to safely design simpler rules, each of which assumes it is working in the service of the same supergoal as the other rules in that subgoal.[9] There is no reason for the agent engineer to include constraints in their rules for knowledge structures that are guaranteed not to exist. An online procedural learning algorithm that is unaware of this uniqueness property and generalizes based purely on the underlying rules, like our initial implementation, learns rules that are so flexible that every independent reference to a supergoal could be a reference to a different supergoal, something that cannot occur in the Soar architecture. Since these rules become more complex with every reference to a singleton feature, they quickly become prohibitively expensive to match.

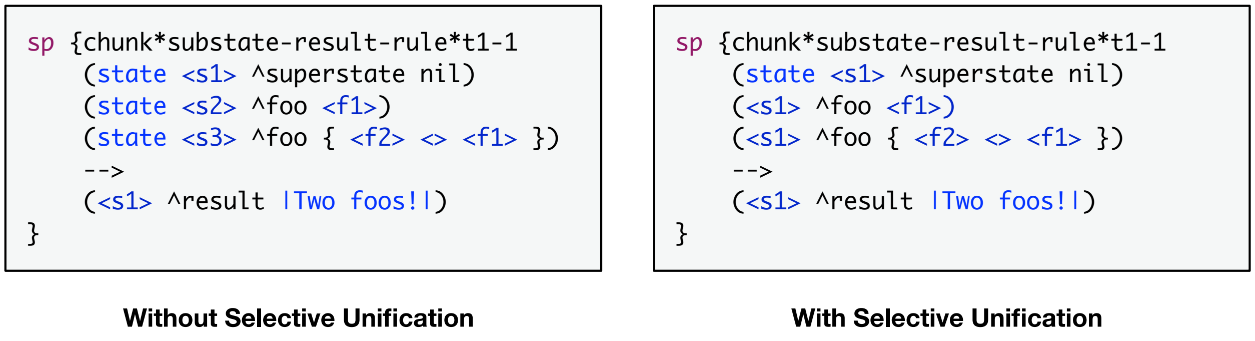
[8]For this section, we are not using our relative definition of generality. Rather, we mean a rule that is more abstract and covers a larger space of situations.

[9]In contrast, if a subgoal could have multiple supergoals, engineers would need to add extra logic in each rule to ensure that it is making an inference on the same supergoal knowledge as another rule in the subgoal.)

5.8.1 Applying Strategy 3: Selective Unification of the SUPERSTATE feature

EBBS eliminates this inherent overgenerality by incorporating the uniqueness of each subgoal's superstate feature into its analysis using a technique we call *selective unification*.

Selective unification works as follows: while backtracing, if EBBS encounters two conditions in a trace that test the same working memory element and that WME involves a feature that is guaranteed to have a single unique value, it will join the corresponding identities of any variables in the two conditions. This forces all references to the feature to be considered as references to the same feature, despite the fact that the underlying rules do not actually test that sameness. With the addition of the singleton mechanism, EBBS eliminates the inherent overgenerality of Section 5.8 by flagging every subgoal's superstate feature as a singleton. Figure 5.12 shows the effect that selective unification has on a chunk learned.



```
sp {chunk*substate-result-rule*t1-1
    (state <s1> ^superstate nil)
    (state <s2> ^foo <f1>)
    (state <s3> ^foo { <f2> <> <f1> })
    -->
    (<s1> ^result |Two foos!|)
}
```

Without Selective Unification

```
sp {chunk*substate-result-rule*t1-1
    (state <s1> ^superstate nil)
    (<s1> ^foo <f1>)
    (<s1> ^foo { <f2> <> <f1> })
    -->
    (<s1> ^result |Two foos!|)
}
```

With Selective Unification

Figure 5.12: A simple chunk with and without selective unification.

The superstate singleton is arguable the only one that the algorithm absolutely must handle, EBBS also treats several other architecturally created features as singletons:

```
(<goal> ^superstate <any>)
(<goal> ^operator   <operator>)
(<goal> ^type       <constant>)
(<goal> ^impasse    <constant>)
(<goal> ^smem       <identifier>)
(<goal> ^epmem      <identifier>)
```

5.8.2 Domain-Specific Unnecessary Generality

It is often the case that agent reasoning also assumes domain-specific object or feature uniqueness that is not reflected in the rules. This can also lead to learned rules with unnecessary generality. For example, in an agent that solves a BlocksWorld puzzle, the agent has rules for manipulating a gripper that picks up and places the blocks. The rules of the agent assume there is only one gripper and references it accordingly. When EBBS attempts to learn rules from this agent's reasoning, it is not aware of the implicit knowledge that the designer had and generates unnecessarily complex rules where each reference to a gripper could be a different gripper.

EBBS handles such learning situations by providing a mechanism to specify domain-specific singletons. Since we cannot determine a priori which objects or features in a particular domain can have multiple values and which ones are singletons, EBBS allows the agent designer to specify it themselves. In other words, if an agent designer knows that certain features are guaranteed to never have more than one value, they can give EBBS the meta-knowledge that it must always treat references to those features as singleton.

For example, to improve the gripper-related rules learned in the BlocksWorld agent, we specify that any WME of the form `[goal] ^gripper [soar identifier]` is a singleton. This unifies the identities of any conditions that reference a gripper in a behavior trace. Consequently, any rules learned by summarizing that trace will only reference a single gripper.

This mechanism is an application of both the strategy 1, acquiring and integrating missing knowledge into the online procedural learning algorithm, and strategy 3, selective unification. Specifically, EBBS uses user-defined singletons to allow agent engineers to provide a specification of one form of implicit knowledge they used in their rules, namely that a feature of this task can only hold one value at a time. Practically speaking, this feature has proven to be powerful if not essential. In many agents, we find that we must define domain-specific singletons to achieve efficient matching of rules.

5.9 Learning from Non-Learning Subgoals

While many agents use chunking to learn rules every time a subgoal returns a result, in complex real-world system, an agent designer may want to learn rules from some subgoals but not others. For example, they may not want to learn rules in a subgoal that they know uses reasoning that chunking cannot correctly summarize, for example in a subgoal that chooses between operators probabilistically. Or they may not want to learn rules in a subgoal that they know uses reasoning that is so abstract or so specific that it will generate a chunk that will provide little utility. For cases like these, Soar provides the capability to selectively enable or disable online procedural learning in a subgoal. Note that these *non-learning subgoals* generate the same exact knowledge structures in supergoals as their learning counterparts. The difference is that a rule is not learned that could generate the structure in the future.

Soar does perform some analysis of the reasoning that occurred in the non-learning subgoals. When learning is disabled, another type of rule called a *justification* is created instead of a chunk. Justifications are transient, architecturally-created rules that allow Soar to ensure that the knowledge structures created in the supergoal only persists as long as they should.

In some ways, justifications are just like chunks. For example, Soar performs the exact same dependency analysis for justifications that it does for chunking to determine which knowledge structures in the supergoal were tested by the subgoal reasoning. Soar performs this analysis to determine whether the working memory elements that are being created in the supergoal should be either persistent ones or context-dependent ones. If the justification tests an operator in the supergoal, Soar knows a result should be persistent (o-supported) and remain in working memory until the agent removes it. If the justification does not test an operator, Soar knows the result is context-dependent (i-supported) and should only stick around while the context of the justification holds. So, it adds the justification to procedural memory; when it no longer matches, Soar will know that it must remove the working memory elements that it created. Justifications are created every time a non-learning subgoal creates a result in a supergoal.

In other ways, justifications are very different from chunks. For example, they are fully specified: they contain absolutely no variables, not even for Soar identifiers, or RHS functions. So, in most cases, a justification looks exactly like its own instantiation. This means that a justification does not encode any indication of how general the underlying reasoning was that led to the result. Without this information, justifications cannot be used, at least in their current from, within an EBBS behavior trace. If chunking attempts to later summarize reasoning in the learning goal, it will be unable to capture the generality encoded in the rules that led to the matched knowledge that came from the non-learning subgoal. To learn an optimally general rule, EBBS must be provided with information that specifies, not only any dependencies that the result has to other knowledge structures, which justification do provide, but also which aspects of those dependencies were reasoned about generally and what constraints there are on that generality, which justifications do not provide.

5.9.1 Applying Strategy 1: Explanatory Justifications

To address this problem, EBBS augments justifications with the same explanatory power that chunks have. It does this by performing the same analysis in non-learning goals that it does in learning goals and adding explanatory meta-information to justifications. EBBS does the following two tasks to create justifications that eliminate the key differences:

1. While the elements of every condition and action in a justification still contain the literal values that matched, they are annotated with an identity that was determined using the same identity creation/propagation rules that chunks use.
2. Each element of the justification is annotated with all constraints placed on the value during reasoning.

These changes allow EBBS to acquire and integrate the missing knowledge that caused this correctness issue and learn optimally general rules from justifications as readily as it can from hand-written rules and other chunks.

5.10 Concluding Comments On Generality

In this chapter, we described how EBBS applies the first three strategies, acquire and integrate missing knowledge, selective specialization and selective unification, to learn rules that capture the generality of the agent's underlying reasoning without unnecessary overgenerality.

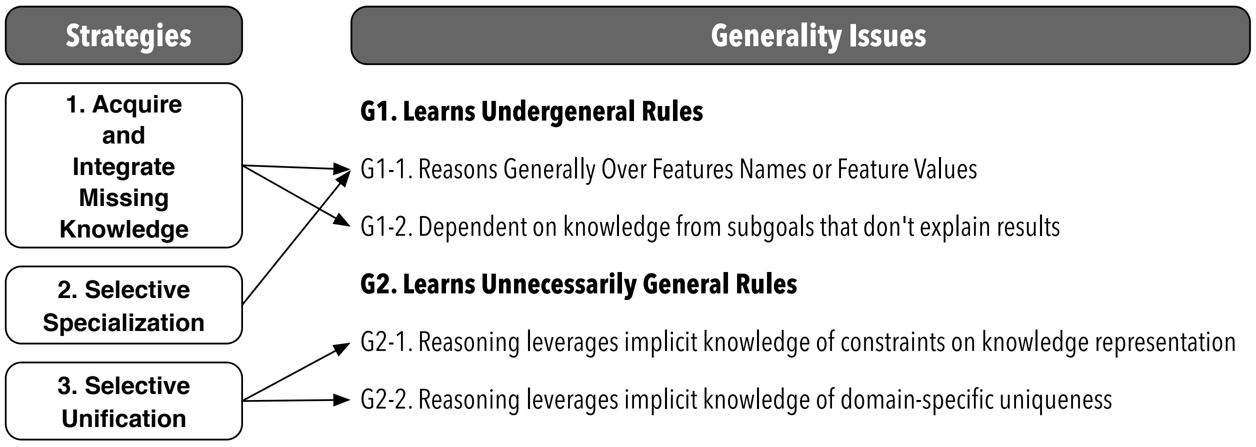

Strategies
Generality Issues
1. Acquire and Integrate Missing Knowledge
2. Selective Specialization
3. Selective Unification
G1. Learns Undergeneral Rules
G1-1. Reasons Generally Over Features Names or Feature Values
G1-2. Dependent on knowledge from subgoals that don't explain results
G2. Learns Unnecessarily General Rules
G2-1. Reasoning leverages implicit knowledge of constraints on knowledge representation
G2-2. Reasoning leverages implicit knowledge of domain-specific uniqueness

Figure 5.13: Three strategies applied to the generality issues of chunking.

Chapter 6 similarly describes how the strategies are applied to mitigate the 12 correctness issues Soar has performing online procedural learning.

CHAPTER 6

Learning Correct Procedural Knowledge

Traditional explanation-based systems learn new procedural knowledge by generating one or more explanation traces that describe how a specific example could lead to a desired or known proposition. In contrast, a procedural learning algorithm in a cognitive architecture learns new procedural knowledge by summarizing a behavior trace of previous agent reasoning that led to a proposition. One of the key ways that this type of behavior summarization differs from learning in other explanation-based systems is that it is possible to learn knowledge that is incorrect and may later lead to unjustified inferences when the learned knowledge is used. We posit that this occurs when a summarization algorithm summarizes reasoning that either (1) has dependencies that are not reflected in the behavior trace, (2) includes reasoning that cannot be expressed in a single rule or (3) includes reasoning that is dependent on unreliable knowledge. The EBBS algorithm uses four strategies to ensure correct summaries of agent reasoning as described in Figure 6.1. This chapter will discuss each of the strategies and show how EBBS uses them to avoid the twelve correctness issues we've identified in Soar's chunking mechanism.

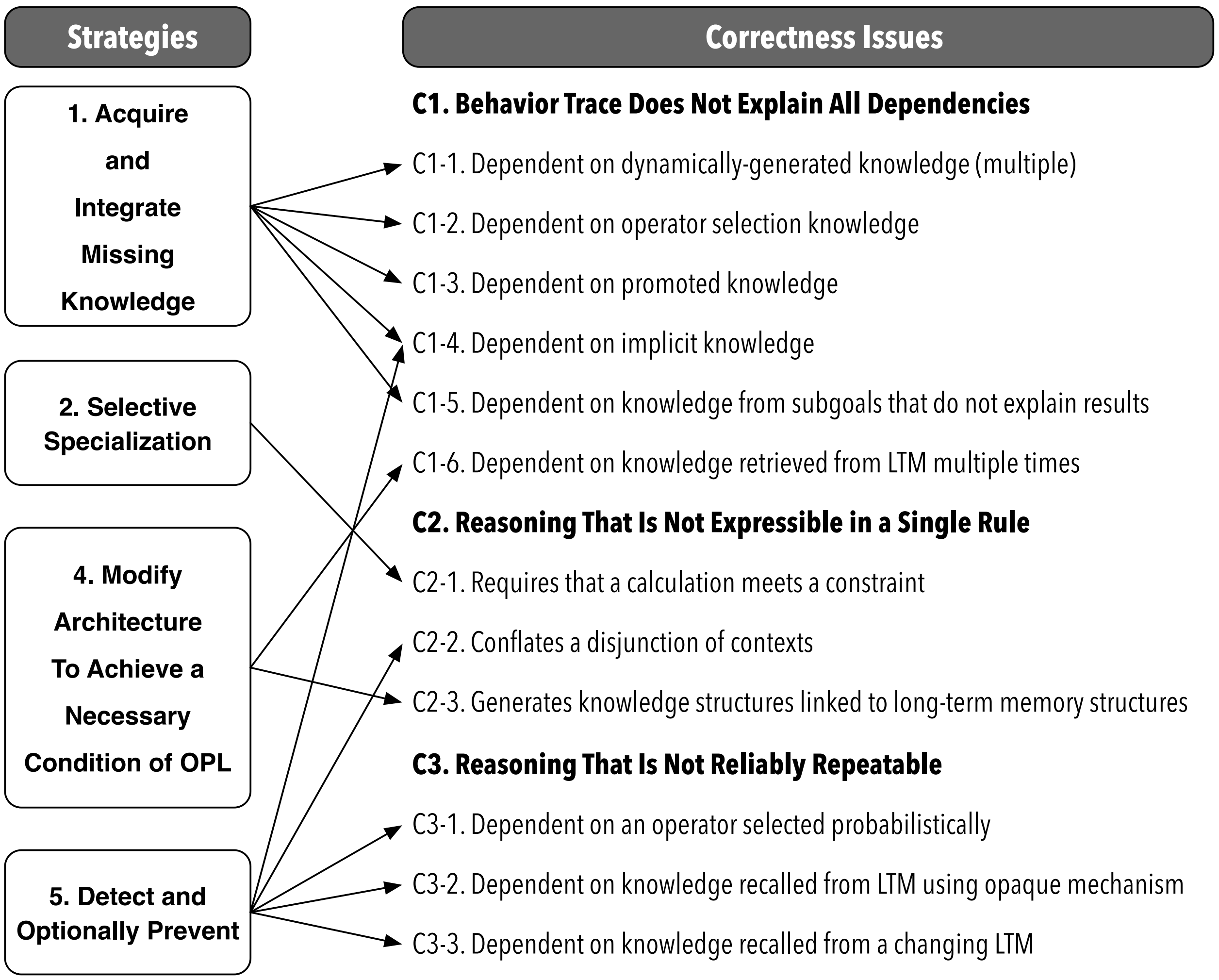

Strategies
Correctness Issues
1. Acquire and Integrate Missing Knowledge
2. Selective Specialization
4. Modify Architecture To Achieve a Necessary Condition of OPL
5. Detect and Optionally Prevent
C1. Behavior Trace Does Not Explain All Dependencies
C1-1. Dependent on dynamically-generated knowledge (multiple)
C1-2. Dependent on operator selection knowledge
C1-3. Dependent on promoted knowledge
C1-4. Dependent on implicit knowledge
C1-5. Dependent on knowledge from subgoals that do not explain results
C1-6. Dependent on knowledge retrieved from LTM multiple times
C2. Reasoning That Is Not Expressible in a Single Rule
C2-1. Requires that a calculation meets a constraint
C2-2. Conflates a disjunction of contexts
C2-3. Generates knowledge structures linked to long-term memory structures
C3. Reasoning That Is Not Reliably Repeatable
C3-1. Dependent on an operator selected probabilistically
C3-2. Dependent on knowledge recalled from LTM using opaque mechanism
C3-3. Dependent on knowledge recalled from a changing LTM

Figure 6.1: Applying the Five Strategies of EBBS to the 12 Correctness Issues.

6.1 Strategy 1: Acquire Missing Knowledge and Integrate

One of the key differences between a behavior trace and an explanation trace is that a behavior trace is an approximate description of all of the factors that contributed to a proposition. If that description is inadequate and does not provide a complete delineation of everything that contributed to the result, an online procedural learning algorithm can learn an overgeneral and hence incorrect rule.

In this section, we discuss one of the more straightforward ways an explanation can be inadequate: missing dependencies. These are cases in which the behavior trace does not describe all necessary dependencies between the agent's knowledge structures and its subgoal reasoning. The following three sub-sections will discuss the three classes of situations we've identified in the Soar architecture in which missing dependencies can occur. In each of them, EBBS acquires the knowledge by building architectural instantiations or augmenting existing instantiations that describe the missing dependencies. It then integrates them into the online learning process by adding the new instantiations to the behavior trace.

6.1.1 Case 1: Dynamically-Generated Knowledge

Knowledge structures in an agent's working memory arise in one of two ways. The first mechanism is the agent's procedural memory: knowledge structures are created when the conditions of a rule in the agent's procedural matches the current state of working memory and transform the knowledge that matched its conditions into new structures.[1] The second mechanism is when the architecture creates knowledge structures dynamically at run-time in response to agent execution. Specifically, Soar creates the following knowledge structures dynamically:

- **Subgoal Meta-knowledge**

  Soar automatically augments the subgoal with several features that describe the nature of the supergoal impasse that gave rise to the subgoal.

- **Long-term Knowledge**

  In response to requests from the agent, Soar creates knowledge structure in working memory based on knowledge structures stored in either its semantic or episodic memory store.

- **Dynamic Actions**

  Cognitive architectures often have special-purpose functions that can be used in an action of a rule to dynamically generate knowledge structures. For example, Soar has a *deep-copy* function that allows an agent to submit a single Soar identifier to create a new knowledge structure that is a copy of the entire sub-graph of working memory rooted at that Soar identifier. Another example of a dynamic action is Soar's math functions, which create new working memory elements by performing mathematical operations on a set of input parameters. Dynamic actions are essentially black boxes to an online procedural learning algorithm.

Solution: Describe Missing Dependencies

[1]The mechanics by which new knowledge structures are created in working memory is a bit more complicated and has an intermediate layer. The actions in a rule create data structures called *preferences* that assert *desired* working memory changes. Those preferences are later resolved into actual working memory changes. This additional layer of processing allows Soar to handle multiple justifications for a working memory element, including ones that may conflict. For simplicity's sake, we ignore preference memory in these discussions unless they play a meaningful role in the problem.

Classical chunking can learn incorrect knowledge in agents whose rules test or create dynamically-generated working elements. In the first two cases, subgoal meta-knowledge and long-term knowledge, the behavior trace does describe the implicit dependencies between dynamically-generated working memory elements and the supergoal structures that they are dependent on. In the third case, dynamic actions, an instantiation of the rules exist but they do not completely describe how the knowledge structures created by the actions are dependent on the contents of working memory from which their values were derived. Without something in the behavior trace that describes these missing dependencies, classical chunking may learn a rule that does not test all of the knowledge structures that gave rise to the reasoning that the algorithm was attempting to summarize and hence is overgeneral and incorrect.

To remedy this deficiency, EBBS creates special *architectural instantiations* for any knowledge structures that are automatically or dynamically created by the architecture. These architectural instantiations describe all dependencies that exist between those architecturally-created WMEs and the supergoal data structures that they are dependent, which allows EBBS to learn knowledge which is not overgeneral.

We will now describe how this approach was applied to Soar's dynamically generated knowledge structures.

6.1.1.1 Case 1 Example 1: Subgoal Meta-knowledge, the `^impasse` and `^item` features

Whenever an impasse occurs that leads to a subgoal, Soar automatically creates knowledge structures in the subgoal to describe the impasse. For example, Soar always creates an `^impasse` feature that indicates which type of impasse occurred. Because most impasses are attempting to resolve a situation where at least one operator is proposed in the supergoal, Soar also creates one or more `^item` features for each operator involved in the impasse. For example, when an agent faces a tie impasse, it has two or more operators are under consideration, so Soar creates an `^item` feature for each one. Similarly, when an agent faces an operator no-change impasse, which occurs when an agent has selected an operator but does not have any rules that will apply the operator in the current situation, Soar creates a single `^item` feature for the stalled operator.

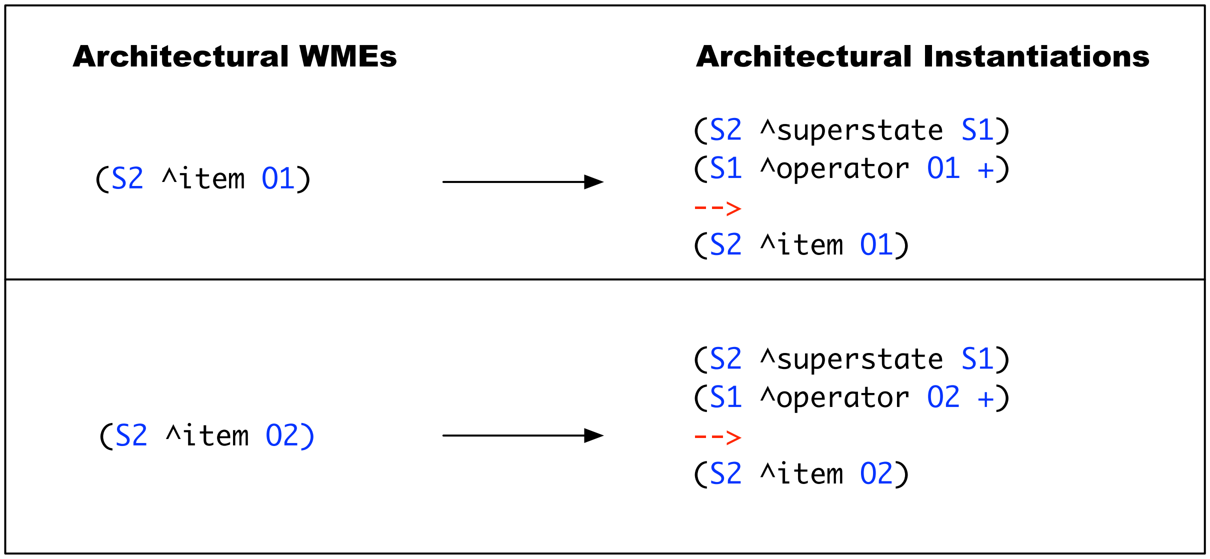
To describe what an architectural instantiation looks like, consider the following example in which an agent had a tie between operators `O1` and `O2` in goal `S1`, which caused Soar to spawn a new subgoal `S2` with the dynamically-generated features:

```
(S2 ^impasse tie)

(S2 ^item O1)

(S2 ^item O2)
```

Clearly, `(S2 ^item O1)` and `(S2 ^item O2)` are dependent on the proposed operators `O1` and `O2` in the superstate.

Solution

To describe these dependencies, both EBBS and classical chunking add architectural instantiations that explicitly describes the dependencies behind each of those features.[2] Figure 6.2 shows the architectural instantiations created for each of the three aforementioned dynamically-generated subgoal features.[3]



```
(S2 ^item O1)      ──────►   (S2 ^superstate S1)
                             (S1 ^operator O1 +)
                             -->
                             (S2 ^item O1)
```

```
(S2 ^item O2)      ──────►   (S2 ^superstate S1)
                             (S1 ^operator O2 +)
                             -->
                             (S2 ^item O2)
```

Figure 6.2: Architectural instantiations that explain subgoal meta-knowledge.

6.1.1.2 Case 1 Example 2: Semantic and Episodic Memories Retrievals

In this example, we describe one of the simpler reasons that chunking can learn incorrect knowledge when summarizing reasoning that tests knowledge that was retrieved from a long-term memory store. To do that, though, we must describe some basic of the basic mechanics of Soar's two long-term memory (LTM) systems, semantic memory and episodic memory, and how they differ from short-term working memory (STM or WM).

Semantic memory contains a declarative representation of the agent's long-term abstract knowledge. Soar places minimal constraints on how the knowledge in semantic memory is structured. In contrast to STM, which can be represented as a single directed graph, because all knowledge

[2]The only new addition that EBBS makes is that it generates identities for the elements of the architectural instantiation.

[3]The + symbol in the figure is special syntax that allows a condition to match an operator that is proposed but not selected.

structures are required to be connected, directly or indirectly, to a goal node, the knowledge in LTM is a directed graph that may contain an arbitrary number of disconnected sub-graphs.

Soar's other LTM system, episodic memory, periodically stores a copy of an agent's working memory to the episodic memory store. This provides an agent with a history of its previous experiences to reason over. We will focus on semantic memory, because some the correctness issues in this chapter only involve it.

Semantic memory is composed of long-term working memory elements (LTMEs) which use their own set of transient identifiers called *long-term identifiers* (LTIs), to group features. We will explain long-term identifiers work and why they pose a significant problem for Soar's chunking mechanism to Section 6.3.1. In this section, we will describe how an agent is able to reason over the LTMEs in semantic memory.

As we know from Section 3.1, an agent's rules can only test knowledge structures in relation to a goal or subgoal. Given that constraint, how does an agent reason over the knowledge in semantic or episodic memory whose contents are in a separate system not connected to an agent's goal structures? In short, an agent cannot access the contents of LTM directly. To utilize the knowledge structures in a LTM like semantic memory, the agent must make a request to the architecture, asking it to bring the desired long-term memory into short-term memory where it can be reasoned over by the agent's rules.

Specifically, an agent creates a request by creating a set of WMEs that describe what it wants. It then connects that description to a special, architecturally created WME called the SMEM link that is added to every goal node. The Soar architecture monitors the SMEM link for long-term memory queries. When Soar finds one, it will attach a set of WMEs that correspond to the semantic knowledge requested to the SMEM link. Once the long-term knowledge is in working memory, the agent's rules can then match against the knowledge producing new inferences. Through this process, the agent is able to reason, albeit indirectly, on the content of its semantic memory.

There are two types of semantic memory requests that an agent can make. The first type is a *query*, which is a request that provides Soar with a partial description of the knowledge the agent wants. In response to a query, Soar will attempt to find all knowledge structures in semantic memory that best matches that description. If at least one structure is found that satisfies the query, the best match is then "moved" to working memory. More accurately, Soar creates a set of WMEs that mirror the LTMEs that best matched the query. The second type of semantic memory request is called *a retrieval* and occurs when an agent knows the root node of which semantic memory it wants.[4] In

[4]The agent may have the identifier of a semantic memory node because it previously retrieved that node using a query. It can also occur when the agent previously stored a structure in working memory into semantic memory, which converts each STI into a LTI.

these cases, the agent submits a retrieval command containing the LTI that it is interested in. Soar will then create WMEs in STM that correspond to all of the LTMEs that are rooted in that LTI, i.e. all of its children LTMEs.

Now that we have described the basics of Soar's long-term memory systems, we describe one of the situations where chunking may learn incorrect knowledge. This problem occurs when reasoning in the subgoal tests knowledge that is retrieved from a semantic memory command *also issued in the same subgoal*. (In other words, the agent issues a semantic memory request to the SMEM command link in a subgoal instead of one in a higher-level goal.) The problem is that the structures generated by long-term memory retrievals are dynamically-created by the architecture, so there is no computational accounting of any dependency between the context of the query and the knowledge retrieved. Because the request was made in the subgoal, Soar attaches the LTM structures directly to the subgoal. Because the knowledge is subgoal knowledge, chunking will ignore the conditions that tests the recalled knowledge when it summarizes the subgoal and will not include them as conditions of the chunk. From the online procedural learning algorithm's perspective, a LTM recall is the equivalent of knowledge that just magically appeared in memory for no reason. If an online procedural learning algorithm attempts to summarize reasoning that tests that knowledge without including the necessary conditions for that knowledge to exist in memory, the learned rule may be ignoring necessary aspects of the context, which means the rule can be considered over-general and incorrect.

Solution

Both EBBS and classical chunking handle this problem by adding an explanation for why the semantic memory structures were created in working memory by creating an architectural instantiation for any structures recalled by semantic or episodic memory in a subgoal. These architectural instantiations provide a description of the features that needed to exist to initiate the acquisition of the long-term knowledge that was used in the behavior trace. (If the rationale for the query is dependent on supergoal knowledge – the query would only have been performed if certain conditions in the supergoal were met – then the knowledge retrieved from long-term memory should also be dependent on those conditions.) The left-hand side conditions of one of these architectural instantiations are the WMEs that matched the rule that created the query, and the right-hand side effects of the instantiation are the LTM structures that were added to the subgoal in response to the query. By backtracing through this instantiation, the online procedural learning algorithms is able to learn correct rules that summarize the reasoning that eventually led to the query.

The key difference between the architectural instantiations created by classical chunking and EBBS is that the EBBS version creates the appropriate identities for the elements of the architectural instantiation.

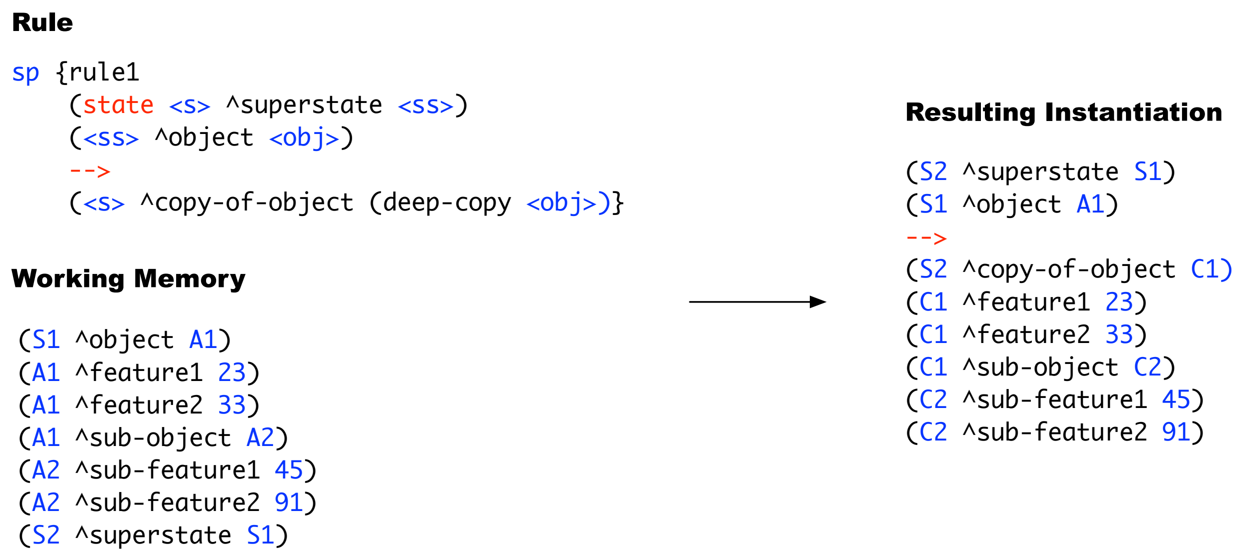
An Architectural Solution

In this section, we explain why we did not pursue another, arguably simpler, solution, which is to modify the architecture so that subgoal retrievals of LTM is not possible. If we remove the ability to retrieve knowledge structures from LTM at multiple levels of the hierarchy and instead had a single SMEM link on the top goal that the agent must use for all semantic memory recalls, this problem simply wouldn't exist. All long-term knowledge would be accessed through the supergoal, so it could never be ignored as subgoal knowledge. So, why did we not do that?

The short answer is that some research projects implement their secondary learning mechanisms using techniques that require subgoal semantic memory recalls. For example, some agents *proceduralize a recall* by retrieving a LTM into a subgoal, testing a cue in the supergoal and then returning the LTM structure as a result. By doing this, the agent learns a rule that, in a future similar situation, can create the LTM structure directly into working memory without paying the computational and time costs of a semantic memory access. Other agents retrieve a knowledge structure from semantic memory directly into the subgoal so that they can use the information in their reasoning without it appearing in the learned rules that summarize the reasoning. In other words, subgoal retrievals of LTMs *provide a way to intentionally exclude conditions from a learned rule*, which is a very powerful capability. This allows researchers to implement a myriad of other forms of secondary, slower online procedural learning mechanisms. In fact, it's so powerful a technique that Stearns (Stearns et al., 2017) was able to use it to implement an entirely different procedural learning algorithm called PRIMS using EBBS as the underlying engine.

6.1.1.3 Case 1, Example 3: Rules with Dynamic Actions

In some sense, rules that use dynamic action are creating a new rule at run-time that is based on structures that exist in working memory. This makes rules with dynamically-generated actions different from the previous two cases. Unlike subgoal meta-knowledge and long-term memory retrievals, they come into existence when an agent rule matches, so they already have a normal instantiation. Consequently, they do not need a new architectural instantiation. For example, Figure 6.3 shows an instantiation that is created when a rule matches that uses the `deep-copy` action to copy a working memory structure in the supergoal.



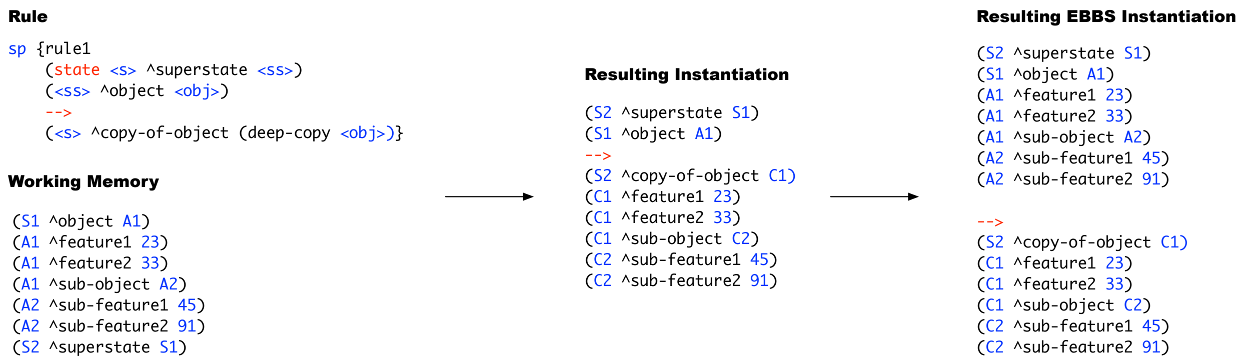
```
sp {rule1
    (state <s> ^superstate <ss>)
    (<ss> ^object <obj>)
    -->
    (<s> ^copy-of-object (deep-copy <obj>)}
```

```
(S1 ^object A1)
(A1 ^feature1 23)
(A1 ^feature2 33)
(A1 ^sub-object A2)
(A2 ^sub-feature1 45)
(A2 ^sub-feature2 91)
(S2 ^superstate S1)
```

```
(S2 ^superstate S1)
(S1 ^object A1)
-->
(S2 ^copy-of-object C1)
(C1 ^feature1 23)
(C1 ^feature2 33)
(C1 ^sub-object C2)
(C2 ^sub-feature1 45)
(C2 ^sub-feature2 91)
```

Figure 6.3: A rule with a dynamic action and the instantiation it leads to.

To explain the missing dependencies and learn correct rules, EBBS modifies the instantiation by adding new conditions that map to each of the dynamically-created actions, as shown in Figure 6.4.



```
sp {rule1
    (state <s> ^superstate <ss>)
    (<ss> ^object <obj>)
    -->
    (<s> ^copy-of-object (deep-copy <obj>)}
```

```
(S1 ^object A1)
(A1 ^feature1 23)
(A1 ^feature2 33)
(A1 ^sub-object A2)
(A2 ^sub-feature1 45)
(A2 ^sub-feature2 91)
(S2 ^superstate S1)
```

```
(S2 ^superstate S1)
(S1 ^object A1)
-->
(S2 ^copy-of-object C1)
(C1 ^feature1 23)
(C1 ^feature2 33)
(C1 ^sub-object C2)
(C2 ^sub-feature1 45)
(C2 ^sub-feature2 91)
```

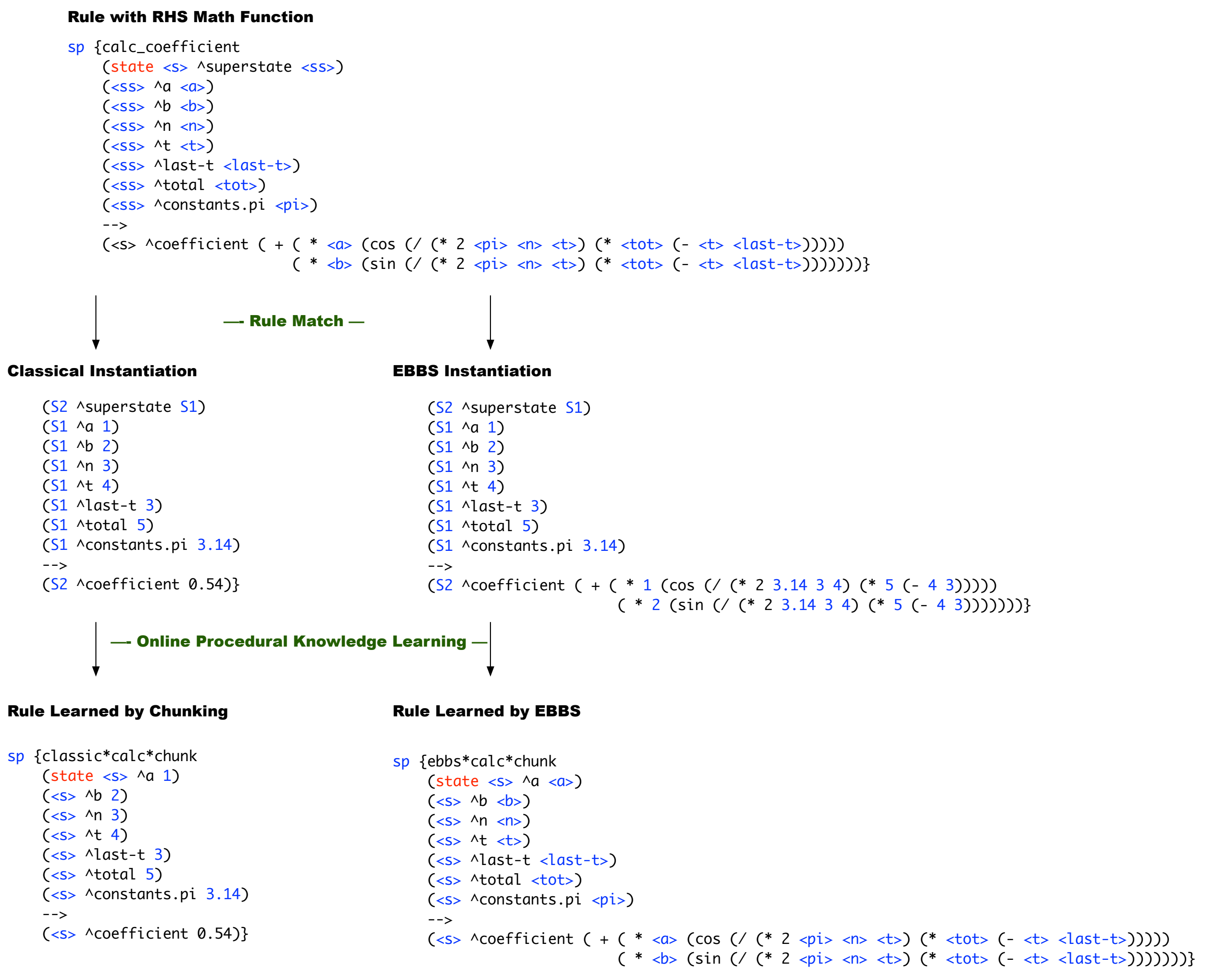
```
(S2 ^superstate S1)
(S1 ^object A1)
(A1 ^feature1 23)
(A1 ^feature2 33)
(A1 ^sub-object A2)
(A2 ^sub-feature1 45)
(A2 ^sub-feature2 91)

-->
(S2 ^copy-of-object C1)
(C1 ^feature1 23)
(C1 ^feature2 33)
(C1 ^sub-object C2)
(C2 ^sub-feature1 45)
(C2 ^sub-feature2 91)
```

Figure 6.4: EBBS adds conditions to an instantiation with dynamic actions.

6.1.1.4 Case 1, Example 3: Rules with Dynamic Actions (Math Functions)

Another example of a dynamic action is a math function, which are functions used in the right-hand side of a rule to create a WME that contains element(s) that are mathematical functions of input elements matched in the conditions. Since DIGU now provides the ability to generalize over numerical reasoning, EBBS should now be able to learn rules with math functions. To do this, EBBS needs more than what is provided by the instantiations used by classical chunking, which did not describe either the math functions or the dependencies to their parameters.[5]

EBBS adds the ability to learn rules with RHS math functions by building instantiations that include right-hand side functions. These instantiations describe the dependency between the arguments of the function and numerical elements in the conditions. Figure 6.5 compares a rule learned from an EBBS instantiation of a rule that uses a math function with one created by classical chunking.



```
sp {calc_coefficient
    (state <s> ^superstate <ss>)
    (<ss> ^a <a>)
    (<ss> ^b <b>)
    (<ss> ^n <n>)
    (<ss> ^t <t>)
    (<ss> ^last-t <last-t>)
    (<ss> ^total <tot>)
    (<ss> ^constants.pi <pi>)
    -->
    (<s> ^coefficient ( + ( * <a> (cos (/ (* 2 <pi> <n> <t>) (* <tot> (- <t> <last-t>)))))
                          ( * <b> (sin (/ (* 2 <pi> <n> <t>) (* <tot> (- <t> <last-t>)))))))}
```

```
(S2 ^superstate S1)
(S1 ^a 1)
(S1 ^b 2)
(S1 ^n 3)
(S1 ^t 4)
(S1 ^last-t 3)
(S1 ^total 5)
(S1 ^constants.pi 3.14)
-->
(S2 ^coefficient 0.54)}
```

```
(S2 ^superstate S1)
(S1 ^a 1)
(S1 ^b 2)
(S1 ^n 3)
(S1 ^t 4)
(S1 ^last-t 3)
(S1 ^total 5)
(S1 ^constants.pi 3.14)
-->
(S2 ^coefficient ( + ( * 1 (cos (/ (* 2 3.14 3 4) (* 5 (- 4 3)))))
                     ( * 2 (sin (/ (* 2 3.14 3 4) (* 5 (- 4 3)))))))}
```

```
sp {classic*calc*chunk
    (state <s> ^a 1)
    (<s> ^b 2)
    (<s> ^n 3)
    (<s> ^t 4)
    (<s> ^last-t 3)
    (<s> ^total 5)
    (<s> ^constants.pi 3.14)
    -->
    (<s> ^coefficient 0.54)}
```

```
sp {ebbs*calc*chunk
    (state <s> ^a <a>)
    (<s> ^b <b>)
    (<s> ^n <n>)
    (<s> ^t <t>)
    (<s> ^last-t <last-t>)
    (<s> ^total <tot>)
    (<s> ^constants.pi <pi>)
    -->
    (<s> ^coefficient ( + ( * <a> (cos (/ (* 2 <pi> <n> <t>) (* <tot> (- <t> <last-t>)))))
                          ( * <b> (sin (/ (* 2 <pi> <n> <t>) (* <tot> (- <t> <last-t>)))))))}
```

Figure 6.5: How EBBS provides the capability to learn rules that use RHS math functions.

While these new instantiations now properly describe the math function and its dependencies, which allows EBBS to learn rules with math functions that chunking could not, they also introduced a new correctness issue. Specifically, EBBS can learn an incorrect rule if a rule with a RHS

[5]This made perfect sense at the time because we knew the algorithm's heuristic-based generalization could not generalize any elements that matched a numerical symbol, so there was no benefit to including RHS math functions in an instantiation.

math function is used to create an intermediate value that another subgoal rule in the behavior trace later tests. We posit that the underlying challenge behind this issue is that EBBS is attempting to learn something that is not expressible in Soar's production syntax. This will be discussed in Section 6.2.1.

6.1.1.5 How EBBS Integrates Architectural Instantiations from 1A, 1B, 1C into online procedural learning

The previous three sections describe how EBBS accounts for missing dependencies by creating architecturally-created rule instantiations for rules that create dynamically-generates WMEs. To utilize these new dependencies into the online procedural learning process, EBBS then adds the architectural instantiations to the subgoal's instantiation graph. EBBS does this by providing a connection between any condition in the trace that tests one these WMEs described in the previous sections to its corresponding architectural instantiation.

As EBBS creates and adds these architectural instantiations to the instantiation graph, it must also determine how to assign identities to the elements of the instantiation. Unlike typical instantiations, architectural instantiations do not provide that information because they don't have *an underlying rule that contains variables which describes which elements are generalizable and which elements are not*. To handle this, EBBS manually assigns and propagates identities to all of the elements of the architectural instantiation depending on the nature of the particular case.

For subgoal meta-knowledge, assigning identities to elements of the instantiations is straightforward; those rules only test supergoal knowledge, so EBBS assigns new identities to each element.

For long-term knowledge retrievals, the conditions of the rule are assigned identities the same way that a standard rule does: a condition that test local knowledge has an identity propagated to it, while a condition that test supergoal knowledge has new identities assigned. The rule's actions, which create new knowledge structure that mirrors the one requested from LTM, will get new identities since they are not based on any existing structures in working memory.

For dynamically created actions, like those of deep-copy, assigning identities is a bit trickier because the structure copied could be subgoal knowledge, supergoal knowledge or even a combination of the two. To handle this, EBBS assumes two things based on the semantics of what a deep copy should do: first, that every element in a dynamic condition or action is generalizable, second, that the identities of each dynamic action maps to the same identities as its corresponding dynamic condition. So, EBBS first creates or propagates an identity for each element of the dynamically-created conditions; it does this based on whether the condition matched subgoal or supergoal knowledge. EBBS then assigns the same identities to each element of the corresponding dynamic actions. (The identities in the non-dynamic conditions and actions are assigned normally based on the initial rule match.)

Adding these architectural instantiations to the behavior trace and assigning/propagating identities to each element provides the missing dependencies that EBBS needs. When EBBS backtraces through these new instantiations, it will add conditions to the new rule that will enforce the dependencies that were necessary for the original reasoning.

6.1.2 Case 2: Operator Selection Knowledge

Operator selection knowledge (OSK) is the knowledge used to choose an operator from a set of potential operators. One of the reasons why OSK poses a problem for procedural learning is that, unlike typical rules that add and remove knowledge structures from working memory, operator selection rules create a different type of knowledge, preferences, which are added to preference memory. Specifically, there are six different types of preference in Soar. Some of them indicates a unary desirability for a single operator (acceptable, best, worst, required, prohibited, indifferent); others indicate a relative desirability between two operators (better, worst, equal/indifferent); and one preference is used to set numerical desirability preference that can be used to decide an operator probabilistically (indifferent). The following are examples of preferences:

```
(S2 operator O1 > O2)
(S2 operator O1 = O3)
(S2 operator O3 = 0.22)
```

The first preference indicates operator O1 should be preferred over O2. The second preference indicates that the agent should be indifferent between O1 and O3, all other things being equal. And the third preference assigns a numerical value to the desirability of O3 to 0.22.

Because operator selection rules do not create working memory structures and the behavior trace is a history of how rules changed working memory, operator selection knowledge is not described in the behavior trace. That means that, even though operator selection knowledge was used to effect reasoning in the subgoal, chunking does not include it in its analysis.

When chunking was first developed, this was not viewed as a problem. The philosophy at the time was that operator selection rules were used only to improve the efficiency of search and should not affect its validity. In other words, the path of operator selections and applications from an initial state in a subgoal to a result would always have all necessary tests in the operator proposal conditions and any goal test, so only those items would need to be summarized. The idea was that in a properly designed agent, a subgoal's operator evaluation preferences lead to a more efficient search of the space but do not influence the correctness of the result. As a result, the knowledge used by rules that produce such evaluation preferences should not be included in any chunks produced from that subgoal. And if that is the case – the operator selection rules are used purely to choose search paths that are more likely to lead to a solution – chunking does indeed learn correct rules.

In practice, it is common for agents to use general criteria to propose operators and then use operator selection rules to ensure that only operators that satisfy necessary goal requirements are considered. Since those rules test aspects of the situation that are required for the goal to be achieved, the trace must show how the agent's reasoning is dependent on the operator selection rules that were used to select the operators in the subgoal. Doing so will allow future summaries of the subgoal to include conditions for the superstate knowledge tested by the operator evaluation rules. We describe how EBBS integrates rules that create preferences into a behavior trace of rules that does not describe how preferences are used is described in Section 6.1.2.2.

Online procedural learning algorithms face a second and much more difficult challenge when incorporating operator selection knowledge into the rules it learns. This challenge is that the algorithm must analyze Soar's preference resolution semantics, the algorithm that determines how a set of preferences interact to choose a winner. How a set of these varied preferences determine which operator to select is a complex, multi-stage process. It is not entirely clear how an online procedural learning algorithm should analyze that process to determine which preferences play a *necessary* role in an operator selection. Whether a preference is relevant is highly dependent on the other types of preferences that exist at the time; for example, a better preference may not be relevant if another rule gives a best preference to an operator. In Section 6.1.2.2, we describe how EBBS analyzes operator selection semantics to determine which preferences to include in a learned rule.

6.1.2.1 Solution: Analyzing and Incorporating Operator Selection Knowledge

EBBS handles this issue by tracking what we call **Relevant Operator Selection Knowledge** (ROSK) for every operator selected in a subgoal and dynamically integrating it into the behavior trace. Relevant operator selection knowledge is the set of necessary operator evaluation preferences that led to the selection of an operator in a subgoal.

EBBS builds a ROSK set for an operator every single time the agent chooses an operator from a set of proposed operators. It does this by analyzing the operator preferences involved in the selection of the operator to determine which preferences played a *necessary role*.

To understand how EBBS determines which of those preferences to include in the ROSK, we must describe the preference resolution algorithm that Soar uses to choose an operator. This algorithm consists of a sequence of seven steps, each of which handles a specific type of preference. At each step, a preference semantics is applied that incrementally filters the candidate operators. When the set is reduced to a single candidate, it is selected as the operator. At each of these steps, EBBS may add some of the preferences that the resolution algorithm is processing to the ROSK set based on what role those preferences played in that step. For example, all necessity preferences (prohibit and require) are added to the ROSK set in the first step since they inherently encode the correctness of whether an operator is applicable in a problem space. In contrast, some desirability preferences (rejects, betters, worses, bests, worsts and indifferents) are only included in the ROSK if they play a necessary role in the selection of the operator. The philosophy we used in this work is that a preference is necessary if removing the preference could lead to a different operator being selected.

6.1.2.2 How EBBS Determines Which OSK Preferences are Relevant

Soar's preference resolution algorithm analyzes the set of all operator preferences currently under consideration in a goal. There are six steps to the algorithm, each of which analyzes one type of preference.[6] The algorithm starts with a list of candidate operators and then iterates through all preferences in memory. As it iterates through the preference types, it removes candidate operators as necessary. The algorithm returns either a winning operator or, if it can't determine a winner, an impasse type with a list of tied or conflicting operators.

The following outline describes the logic underlying the algorithm. At each step, we (1) define the preference type will be iterated over and analyzed in that step, (2) describe how Soar uses that type of preference to select an operator and (3) describe which of those preferences EBBS chooses to add to the ROSK set and why.

Note that the algorithm only proceeds to the next step if there is more than one candidate operator left. If one or no candidate is left, the algorithm terminates.

1. **Require Preferences**

   A require preference indicates that an operator must be selected if the goal is to be achieved.

   Soar If there is exactly one candidate operator with a require preference and that candidate does not have a prohibit preference, preference semantics terminates and that candidate is returned as the winner.

   ROSK If an operator is selected based on a require preference, that preference is added to the ROSK. The logic behind this step is straightforward: the context of the require preference directly resulted in the selection of the operator and should be included in any rules that summarize reasoning dependent on that operator.

2. **Prohibit/Reject Preferences**

[6]In some cases, a step analyzes two closely related preferences, for example better and worse preferences.

A reject preference indicates that an operator is not a candidate for selection, while a prohibit preference states that the value cannot be selected if the goal is to be achieved.[7]

Soar This step removes from consideration any candidate operators that have a prohibit or reject preference. If only one candidate operator remains, preference semantics terminates and that candidate is returned as the winner.

ROSK If there exists at least one prohibit or reject preference, all prohibit and reject preferences for the eliminated candidates are added to the ROSK. The logic behind this stage is that the conditions that led to the exclusion of the prohibited and rejected candidates is what allowed the final operator to be selected from among that particular set of surviving candidates.

3. **Better/Worse Preferences**

Better and worse preferences indicate that one operator should be preferred over another.

Soar This step removes any candidates that are worse than another candidate.

ROSK For every candidate that is not worse than some other candidate, add all better/worse preferences involving the candidate. The logic behind this step is that these better/worst preferences allowed the candidate operator to survive this step and eventually be chosen.

4. **Best Preferences**

A best preference indicates an operator should be chosen over all other operators under consideration.

Soar If at least one remaining candidate has a best preference, this step removes any candidates that do not have a best preference. If no best preferences exist, this step is skipped.

ROSK Add any best preferences for surviving candidates to the ROSK.

5. **Worst Preferences**

A worst preference indicates an operator should only be chosen if no other operators are currently under consideration.

Soar This step removes any candidates that have a worst preference. If all remaining candidates have worst preferences or there are no worst preferences, this step has no effect.

[7]While there is a slight difference between the two preferences, we can consider them equivalent for this discussion.

ROSK If any remaining candidate has a worst preference which leads to that candidate being removed from consideration, that worst preference is added to the ROSK. The underlying logic is that part of the reason that the winning operator was selected was that other candidates were removed from consideration by these worst preferences. Hence, the context that led to those worst preferences should be included in any rules that summarize reasoning dependent on that operator.

6. **Indifferent/Numerical Preferences**

A unary indifferent preference indicates that an operator is as good a choice as any other expected alternative. A binary indifferent preference indicates that an operator is as good a choice as another operator. A numerical indifferent preference assigns a preference value to an operator that can be used in tandem with whatever exploration policy is set to select amongst a set of operators that all have numerical preferences. Numeric preferences can be used to select operators probabilistically or as the basis for Soar's reinforcement learning mechanism.

Soar This step is a bit more complex than the others. It traverses the remaining candidates and marks each candidate that has either a unary indifferent preference or a numeric indifferent preference. This step then checks every candidate is not marked to see if it has a binary indifferent preference with every other candidate. If one of the candidates fails this test, then the procedure signals a tie impasse and returns the complete set of candidates that were passed into this step. Otherwise, the candidates are mutually indifferent, in which case an operator is chosen according to whatever indifferent selection or exploration policy was selected for the agent.

ROSK Indifferent preferences are added to the ROSK set depending on whether any numeric indifferent preferences exist.

(a) If there exists at least one numeric indifferent preference, then every numeric preference for the winning candidate is added to the ROSK. There can be multiple such preferences. Moreover, all binary indifferent preferences between that winning candidate and candidates without a numeric preference are added.

(b) If all indifferent preferences are non-numeric, then any unary indifferent preferences for the winning candidate are added to the ROSK. Moreover, all binary indifferent preferences between that winning candidate and other candidates are added.

The logic behind adding binary indifferent preferences between the selected operator and the other final candidates is that those binary indifferent preferences prevented a tie impasse and allowed the final candidate to be chosen by the exploration policy from among those mutually indifferent preferences.

If, at any point in the process, a winner is chosen, the decision process will exit with a finalized ROSK set for the selected operator. If a winner is not chosen and an impasse occurs, the ROSK set is emptied.

Integrating the ROSK Set into the Behavior Trace

To utilize the knowledge encoded in the ROSK set when learning a rule, EBBS must somehow link the ROSK set to the behavior trace. As previously mentioned, one reason that utilizing preference knowledge when learning rules is not straightforward is the fact that the trace describes working memory changes and operator selection rules do not create working memory structures. EBBS handles this dilemma as follows. Whenever an instantiation is created for a rule that tests an operator, EBBS also includes that particular operator's ROSK set of preferences. So, as EBBS backtraces through each rule instance in the trace, it will also branch out and backtrace through the instantiations that created each of the ROSK preferences for the tested operator.[8] By backtracing through these additional set of rules at each step of the backtrace, EBBS incorporates the conditions that led to the creation of the ROSK preferences.

Limitations

How we select which preferences to include in the ROSK is still an open question that needs further study. While the logic underlying how we determine the ROSK set seems reasonable and has produced agents that can learn the rules we hoped they would learn, it is certainly possible to come up with different rationale that would lead to a different set of preferences in the ROSK.

This approach is also limited by the fact that operator selection knowledge is non-monotonic. A single rule can fire and change the relevancy of the other rules in the OSK. Even though that priority-shuffling rule did not match in the situation being learned form, it could have matched in other contexts. An agent whose reasoning included OSK rules that interact in such ways may learn incorrect rules.

As a result of these factors, we consider this aspect of the EBBS algorithm a work in progress. We characterize this implementation as a first pass at exploring of the viability of the ROSK approach in general and our preference semantics in particular.

[8]If the rule does not test an operator, the ROSK set will be empty and no additional backtracing occurs.

6.1.3 Case 3: Promoted Knowledge

In many agents, a single subgoal often returns multiple results to a supergoal before the impasse is resolved. It is not uncommon for subgoal reasoning to build a knowledge structure in the subgoal, return it as a result and then continue to use that structure in the subgoal for further reasoning. This can pose a problem for Soar's procedural learning algorithm. Specifically, chunking can learn an incorrect rule if it attempts to summarize reasoning that augments or tests knowledge structures that were previously returned as a result.

This problem can be more easily explained if we introduce the notion of knowledge promotion. We will define knowledge promotion as follows. A knowledge structure in a subgoal is promoted from local knowledge to supergoal knowledge when it becomes accessible to a supergoal. So, every rule action that returns a result is doing one of two things: it is either (1) creating a new knowledge structure directly in the supergoal or (2) connecting a local knowledge structure to an existing supergoal structure, hence promoting it.

Knowledge promotion makes procedural learning difficult because working memory indicates that the promoted knowledge is supergoal knowledge while the behavior trace only describes that knowledge relative to the subgoal.

For example, consider a rule that tests a subgoal structure and returns it as a result. This rule "promotes" the structure from subgoal knowledge to supergoal knowledge. After its promotion, there will be two paths to the result: one in relation to the supergoal and one in relation to the subgoal. At this point, any instantiation in the trace that tested the promoted structure *in relation to the subgoal* becomes a problem. The online procedural learning algorithm will know that the knowledge tested is supergoal knowledge but it will not know how to build conditions that describe the knowledge relative to the supergoal, since the rule that matched only described the structure relative to the subgoal.

Now consider what happens when another rule fires, tests the promoted structure relative to the subgoal and returns a new result. Chunking will attempt to learn a rule by backtracing through the trace that led to the result. During the backtrace, it will encounter the condition that tests the promoted structure. It will look at the promoted WME and determine that it is accessible to the superstate, so it adds it as a condition to the chunk it is building. What it won't add are any additional conditions that connect the promoted structure to the supergoal goal; this is because the rules in the trace still only reference the structure in respect to the subgoal. Consequently, chunking learns a malformed rule with a condition that is not connected to a goal node; this rule is inherently incorrect because it can never match in any of the situations that original subgoal reasoning applies in. In summary, chunking fails because it cannot determine how to specify conditions that match the previously promoted structure in relation to the supergoal. This will produce a rule with unconnected conditions, which are not only incorrect but impossible to add to Soar's production memory.

With classical chunking, this problem could only be avoided by modifying the agent's original rules so that they never test or augment the substructure of a previous result. In fact, whenever classical chunking learned a malformed rule, it would suggest that the agent designer check whether the agent may have inadvertently tested a previous result.[9] The rules would need to be modified so that the conditions of any rules that test the promoted knowledge are changed to test them relative to the supergoal rather than the subgoal. Only with such changes would classical chunking be able to learn a correct rule.

In the next section, we will discuss how EBBS is able to learn correct rules in such situation without any modification of the agent's reasoning using a new rule repair algorithm.

6.1.3.1 Rule Repair

If an unconnected action or condition is found when validating a learned rule, EBBS will know it must have tested promoted knowledge and will attempt to repair the rule by adding new conditions that provide a link from a goal node to the unconnected condition or action. To determine a path to the promoted knowledge, EBBS examines the agent's working memory and looks for the promoted knowledge, starting at all goal nodes referenced in the malformed rule. It attempts to find the shortest path from one of those goal nodes to the top element of the promoted knowledge, which will be the soar identifier that matched the unconnected connection. Because there may be multiple paths from a state to the unconnected identifier, EBBS does a breadth-first search to find the one with the shortest distance. It will then create a set of conditions based on the set of WMEs that provided the path from the goal node to the WME that matched the disconnected condition. So, these new *grounding conditions* provide a path from a goal that is tested somewhere else in the rule, to the unconnected condition or action. EBBS then variablizes those conditions in much the same way that it variablizes the conditions it adds for dynamic actions as described in Section 6.1.1.3.

Note that while a repaired rule is correct, there's a chance it may also be undergeneral. This is because the conditions are based purely on what happened to be in working memory at that point and nothing in the explanation dictated that particular path found during the search. The chunk will only match future situations where the previous result can be found on that same path. Fortunately, in most cases, we suspect that they won't be undergeneral in practice. The rules are only overspecialized when the knowledge in the supergoal is moved after the results is returned

[9]In some sense, that warning was classical chunking's detect and prevent mechanism for this correctness issue.

but before a second chunk is learned, which seems like a peculiar edge case that is not likely to happen. Even if such the rule learned was indeed overspecialized, new chunks can be learned to handle other cases. If a similar situation is encountered in the future where the knowledge structures end up in a different location relative to the supergoal, the chunk won't fire, because the WM-based grounding conditions won't match; this should cause the agent to subgoal and learn a new, similar chunk with a different set of grounding conditions. In most cases though, the path to the previous result found by the search should exist, and a repaired rule should match in all of the same situations that the original reasoning does.

A better approach might be to infer a path to the promoted knowledge by examining the rule that previously promoted the knowledge. Because the final rule in that trace is the one that promoted the knowledge, it must describe how it connected the knowledge to the supergoal. If EBBS could infer the path from that rule, it could then repair the malformed rule by adding conditions that specify that inferred path to the promoted knowledge. Though we did plan to experiment with this idea, we assigned it a low priority since we already had the rule repair mechanism available from a different correctness issue. Moreover, an algorithm that inferred the path from the first chunk learned would generate the exact same conditions as those generated by rule repair, except in one edge case: the repaired rule will be undergeneral only if another rule fires, after the knowledge is promoted but before the second result is returned, and moves the knowledge around in the supergoal. A rare case of undergenerality did not seem to warrant the effort it would take to add a significant new mechanism.

6.1.4 Case 4: Implicit Knowledge

It can be difficult if not impossible to learn correct procedural knowledge when summarizing reasoning in a subgoal that is implicitly dependent on knowledge that is not encoded or a situation that cannot be described easily. For this correctness problem, we acquire the missing knowledge by making the implicit knowledge explicit. We've seen this strategy applied to a few problems already. For example, user-defined singletons allow an agent engineer to describe domain-specific uniqueness properties that they implicitly assumed while writing the agent's rules. Moreover, several of the solutions to missing dependencies in Section 6.1.1 could be described as examples of making implicit knowledge explicit. In those cases, EBBS creates architectural instantiations that explicitly define dependencies for subgoal meta-knowledge that agent engineers would previously test because of the implicit knowledge they had about the semantics of that meta-knowledge.

6.1.4.1 Case 4, Example 1: Implicit Knowledge of Soar Operation

Consider an agent that needs to determine that all operators have applied in a goal. A convenient way to do that is to allow all operators to be applied, wait for the impasse that occurs when no further operators are available, which is called a state-no-change impasse. The agent can then use a simple rule in the resulting subgoal that tests if the impasse type is state-no-change, and, if it is, create a result in the supergoal indicating that *operator exhaustion* has been reached. Chunking obviously does not have much to summarize in this case. Because the reasoning in the subgoal is only dependent on the existence of the subgoal, chunking will learn the most overgeneral rule possible, a rule with no conditions, which it will discard. Even though the rule was thrown out, the superstate result, in this case a flag that indicates that operator exhaustion was reached, is still created by the subgoal rule that matched. Any rule that is later learned by summarizing reasoning that tests that exhaustion flag will be over-general since the rule will not contain any conditions that ensures that the chunk will fire only after all others operators were first exhaustively applied.

A similar problem can occur when an agent uses a worst preference to make sure that an operator is selected only after all other operator options have been exhausted. If this final operator then produces a result that is implicitly dependent on it occurring after all other operators, it will be overgeneral and incorrect because that requirement will not be captured in any of the conditions of the learned rule.

To make the implicit knowledge explicit, EBBS could create an operator exhaustion feature in the supergoal right before it creates the state-no-change impasse. Agents could then test that feature to detect that all operators have been applied. Moreover, an agent that is designed to subgoal before testing that feature could learn a rule that, in future similar situations, will fire after all operators in the goal have been applied but before the state-no-change impasse is created.[10]

6.1.4.2 Case 4, Example 2: Testing the Absence of Knowledge Structures

Unfortunately, making implicit knowledge explicit may not always be possible. For example, consider the case of learning a rule from a behavior trace that includes *conditions that test for the absence of a working memory elements in the subgoal*. A chunk that summarized that subgoal should not fire in situations that could give rise to the WME that violated the negative condition. Since there is no practical way for chunking to generate a set of conditions that describe all possible situations in which such WMEs could be created, chunking can't represent that aspect of the problem-solving. Chunks learned from such traces have the potential to fire in situations where the subgoal processing would not have occurred and consequently are incorrect.

In many cases, though, summarizing traces that include negative tests will generate rules that are perfectly safe to use. Specifically, such rules are correct as long as no other rules exist that could have created a WME that matched the negative condition in a hypothetical lower level goal.[11] In

[10]Adding these new features is not yet implemented but is on the road map of future work.

[11]We say "could have" and hypothetical, because the chunk will fire before an impasse occurs and a subgoal is spawned.

other words, summarizing traces with local negations will not lead to learning incorrect rules if the only rules that create the prohibited WMEs are also subgoal rules that only test subgoal knowledge.

And this is why we describe this as an issue involving implicit knowledge. In some sense, all rules learned from behavior traces that include tests for the absence of knowledge in the subgoal are implicitly dependent on any supergoal rules that could potentially create knowledge that match the negative condition(s), not existing. If no such supergoal rules exist, rules learned from the trace will indeed be correct. If such rules do exist, then the chunks may incorrectly generate inferences in situations where the original subgoal reasoning would not have. (Specifically, those chunks should not fire whenever the conditions of rules that can create the prohibited WME match.) In other words, *an agent designer should only include such tests in the agents they write when they have knowledge that such rules do not exist*.

While we do not have an algorithmic solution to this problem, there is a workaround, as well as a mechanism to help detect when that workaround is necessary. If the agent engineer knows that a local feature that they have a negative test for can be created by a supergoal rule, they can reformulate their rules so that they *create the prohibited WME in the supergoal*. This lets them change the negative condition so that it test for the *absence of knowledge in the supergoal rather than the subgoal*, which will allow both EBBS and classical chunking to learn a correct rule that includes the negative condition. To help an agent designer figure out when this technique is necessary, both chunking and EBBS provide a limited form of detect and prevent that detects when it has summarized a rule that include a negative test of a local knowledge structure. It is limited in the sense that it can detect when a local negation is used in a behavior trace but it cannot determine whether that use is problematic or not. This mechanism also includes an option to prevent it from learning rules based on traces that test local negations, but it is not enabled by default.

6.1.5 Case 5: Learning from Non-Learning Subgoals

In Section 5.9, we explained why chunking may not learn an optimally general rule from a trace that tests knowledge previously generated by a non-learning subgoal. What we did not mention was that those learned rules could also be incorrect. If the non-learning subgoal involves any of the problematic types of reasoning that we describe in this chapter, the justification learned may only provide a partial description of the context that led to the result. And, if the knowledge produced by the justification is involved in a future learning episode in the supergoal, the new rule learned would inherit the same incomplete description and lack of correctness.

To handle this problem, EBBS enables all of the correctness mechanisms described in this chapter in non-learning subgoals. For example, EBBS adds the new architectural instantiations to the behavior trace of non-learning subgoals. It also analyzes operator selection knowledge and backtraces through them. EBBS will even perform rule repair on justifications if it finds disconnected conditions, just like it does in learning subgoals. By utilizing all of its correctness mechanisms in non-learning subgoals, EBBS ensures that it fixes any correctness issues before they are propagated to a supergoal where learning is indeed enabled.

6.2 Strategy 2: Selective Specialization

Selective specialization is another technique EBBS uses to fix incorrect rules. If EBBS determines that an aspect of an agent's reasoning is problematic to generalize over, it sets the identity of the involved elements to the null identity, which prevents generalization of those elements. While the learned rule loses some generality, it gains correctness.

6.2.1 Case 1: Behavior Traces with Constraints on Intermediate Results

In many real-world systems, agents need to perform mathematical calculations using functions in the actions of their rules to calculate them. Since the DIGU algorithm allows us to generalize over all data types, EBBS is now able to learn more expressive rules whose actions perform arbitrary right-hand side functions with variablized arguments. While this is a powerful capability, it does introduce a correctness issue. If another rule in the chain of reasoning tests whether the output of the function meets some mathematical constraint, chunking is not able to summarize the reasoning.

The underlying problem is that the chunking is attempting to learn a rule that summarizes reasoning that is not expressible in a single rule in the cognitive architecture's production syntax. For example, consider the case of one rule that used a RHS function to add two numbers. Now consider another rule that matched the output of the RHS function, but only if it was less than 5. If the second rule matched, it would return the total as a result. How could we encode the reasoning of those two rules into one rule? Since Soar's production syntax does not allow using RHS function as constraints in conditions, there is no way to ensure that the two numbers add up to something less than 5 in a single rule. In other words, there is no way to write a *single* rule that encodes "if (x + y) < 5, make this inference," because there is no way to perform a calculation in a condition test. So, even though it's easy to encode this reasoning using two rules, chunking cannot summarize it into one.

Solution: Selective Specialization of Constrained Intermediate Results

Because the chunk's conditions can't represent constraints on the output of intermediate RHS functions, EBBS sets the identities of the variables that appear as arguments to the intermediate RHS function to the null identity, as well as the identities in any conditions that test the output of the RHS function. That ensures the output value of the RHS function is constant and guarantees that any constraints in conditions that test the output of that RHS function will be met. While this will make the learned rule more specific, it will also ensure that the rule is correct.

It is worth noting that strategy 4, modifying the architecture, could also be applied to this correctness issue. In this case, the architecture modification would be a new type of constraint that allowed mathematical calculations, i.e. it would calculate the intermediate result during that matching process to determine whether the rule should match. Another way that strategy 4 could also be used is by changing the online procedural learning algorithm so that it learns two rules.[12] In some sense, this would be like partitioning the behavior trace and learning a rule for each partition.

6.2.2 Comments on Selective Specialization

Selective specialization is a technique that allows an online procedural learning algorithm to trade some generality for correctness. Selective specialization is particularly easy to apply to different correctness issues, which makes it an excellent placeholder mechanism that can be used by a cognitive architecture until the correctness issue is better understood and a more effective technique developed. In fact, that occurred in this project; previous versions of our online procedural learning algorithm used selective specialization to handle several issues that we now have better solutions for.

In some sense, classical chunking is a great example of using selective specialization as a placeholder. Its straightforward generalization mechanism, which replaces all Soar identifiers with a corresponding variable, is equivalent to selectively specializing every element of the learned rule that did not test a Soar identifier. So, in some sense, chunking used a wide-spread form of selective specialization as a placeholder mechanism to avoid some of the correctness issues that we have described in this thesis, and, once it became possible to study the issue further, it replaced that application of selective specialization with a more powerful architectural modification, the DIGU algorithm.

We could also rephrase an argument made in 3.6.1 in these terms, by saying that classical chunking cannot be used by many agents because the generality-correctness trade-off it incurs through selective specialization is too high and leads to overly specialized rules that do not provide benefits to many agents.

[12]The architecture could use some sort of semaphore in the supergoal to make sure that the rules fire in sequence.

6.3 Strategy 4: Modify Architecture to Ensure Correct Learning

As described in Section 4.4, the fourth strategy EBBS utilizes is to modify mechanisms in the architecture to ensure its processing does not violate one of the four necessary conditions for an online procedural learning algorithm to learn correct rules, as described in Section 2.2. If the change succeeds in making the processing of the cognitive architecture parsimonious with the online procedural learning algorithm, the correctness issue can be avoided entirely.

As discussed earlier in this thesis, the first example of this strategy being applied to improve Soar's procedural learning can be found in the work behind Robert Wray's 1998 thesis on ensuring consistency in hierarchical architectures (Wray et al., 1996). In his research work, Wray described how changes to the environment that occurred during problem-solving could lead to learning incorrect, logically inconsistent rules. To eliminate this source of incorrect rules, Wray modified basic Soar operation, namely how it manages subgoals, by adding a new architectural mechanism called the Goal Dependency Set (GDS). (Wray and Laird, 2003) The GDS monitors knowledge dependence in every subgoal to ensure that the reasoning in each is always consistent with higher-level knowledge structures. By eliminating the possibility that a behavior traces will contain inconsistent assertions, Wray's GDS system eliminated the possibility that chunking would learn an incorrect, non-contemporaneous rule that could never match.

In the next section, we describe how we modify a major component of the Soar architecture to allow EBBS to learn correct procedural knowledge when knowledge structure retrieved from Soar's long-term memory systems interact in unusual ways across subgoals.

6.3.1 Long-term Memory Elements and Long-Term Identifiers

One of the most important differences between short-term and long-term memory systems is that LTM is composed of long-term working memory elements (LTMEs) which use their own set of transient identifiers called *long-term identifiers* (LTIs), to group features. Structurally, an LTME is identical to a WME. An LTME also consists of three symbols: the *long-term identifier*, a unique special symbol analogous to a typical Soar identifier, an *attribute*, which specifies a feature label, and a *value*, which specifies either a constant value or another LTI.

Semantic memory ***only*** *contains LTIs in its knowledge structures. In contrast, working memory can contain both STIs and LTIs.* When Soar creates knowledge structures in working memory in response to a query or retrieval, *it will use the same LTIs in the new WMEs it creates that were used* *in the LTMEs that they were based on.*[13] LTIs are also created in short-term memory when an agent requests that a WME is stored to LTM; Soar immediately converts all the STIs in the WME to be stored to LTIs. Either way, an LTIs in short-term memory is, in some sense, a hybrid identifier that indicates a structure is simultaneously in both memory systems. As will become clear in Section 6.3.2, it is this dual, hybrid nature that underlies the key problem that our online procedural learning algorithms faces when trying to learn from knowledge retrieved from long-term memory stores.

6.3.2 Case 1: LTM Knowledge Structures Interacting Across Subgoals

When we began testing early versions of our online procedural learning algorithm with ROSIE, a complex agent that learns by instruction and uses knowledge retrieved from long-term semantic memory, we encountered a variety of show-stopping problems. Not only would ROSIE learn incorrect rules, but it would often not achieve the same goals it could achieve when learning was disabled. These issues were particularly challenging – we spent nearly two years analyzing them and exploring solutions – because they could not easily be recreated and only occurred with our most complex agents. To simplify this discussion, we will avoid describing the convoluted particulars of the variety of ways that this problem manifested itself and simply explain the underlying issue.

The reason that Soar's semantic memory model introduces a fundamental problem for Soar's procedural learning algorithm is that knowledge structures can be simultaneously retrieved at multiple levels of the goal hierarchy. If the same knowledge structure is retrieved by two different queries in two different subgoals of the goal stack, unintended interactions can occur between the reasoning in the two subgoals. In these cases, the two queries in the two subgoals will both return the same long-term identifier. Because Soar identifiers are unique, the same structure will be connected to both subgoals, which makes it difficult for chunking to summarize either subgoal. All rules that test knowledge structures recalled from semantic memory in the subgoal must now be treated as tests of supergoal knowledge; similarly, rules that augment knowledge structures recalled from semantic memory in the subgoal must now be treated as results to the supergoal.

In some sense, multiple semantic memory recalls interacting with each other is a challenge for the same reason that using *promoted knowledge* is a challenge. Consider what happens when a knowledge structure is retrieved from semantic memory in a subgoal and is then retrieved again by a different query in a higher-level goal. The moment that the knowledge is retrieved in the higher-level goal, it has effectively been promoted to supergoal knowledge. The semantic memory knowledge structures then become problematic for the same reason that a promoted chunk result is

[13]This design allows the agent to perform operations on semantic memory, for example updating the contents of semantic memory by adding new structures and then re-storing it.

problematic. If chunking tries to summarize a behavior trace that utilizes that promoted knowledge, it will know that it needs to create conditions that test the semantic memory structures in the learned rule, but it will not how to build conditions to describe the knowledge relative to the supergoal because the behavior trace only describes how to access that knowledge structures in terms of the subgoal. Similarly, if a behavior trace augments one of these multiply-retrieved semantic memory knowledge structures, it will detect that it is making knowledge available to the supergoal, but it will not be able to build conditions from the trace that describe where that knowledge was added.

To make things even more difficult, the fact that these problems stemmed from interaction between subgoals meant that these issues were highly dependent on the timing of the queries and the unexpected coincidences between the reasoning at different levels of the goal hierarchy. This is why we saw so many different manifestations of the problem, many of which were hard to recreate.

Episodic memory can also be accessed at any subgoal level, but it did not introduce any new problems for our online procedural learning algorithm. We posit that the reason that similar problems did not occur is that episodic memory made a different design choice. Specifically, it did not utilize hybrid symbols to indicate that a structure in working memory is also in episodic memory. Instead, it generates new identifiers every time an historical episode is requested from episodic memory. We did face a few problems learning from knowledge recalled from episodic memories, but they all arose because a reconstructed episodic memory recreates any LTIs that existed in the agent's working memory at the time the episode was stored. In other words, they were just manifestations of the same challenge we were facing learning directly from semantic memory knowledge. The strategies we describe in the following section to handle these semantic memory related correctness issues also eliminated the problem from occurring when using episodic memory.

6.3.2.1 Early Solutions

The problems caused by unintended interactions between semantic memory knowledge structures across subgoals is another example of a correctness issue that occurs when a behavior trace does not explain all dependencies. In this case, the missing knowledge is a description of how the semantic memory structure is linked to a supergoal. To generate this missing description, our early version of EBBS used rule repair. As discussed in 6.1.3.1, rule repair can be used to fix any malformed rule with conditions that are not connected to a goal, which is exactly what happens when chunking attempts to learn a rule in one of these problematic semantic memory cases.[14] While this was a serviceable solution – the rules EBBS would learn would correctly summarize the reasoning in the subgoal – the agents still had problems in many situations because the rules were codifying unintended interactions between subgoals. In other words, agents were learning

[14]Despite the fact that rule repair currently only corrects issues that stem from traces that test promoted knowledge, it was originally developed as a response to semantic memory issues.

rules that summarized and re-created undesirable, coincidental interactions between reasoning in different subgoals.

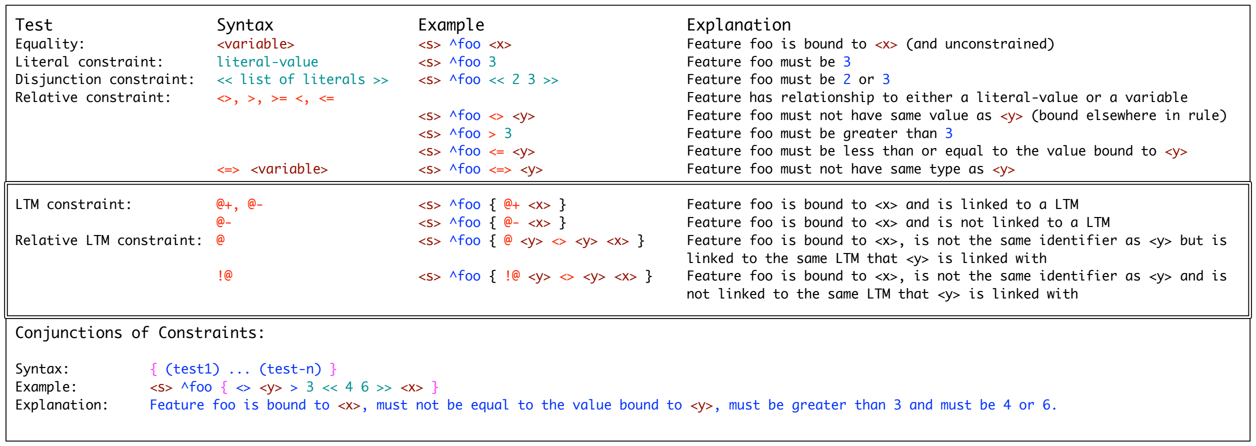
Eventually, we realized that the real problem is that Soar's model of semantic memory was in conflict with a core assumption of Soar's online procedural learning algorithm, which is that knowledge structures are created by rules and that dependence between reasoning and knowledge structures can be determined by examining a trace of rules. Chunking cannot summarize how subgoal reasoning relates to a supergoal knowledge if subgoal knowledge can become supergoal knowledge at any time without explanation. So, our solution was to fundamentally change how Soar represents semantic memories in working memory so that each semantic memory retrieval is independent and does not interact with other retrievals. This change, which we describe in 6.3.2.2, was successful and allowed us to completely eliminate both the unexpected interactions between memory retrievals and the incorrect rules they gave rise to.

One may wonder why we didn't eliminate this issue by simply removing the ability to retrieve knowledge structures from LTM at multiple levels of the hierarchy. In other words, why didn't we just change the architecture so that there is a single SMEM link on the top level that the agent must use for all semantic memory recalls? The answer is that it doesn't completely solve the problem. It would still be possible for an agent to "promote" subgoal knowledge by issuing a command to store a short-term memory structure into semantic memory. After the structure is stored, Soar replaces all of the short-term Soar identifiers in the stored structure to long-term identifiers, which means that it's now possible for reasoning in a supergoal to retrieve the local structure that was just stored, which would lead to the same problematic situation we faced before. The problems would be a lot less common, but they would still be possible. So, while having multiple SMEM links does exacerbate this correctness issue to the point that it became extremely widespread, the real underlying issue was the hybrid nature of long-term identifiers and our model of semantic memory.

6.3.2.2 Instance-Based Semantic Memory

Our ultimate solution was to replace Soar semantic memory system with a new model in which every retrieval is an independent instance of a long-term memory structure. In this new model, long-term identifiers do not appear in short-term memory at all; all semantic memory requests made by an agent return knowledge structures that use standard Soar identifiers. For example, if an agent performs two identical recalls in two different subgoals, Soar would return a set of WMEs in each subgoal that are structurally identical but use different short-term identifiers. In our first model of semantic memory, there was always a 1-to-1 relationship between a structure in LTM and one in STM; in our new model, there can be 1-to-N relationships. We say that each of these N knowledge structures is an independent instance of the long-term memory upon which they are based, and we call the new system **instance-based semantic memory**.

A short-term instance of a LTM looks like any other set of WMEs, but underneath the hood, it contains additional information. Specifically, Soar maintains meta-information that links each short-term identifier in the recalled structure with the long-term identifier it was based on. While these links are not directly inspectable by the agent, Soar keeps track of all of them so that instance-based semantic memory can provide the same functionality that our previous hybrid model of semantic memory provided. To do so, though, additional capabilities were needed that allow an agent to test the *long-term identity* of a short-term knowledge structure in the condition of a rule. For example, instance-based semantic memory introduces a new test that is only fulfilled when two short-term working memory structures are instances of the same long-term structure. Specifically, this new test compares whether a short-term identifier in a condition is linked to the same long-term identifier as short-term identifier in another condition of that rule. Figure 6.6 lists all of the new tests and actions that instance-based semantic memory adds.



Test	Syntax	Example	Explanation
Equality:	<variable>	<s> ^foo <x>	Feature foo is bound to <x> (and unconstrained)
Literal constraint:	literal-value	<s> ^foo 3	Feature foo must be 3
Disjunction constraint:	<< list of literals >>	<s> ^foo << 2 3 >>	Feature foo must be 2 or 3
Relative constraint:	<>, >, >= <, <=		Feature has relationship to either a literal-value or a variable
		<s> ^foo <> <y>	Feature foo must not have same value as <y> (bound elsewhere in rule)
		<s> ^foo > 3	Feature foo must be greater than 3
		<s> ^foo <= <y>	Feature foo must be less than or equal to the value bound to <y>
	<=> <variable>	<s> ^foo <=> <y>	Feature foo must not have same type as <y>
LTM constraint:	@+, @-	<s> ^foo { @+ <x> }	Feature foo is bound to <x> and is linked to a LTM
	@-	<s> ^foo { @- <x> }	Feature foo is bound to <x> and is not linked to a LTM
Relative LTM constraint:	@	<s> ^foo { @ <y> <> <y> <x> }	Feature foo is bound to <x>, is not the same identifier as <y> but is linked to the same LTM that <y> is linked with
	!@	<s> ^foo { !@ <y> <> <y> <x> }	Feature foo is bound to <x>, is not the same identifier as <y> and is not linked to the same LTM that <y> is linked with

Conjunctions of Constraints:

Syntax:	{ (test1) ... (test-n) }
Example:	<s> ^foo { <> <y> > 3 << 4 6 >> <x> }
Explanation:	Feature foo is bound to <x>, must not be equal to the value bound to <y>, must be greater than 3 and must be 4 or 6.

Figure 6.6: The new tests in the middle pane can be used in rule conditions to test how a short-term WME is linked to a LTM.

With these new capabilities, instance-based semantic memory not only allows all of the same types of reasoning that was possible with our old model of semantic memory, but it also makes several other use cases possible or at least much easier. For example, it is now much easier to use the same semantic knowledge when reasoning about multiple hypothetical situations, because the engineer no longer needs to worry about the knowledge structures used in one hypothetical affects the structures in a different hypothetical.

Instance-based semantic memory also eliminated correctness problems chunking had learning rules when agents also used episodic memory. Those problems were essentially the same as the semantic memory issues and arose because episodic memory would have a record of all LTIs that were in the agent's memory at the time the episode was recorded. The only unique aspect of this version of the correctness problem is that the LTIs involved in the unexpected interaction with reasoning in another subgoal were acquired from an episodic memory recall rather than a semantic memory recall.

6.3.3 Case 2: Subgoal Results Based on Long-Term Memory Structures

When discussing Soar's taxonomy of correctness issues in Section 3.5, we described how many correctness issues arose when Soar gained additional capabilities. The following correctness issue is an example of that. This issue did not exist until we replaced Soar's model of semantic memory with an instance-based one.

EBBS may learn an incorrect rule when it summarizes subgoal reasoning that generates a result which includes WMEs that are instances of LTMs, i.e. WMEs that were created in response to a semantic memory request. While EBBS will learn a chunk that creates structurally identical WMEs in future similar situations, those WMEs will not include internal, meta-knowledge that links it to the LTM structures that the WMEs that were originally generated by the subgoal were linked to. If the agent performs further reasoning that require that the LTM links exist – for example, the rule may use one of the new semantic memory tests to test that two short-term working memory structures were linked to the same semantic memory – the reasoning would match against the original results but would not match against the WMEs produced by the chunk. That is a problem because it could lead to different behavior. So, even though instance-based semantic memory solved many of the problems that EBBS had learning from agents that reason over LTM, it also introduced a new situation in which EBBS cannot learn a correct rule.

In Section 3.5, we also posited that our analysis of the underlying reasons for correctness issues - violations of the four necessary conditions for correct behavior summarization – can help architectures identify new correctness issues. If we examine the problem in those terms, we can see that this issue is an example of attempting to summarize reasoning that is not expressible in a single rule, or in this case, any rule. The learned rule produces different behavior because it doesn't have the language to express one of the effects that the original subgoal reasoning brought about. Specifically, the learned rule doesn't have the language to create the invisible links between the working memory structures it creates and the long-term memory structures they were based on.

Our Solution

To remedy this problem, we modified the expressibility of the architecture's rule language so that learned rules have the ability to effect the same changes that the subgoal reasoning did. Specifically, we added a new rule action that creates a link between a short-term working memory structure and a long-term one. At the end of a learning episode, EBBS will check if any of the WMEs in a result are linked to LTM structures. For each WME that is linked, EBBS adds a RHS action to the learned rule that will create a link between WME that the chunk adds and the LTWME that the original result WME was linked to. When the learned rule matches, the standard RHS actions will first create the new WMEs and then the new semantic memory linking actions will link them to the LTME elements that the original results were linked to. In summary, by increasing the expressibility of the architecture's rule language so that a rule can link WMEs it creates to LTMs, EBBS is able to learn a correct rule that produces effects identical to those of the original subgoal.

6.4 Strategy 5: Detect and Prevent Incorrect Rules

The final strategy that EBBS uses to avoid learning incorrect rules is to detect whether the behavior trace uses reasoning that cannot always be summarized correctly. If it finds evidence that it's trying to summarize reasoning that the algorithm cannot correctly summarize, EBBS discards the chunk. This forces the agent to re-deliberate in future similar situations, which will produce the agent's correct intended behavior.

This strategy is used in two ways. First, it is applied to problems that may not be solvable or do not have clear architecture-level solutions, for example implicit dependencies. Second, this strategy is used as a placeholder for proposed mechanisms that may be able to learn correct rules from traces that EBBS cannot currently handle. In fact, this has already occurred. Several of the correctness issues that are now handled by other mechanisms were previously avoided using detect and prevent.

We will now describe all of the types of problematic traces that this strategy can detect. The next few examples are correctness problems faced when summarizing reasoning that violates the requirement that any summarized reasoning must be reliably repeatable.

6.4.1 Case 1: Unreliable Knowledge from Operators Chosen Probabilistically

If the problem-solving in a subgoal involves operators that were selected probabilistically, chunking will not be able to summarize the agent's reasoning into a correct rule. For a rule to be correct, it must always produce the same result that the subgoal would have produced if the learned rule were not in production memory. In this case, even redeliberating with only the original rules may not produce the same results twice: since the operator selection was probabilistic, it's possible that an entirely different alternate operator would be selected next time which may lead to entirely different reasoning and results. Clearly the first learned rule would not be summarizing that alternate reasoning, which means the learned rule is incorrect.

EBBS can easily detect these cases while it's analyzing operator selection preferences to build the

ROSK set as described in Section 6.1.2. After EBBS annotates the operator with a ROSK set of operator preferences, it also adds flag to indicate whether the operator was selected in an unreliable way. A selection is considered unreliable if it used random selection, numeric preferences or reinforcement learning. In future learning episodes, EBBS checks for these flags as it backtraces through a behavior trace during the learning episode. If it encounters any rules that test operators that are flagged as selected unreliably, it will flag the entire learned rule as unreliable as well. And, if the prevent option is on, it will not add the rule to procedural memory. [15]

6.4.2 Case 2: Unreliable Knowledge from LTM

Long-term knowledge can be considered inherently unreliable and, hence, a potential source of correctness issues. There are two main reasons for this:

1. **The rationale for a LTM query is opaque**

   The underlying problem is that the retrieval mechanism of the memory system is opaque to the learning algorithm and may not reliably produce the same retrieval in future similar or even identical situations.

   As an example, consider an agent that reasons in a subgoal using knowledge structures retrieved from semantic memory. If there are multiple matches, the semantic memory system will return the memory structure with the highest activation level. (Soar's semantic memory activation is, by default, biased towards recently stored and retrieved memory structures, but it also has three other activation methods available: frequency, base-level and spreading.) Regardless of which activation method is used, the problem for online procedural learning is that activation levels of semantic memories can change as the agent uses semantic memory. It is entirely possible that activation levels change in a way that the same exact retrieval returns a memory structure that is different from the one used in original summarized reasoning. So, if the agent re-executed its original deliberate reasoning instead of the chunk, the new semantic memory structure returned may lead to different subgoal reasoning, which could then lead to a different result in the supergoal. The chunk is clearly not correct in such cases because it does not generate the same inferences as the original reasoning.

2. **Recalled LTM can change after the learning episode**

   Soar's semantic memory model allows the agent to intentionally add and modify its contents. Knowledge retrieved from a memory store whose contents can change can be problematic because, even if activation levels all stay the same, future retrievals may still return memories

[15]Instead, EBBS will convert the learned rule to a justification and adds it instead.

with different features. Since a different retrieval could produce different problem-solving, the rules learned from the initial case may not be correct.

To allow EBBS to detect these cases while backtracing, Soar now marks every instantiation with a flag indicating whether it is considered reliable in terms of LTM usage. Specifically, this flag is set to true if either (a) any of the conditions matched a WME that is linked to a LTM or (b) any of the conditions matched a WME created by an instantiation already flagged as unreliable. If EBBS encounters an instantiation with one of the flags set, it will mark the rule summarizing the trace as unreliable in terms of LTM usage. And, if the prevent option is on, it will not add the rule to procedural memory.

6.4.3 Case 3: Reasoning that Conflates a Conjunction of Contexts

An incorrect rule can be learned when multiple rules fire that test different structures in the supergoal *but create the same WME in the subgoal*. For example, consider an agent that has a rule which has only one action, which creates a single, specific WME in the subgoal. Assume this rule matches three different WMEs in the supergoal, so three instantiations are added, each of which creates the same WME *in the subgoal*. Another rule then tests this WME and adds a new Soar identifier as result to the supergoal, which causes a chunk to be learned.

In the original subgoal processing, the three matches produced one subgoal WME, and that one subgoal WME created a single Soar identifier as a result in the superstate. The chunk on the other hand will match three times, once for each of the WMEs that matched our original rule. Each one of those matches will create a Soar identifier, creating three Soar identifiers. This is different behavior than the original subgoal, which only created one Soar identifier. Even though the chunk is not overgeneral, it is incorrect because it produces different inferences than the original subgoal reasoning.

This is an example of procedural learning attempting to learn something that the architecture cannot express in its syntax. If Soar could express a disjunction of conditions in a chunk, it could learn a correct rule. Such a rule would have a clause in the disjunction for each of the instantiations that had previously created the subgoal WME that was repeatedly asserted.

Fortunately, this problem is easily avoided. If this type of reasoning is needed, agents can move the conflating WME to the superstate. The rule learned would then produce only one result regardless of the number of rules that repeatedly asserted that WME.

6.5 Final Comments on Improving Correctness

In this chapter, we have shown an example of how a cognitive architecture can use our strategies to learn rules that correctly summarize the reasoning an agent performs. We do this by showing how EBBS uses four of our five strategies to address chunking's deficiencies and eliminate or minimize the negative effects of the twelve issues listed in Figure 6.7.

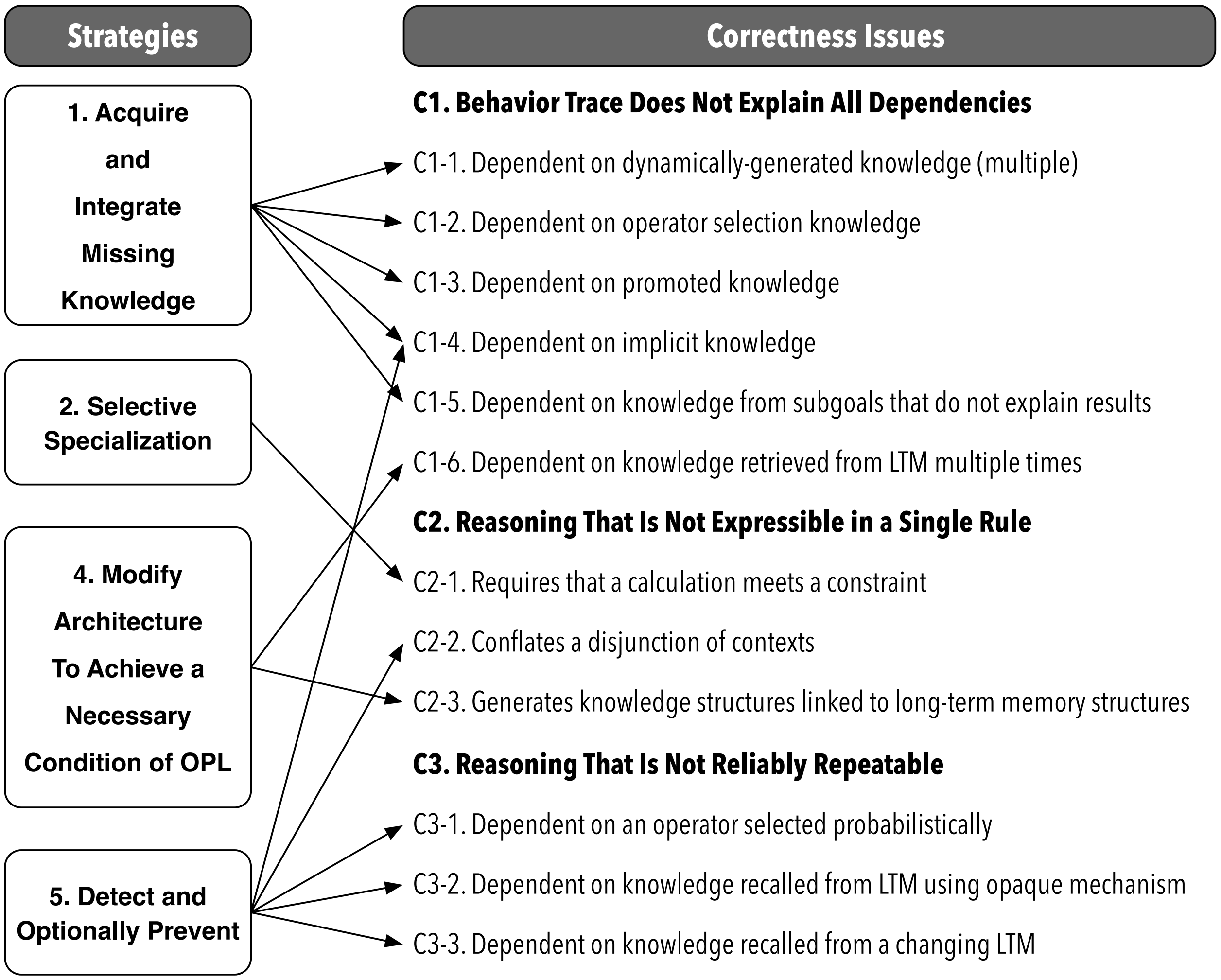

Strategies
Correctness Issues
1. Acquire and Integrate Missing Knowledge
2. Selective Specialization
4. Modify Architecture To Achieve a Necessary Condition of OPL
5. Detect and Optionally Prevent
C1. Behavior Trace Does Not Explain All Dependencies
C1-1. Dependent on dynamically-generated knowledge (multiple)
C1-2. Dependent on operator selection knowledge
C1-3. Dependent on promoted knowledge
C1-4. Dependent on implicit knowledge
C1-5. Dependent on knowledge from subgoals that do not explain results
C1-6. Dependent on knowledge retrieved from LTM multiple times
C2. Reasoning That Is Not Expressible in a Single Rule
C2-1. Requires that a calculation meets a constraint
C2-2. Conflates a disjunction of contexts
C2-3. Generates knowledge structures linked to long-term memory structures
C3. Reasoning That Is Not Reliably Repeatable
C3-1. Dependent on an operator selected probabilistically
C3-2. Dependent on knowledge recalled from LTM using opaque mechanism
C3-3. Dependent on knowledge recalled from a changing LTM

Figure 6.7: Applying the Five Strategies of EBBS to the 12 Correctness Issues.

Because the descriptions of how EBBS applies the strategies involve many, varied, complex issues, we will conclude this chapter by quickly summarizing them:

- In Section 6.1.1, we describe how EBBS remedies several deficiencies in the behavior trace by building or augmenting instantiations so that the trace provides a more complete description of the knowledge dependencies that were involved in the agent's reasoning. Specifically, we explain how EBBS builds/augments instantiations for subgoal meta-data, basic LTM recalls and rules with dynamic RHS actions.

- In Section 6.1.2 we describe how EBBS analyzes an agent's use of operator selection knowledge so that it can build a behavior trace that describes and learn rules that incorporate how knowledge was used to select the operators involved in the agent's reasoning.
- In Section 6.1.3, we describe why knowledge that has been promoted by being returned as a result can make it difficult for an online procedural learning algorithm to summarize further reasoning in a subgoal.
- In Section 6.1.3.1, we describe how EBBS repairs malformed rules, including the ones that can occur when summarizing a trace that tests promoted knowledge, by searching through working memory to create a set of conditions that connect disconnected aspects of the malformed rule to a goal.
- In Section 6.1.4, we describe how agent designers sometimes use implicit knowledge in the rules of their agents, why that is a problem for online procedural learning, and how EBBS can sometimes correct those issues by making the implicit knowledge explicit.
- In Section 6.2.1, we describe how EBBS uses selective specialization to learn correct rules when summarizing traces that perform mathematical reasoning.
- In Section 6.3.2.2, we describe how we performed a major architecture modification to eliminate learning issues that occurred when long-term memory recalls occurred in multiple subgoal and interacted in unintended ways. Specifically, we describe how we replaced a semantic memory system that produced hybrid long-term knowledge structures with an instance-based semantic memory.
- In Section 6.4, we describe how EBBS is able to algorithmically detect several types of behavior traces that may lead to incorrect rules: traces that test knowledge deemed to be unreliable, because it tested either operators that were selected probabilistically or knowledge recalled from LTM, and traces that conflate a conjunction of contexts.

In the next chapter, we evaluate our system and present experimental results that show how the collective mechanisms of EBBS have made significant progress towards achieving the online procedural learning desiderata we outlined in Section 2.6.

CHAPTER 7

Evaluation

In this section, we present our evaluation of how well our approach improves Soar's online procedural learning against our desiderata. Specifically, our main goal is to answer the following four questions:

- **E1**. Task Performance: Does the EBBS approach improve task performance as much or more than the current state of the art online procedural learning system?
- **E2**. Generality: Does the EBBS approach learn more optimally general rules than the current state of the art online procedural learning system?
- **E3**. Computational Costs: Does the computational overhead incurred by the EBBS approach prevent an agent from maintaining reactivity?
- **E4**. Correctness: Does the EBBS approach learn correct rules in situations where the current state of the art online procedural learning system cannot?

7.1 Evaluation Overview and Experiment Setup

To better answer these evaluation questions, we have developed a special version of Soar, which we call EVAL-SOAR, that is highly-instrumented and performs meta-analysis on its own reasoning and learning. This version of Soar allows us to both more accurately compare the performance of classical chunking and our EBBS approach and measure the impact that various mechanisms of EBBS has on the quality of rules learned. It does this by including the following capabilities:

- Highly-instrumented online procedural learning code that tracks several new metrics that we use to evaluate our approach.
- A testing framework that repeatedly runs agents under different settings with some control for randomness. This tool runs agents thousands of times to generate data for our analysis.

- Simulated classical chunking

  For practical and theoretical reasons, we generate data by simulating classic chunking using a version of EBBS in which identity-based generalization is replaced with chunking's heuristic-based mechanism. Using a simulated classic chunking mode allows us avoid re-implementing our testing framework in a historical version of the architecture, which we stopped updating and maintaining in 2012. So many aspects of the Soar architecture have changed since then that it would be difficult to generate data that we'd be confident are comparable. Using a simulated version of classical chunking also allows us to include agents in our evaluation that use reasoning that historical versions of Soar cannot correctly summarize. If some of our agents are run in an older version of Soar, they learn classic chunks that prevent them from solving the task. Because task failures would prevent us from fairly comparing generality, task performance and computational overhead across the two approaches, we allow the simulated classic chunking mode to use some of the correctness mechanisms of EBBS, for example the integration of operator selection knowledge into the rules learned. (This means that we are essentially comparing EBBS against an enhanced version of classical chunking, which implies that EBBS would provide further improvements over classical chunking than this chapter reflects.)

- The ability to turn off individual mechanisms of EBBS piecemeal. This allows us to explore the impact that individual mechanisms have on correctness and generality. The system can even turn off forward identity propagation, allowing us to examine what effect the DIGU algorithm's distributed approach has on efficiency.

7.1.1 Experiments

For each experiment, we use EVAL-SOAR's testing framework to repeatedly run the same agent using different learning algorithms and settings. Each experiment consists of one agent performing 100 trials. In each trial, there are three sets of agent runs, one for each learning approach. In each set of runs, the agent performs separate learning and post-learning runs to isolate the effects of the learned rules. We do this to avoid confounding factors in the initial learning run that make it more difficult to measure the benefits that the learned rules provide. For example, a learned rule can improve performance because it replaces deliberate reasoning with an immediate inference, but it can also improve performance simply because it serendipitously changed the agent's search path to one that more quickly led to a solution. By performing multiple non-learning runs using a fixed set of learned rules, we can test the qualities of those rules more accurately and more fairly compare the results across learning algorithms. Unless otherwise stated, all of the data in this chapter are derived from experiments that are structured as follows:

- **100 trials**
  1. **Non-Learning run**
     (a) One run with all procedural learning disabled that is used as a baseline.
  2. **Classical chunking runs**
     (a) One learning run (classical chunking enabled)
     (b) 50 post-learning runs: learning is disabled, but agent uses classic chunks learned in the first run.
  3. **EBBS runs**
     (a) One learning run (EBBS enabled)
     (b) 50 post-learning runs: learning is disabled, but agent uses EBBS chunks learned in the first run.

Because some of our evaluation agents do not terminate when they are not able to solve their task, our testing framework considers any run that doesn't find a solution within 50,000 decision cycles a failure.

7.1.2 Controlling for Randomness

To reduce the effects of the probabilistic aspects of agent execution, the testing frameworks sets the same starting conditions and random seeds in corresponding runs.[1] In other words, it ensures that run 1 of an EBBS learning run uses the same starting configuration and random number generator seed as run 1 of the non-learning and classic chunking runs. For agents that are deterministic, using the same random seed produces the same reasoning across runs. For agents that are not deterministic, for example the ones that perform a search through a problem space, the agent will only produce the same behavior as another agent using that seed until the point where the two agents do anything differently that uses a random number. This can happen, for example, if a learned rule provides knowledge that allows it to avoid a subgoal in which an operator is selected randomly from a set of proposed operators. When something like that occurs, the agent that avoided the subgoal will not "consume" a random number from the random number generator, while the agent that subgoals will. At that point, having the same random seed no longer ensures that the agent is facing the same situations. Nonetheless, it still provides benefits insofar as it allows each of the runs to explore the problem space differently in a way that is still replicatable.

[1]One way to reduce variability across runs is by setting the random number generator's initial seed. This forces the same sequence of pseudo-random numbers to be used by the architecture internally.

7.1.3 Conventions Used in Figures

In all of the figures of this chapter, we use red for results from the baseline non-learning run, blue for results from classic chunking runs, and green for EBBS runs. As shown in the sample figure of 7.1, the type of run can often also be found in the names of each agent. We use "NL" as shorthand for non-learning runs, "CC" for classic chunking runs, and "EBBS" for EBBS runs. Each bar in a figure indicates the average value of a metric and includes an error bar that reflect the standard deviation of the underlying data. If an agent fails to solve a task using a particular learning approach, we add a graphic that looks like a head with an X on its forehead.

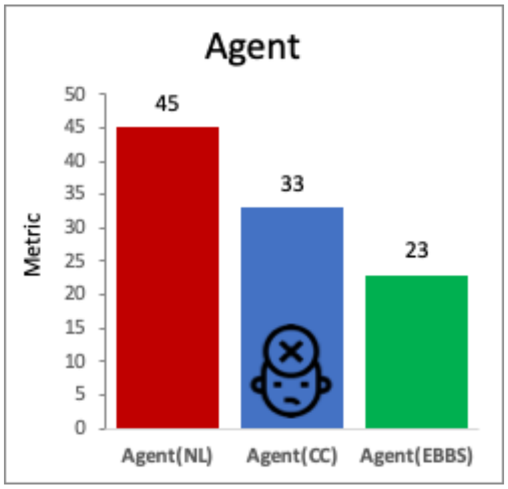


Figure 7.1: Sample figure used in this chapter.

All of the agents in this chapter and the data generated from them can be found at https://soar.eecs.umich.edu/downloads/files/EBBS_Evaluation_Agents_and_Data.zip.

7.2 Agents and Tasks Used in Evaluation

In this section, we describe the 14 agents that we use in our evaluation and the 8 different tasks they perform. We split the agents that we use for evaluation into two groups. The first group consists of four agents that were either developed or significantly updated after this research project was undertaken.[2] These four agents perform four different tasks: solving arithmetic problems, finding a path in a graph using A* search, solving a Sudoko-like game called Kenken, and parsing natural language sentences for interactive task learning. The second group consists of 10 different agents that solve 4 classic AI puzzles: BlocksWorld, Water-jug, Eight-puzzle, and Mice and Cats. These agents pre-date EBBS, and most of them solve their task using a form of search. All the agents were programmed by the original Soar agent programmer, John Laird.

One reason that these agents are used is simply necessity; these are the agents that we had available to us. But part of it is also intentional. Evaluating agents that were not designed with EBBS in mind and, in some cases, would not work at all with classical chunking, allows us to examine how well we are progressing towards an automatic mechanism that does not require agent alteration.[3]

[2]While some of the agents are recent and some were updated for this project, none of them were developed specifically to be used as an evaluation agent.

[3]To be fair, a few of these agents do have some updates. In most cases, the alterations are trivial required ones, for

But there are disadvantages to using these agents as well. For example, several of the agents we use in our evaluation were designed with classical chunking in mind. (A discussion of how agent designers can represent a problem in a way that avoids classical chunking inability to capture all of the generality of an agents can be found in Section 3.6.2.) Consequently, there is little to no additional generality for EBBS to capture. In those agents, we expect EBBS to learn the same or similar rules and provide little to no additional benefits over classical chunking.

The agents in the first group will be our focus, because they use representation and reasoning that is more varied and closer to the types of reasoning that real-world agents, who motivated this work, use.

7.2.1 Task 1: Solving Multi-column Arithmetic Problems

The goal in this task is to perform arithmetic and subtraction between two three-digit numbers. The agent used for this task, which consists of 99 rules, formulates the problem in terms of multiple columns, using carries or borrows, and does not use any math functions. Instead, the agent performs math using one or more tables of simple math facts and symbol manipulation. For addition problems and one of the strategies for subtraction problems, the agent uses a table of all single digit addition facts. For the second subtraction strategy, it uses a table of simple subtraction facts and another table of addition-by-ten to single digits facts.

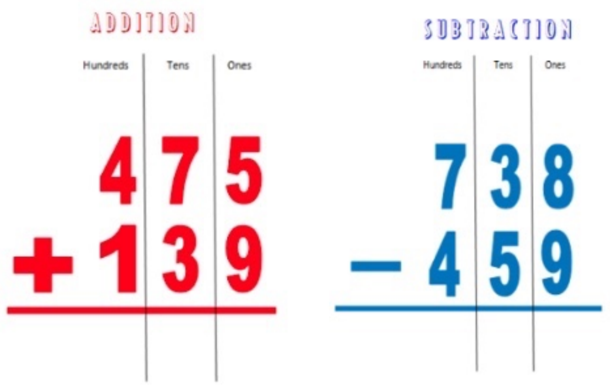


Figure 7.2: Examples of multi-column arithmetic problems.

example changes to the syntax of commands or the addition of domain-specific singletons. But, there are a few agents that were updated to replace reasoning that relied on limitations or idiosyncrasies of classical chunking that no longer exist with EBBS.

7.2.2 Task 2: Finding a Path in a Graph using A* Search

The goal in this task is to take a waypoint graph and find the minimal path from a starting location to a target location. The agent used for this task consists of 123 rules and uses iterative A* search to find the shortest path through the graph.

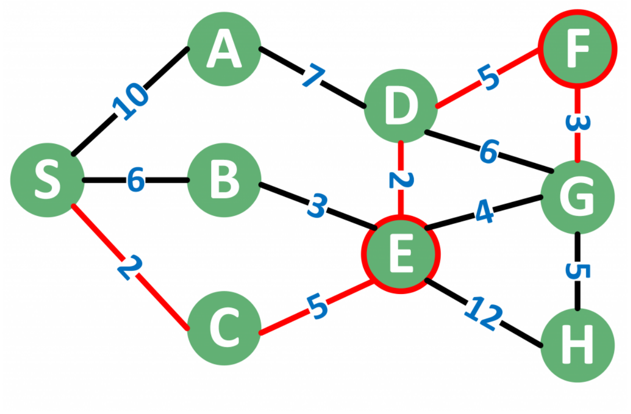


Figure 7.3: A representation of a graph search problem.

7.2.3 Task 3: Solving a Kenken Puzzle

Kenken is very similar to the classic game Sudoku. Like Sudoku, the goal of Kenken is to fill a grid with integers so that every row and every column contains all the integers from 1 through N. In Kenken, though, there are additional mathematical constraints that must be met in subsets of cells, called cages. (In Figure 7.4, the cages have a heavy outline.)
The kenken agent consists of 284 rules and uses reasoning that combines constraint satisfaction and forward search. It is important to note that the Kenken agent is not able to complete the task when learning classic chunks, so it cannot be used to compare classical chunking to EBBS.

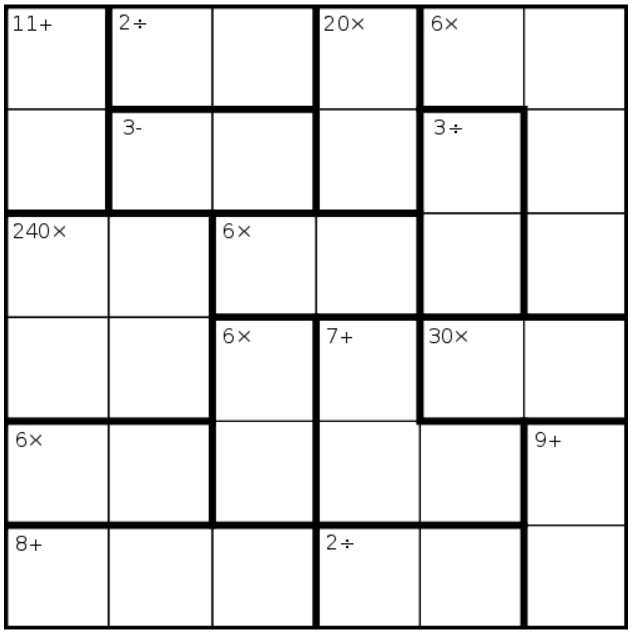


Figure 7.4: A Kenken puzzle.

7.2.4 Task 4: Parsing Natural Language

The NLP agent parses natural language utterances into a format that the ROSIE instructable agent can use. This agent, which consists of 443 rules, utilizes ideas from embodied construction grammars and performs left-to-right incremental parsing with repair.

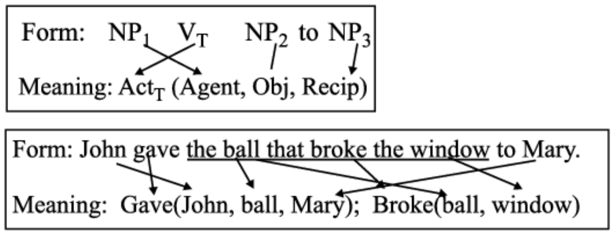

Form: NP1 VT NP2 to NP3
Meaning: ActT (Agent, Obj, Recip)
Form: John gave the ball that broke the window to Mary.
Meaning: Gave(John, ball, Mary); Broke(ball, window)

Figure 7.5: An example of a parsed sentence.

7.2.5 Task 5: BlocksWorld

As described by Russell and Norvig (2010), "the blocks world is a planning domain in artificial intelligence. The algorithm is similar to a set of wooden blocks of various shapes and colors sitting on a table. The goal is to build one or more vertical stacks of blocks. Only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved."

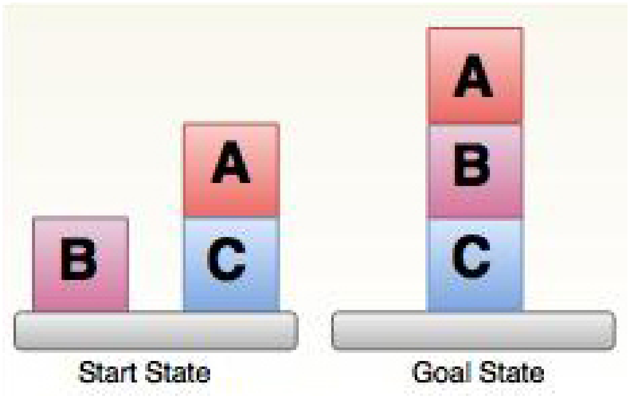


Figure 7.6: The blocks-world task.

- BlocksWorld Operator Subgoaling Agent
  This agent consists of 20 rules and uses a deterministic strategy that includes means-ends analysis and operator subgoaling. Means-ends analysis involves proposing operators that can achieve part of the goal, even if those operators may not apply to the current state.
- Blocks-world Hierarchical Agent
  This agent consists of 52 rules and solves the blocks-world problem using a hierarchy composed of three levels of problem spaces. This agent includes evaluation knowledge that eliminates search/uncertainty at every level, so its behavior is deterministic.
- Blocks-world Look-Ahead Agent
  This agent consists of 99 rules and solves the blocks-world problem using a single problem space and non-deterministic look-ahead search.
- Blocks-world Look-Ahead State Evaluation Agent
  This agent consists of 104 rules and solves the blocks-world problem using a single problem space and non-deterministic look-ahead search. Unlike the standard look-ahead ahead, this

agent will not lock onto one particular path. This agent instead evaluates the states that the search considers, which allows it to perform more exploration of the search space.

- Blocks-world Hierarchical Look-Ahead Agent
  This agent consists of 158 rules and solves the blocks-world problem using a hierarchy composed of three levels of problem spaces with non-deterministic look-ahead search in the middle problem space. For the top and bottom problem spaces, there is pre-existing selection knowledge that is sufficient to eliminate search at those levels.

7.2.6 Task 6: Water-Jug Puzzle

The water-jug problem is a classic logic puzzle involving a water tap and 2 containers, each with a different capacity. Each container has no markings except for that which gives you it's total volume. Jugs can be filled to the top using the running tap. To solve the puzzle, the containers must be filled and emptied until one of the jugs has the target amount of water. In our formulation, there are two jugs, one that holds 3 units of water and one that holds 5, and the goal of the task is to find the sequence of emptying and filling actions that leads to the three gallon jug containing exactly one gallon of water.

Figure 7.7: The water-jug puzzle.

Our evaluation includes three different water-jug agents: water-jug hierarchical (22 rules), water-jug tie (30 rules), water-jug look-ahead (97 rules). Because these agents utilize search strategies that are very similar to the ones used by the blocks-world agents, we will not describe each one here individually.

7.2.7 Task 7: Eight Puzzle

The eight-puzzle is a sliding puzzle having 8 square tiles numbered 1–8 in a frame that is 3 tiles high and 3 tiles wide, leaving one unoccupied tile position. Tiles in the same row or column of the open position can be moved by sliding them horizontally or vertically, respectively. The goal of the puzzle is to place the tiles in numerical order.

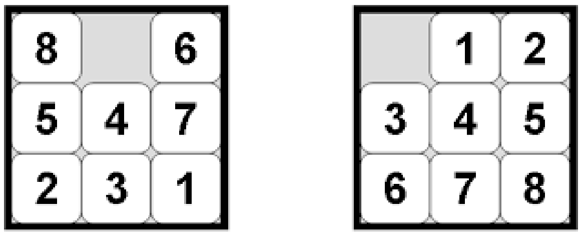


Figure 7.8: The 8-puzzle task.

The agent we use to evaluate this task consists of 87 rules and solves the puzzle using look-ahead search with a heuristic evaluation function.

7.2.8 Task 8: Mouse and Cats

In the mouse and cats problem, three mice and three cats must cross a river using a boat which can carry at most two animals, under the constraint that, for both banks, if there is a mouse present on the bank, they cannot be outnumbered by cats. (In this world, it always takes two cats to trap and eat a mouse).

The agent we use to evaluate this task consists of 95 rules and solves the puzzle using look-ahead search with a heuristic evaluation function.

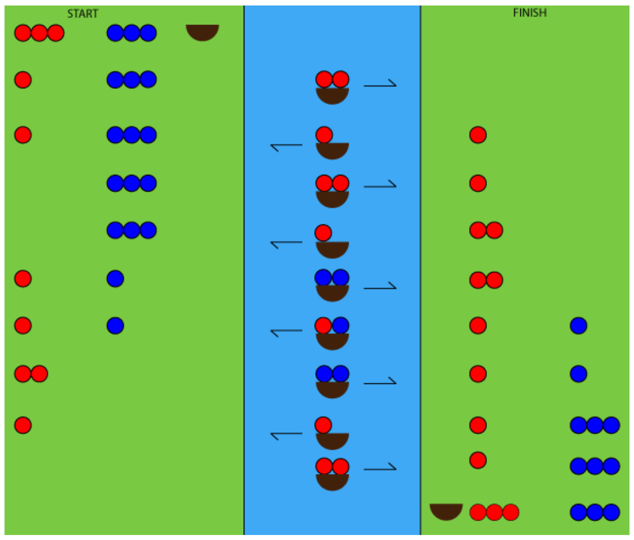


Figure 7.9: The mouse and cats task.

7.2.9 Special Allowances for Some Agents

In our experiments, we found that some of the agents that used a look-ahead search strategy need to solve multiple problems to learn enough rules that they impact performance. The Mouse and Cats Planning agent needs about fifteen runs. The BlocksWorld Look-ahead needs about six, and the Water-Jug Look-ahead agent needs two. Consequently, we allow those agents multiple learning runs instead of just one.

This issue also exists for the arithmetic agent but is handled differently. Unlike the other agents, the arithmetic agent has Soar rules that generate a specified number of math problems at the beginning of a run that the agent then solves. Instead of increasing the number of runs, we allow it to solve a set of 1000 math problems. On a related note, we do not include any of the processing that was used to generate those math problem in any of our metrics.

7.3 Evaluating Task Performance Improvements (E1)

In this section, we attempt to answer whether the EBBS approach improves task performance and how that improvement compares to the performance improvement that the current state of the art online procedural learning system, chunking, offers.

Note that evaluating task performance will not include experiments that measure the time to solve the task. While timing metrics are also a measure of performance, this section focuses on metrics that measure *task performance* in terms of the quality of the solution. We leave our analysis of timing data to Section 7.5, where we evaluate computational overhead.

7.3.1 Task Success

The most obvious metric to use to evaluate task performance is whether the agent successfully solved the problem. In terms of comparing agents and learning approaches, task success unfortunately is a metric of limited interest in these agents; nearly all of our agents in group one can solve their task with all learning approaches, as shown in Figure 7.10.

The two notable exceptions are the runs where the Kenken agent and the NLP agent use classical chunking. In those two cases, we can claim that EBBS trivially improves task success more than classical chunking because classic chunks prevent task success.

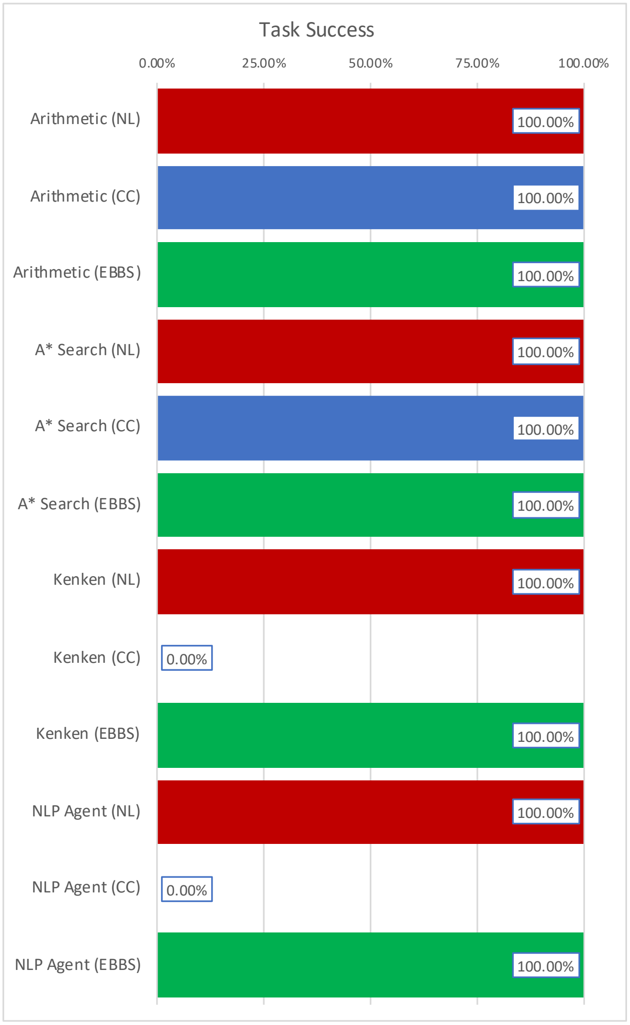


Figure 7.10: Percentage of runs in which agents from group one achieve their task goal.

In the second set of agents, there are only four cases where agents do not reliably solve their task, as we can see in Figure 7.11. And, even in the two agents that do fail to reliably find a solution, blocks-world hierarchical look-ahead and water-jug look-ahead, both EBBS and classical chunking increase the number of successful runs by the same amount. So, *in terms of comparing learning algorithms*, task success reveals nothing of interest in these agents.

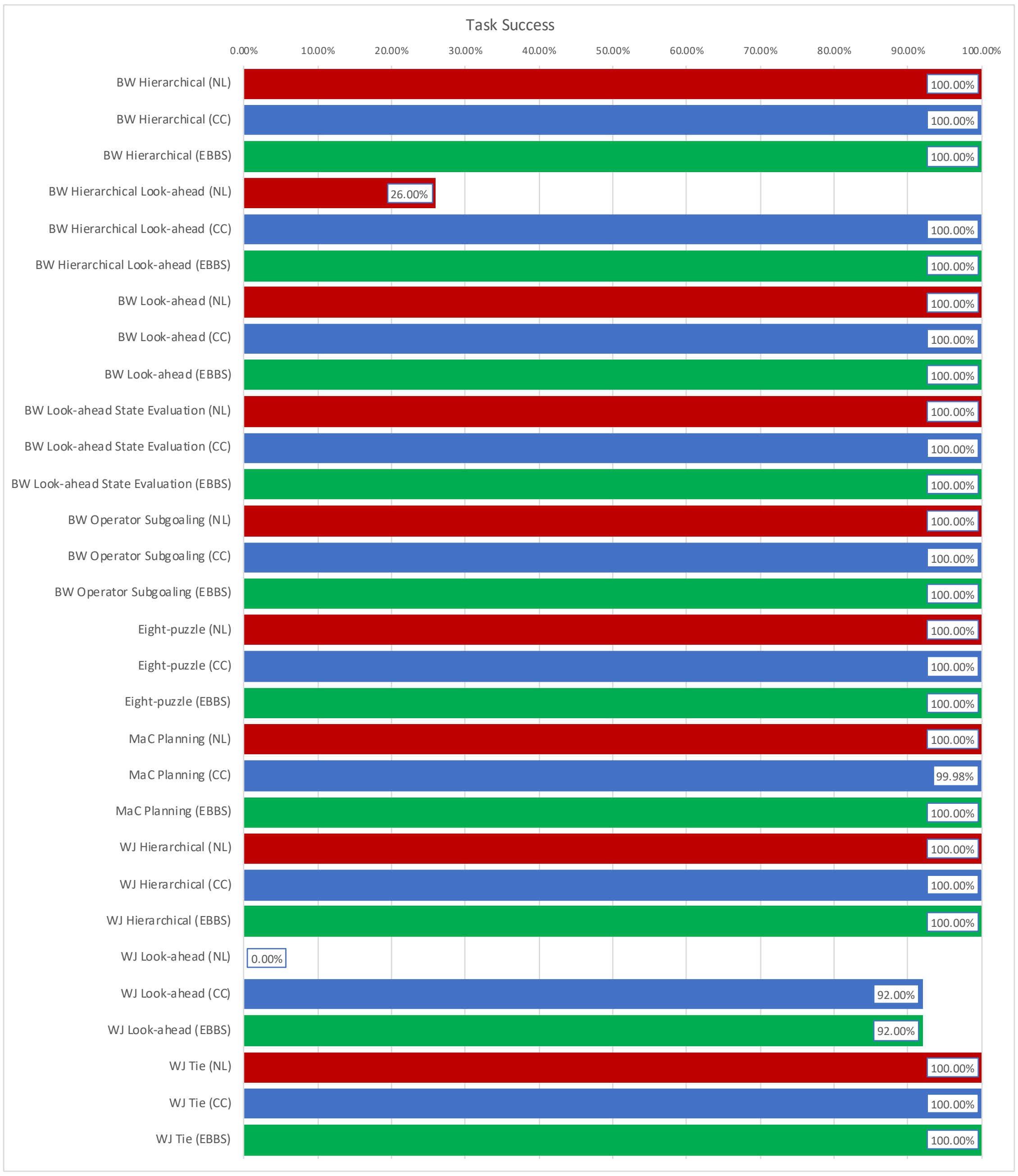


Figure 7.11: Percentage of runs in which an agent from group two achieves its task goal.

7.3.2 Number of Decisions Cycles Needed to Solve Task

The second and more important metric we use is the number of Soar primitive cognitive execution cycles (decisions) that are needed to achieve success in a task. Because the reasoning in a subgoal can span multiple decision cycles and a learned rule fires in a single decision cycle, an agent that learns rules that replaced subgoal reasoning should improve its task performance by solving the problem in fewer decision cycles.

7.3.2.1 Evaluation of Agent Group 1

As shown in Figure 7.12, EBBS reduces the number of decision cycles to solve a task by more than a factor of 2 in both the arithmetic agent and the A* search agent. And in the Kenken and NLP agents, EBBS rules dramatically reduce the number of decision cycles needed to solve the task while classic chunks prevent task success.

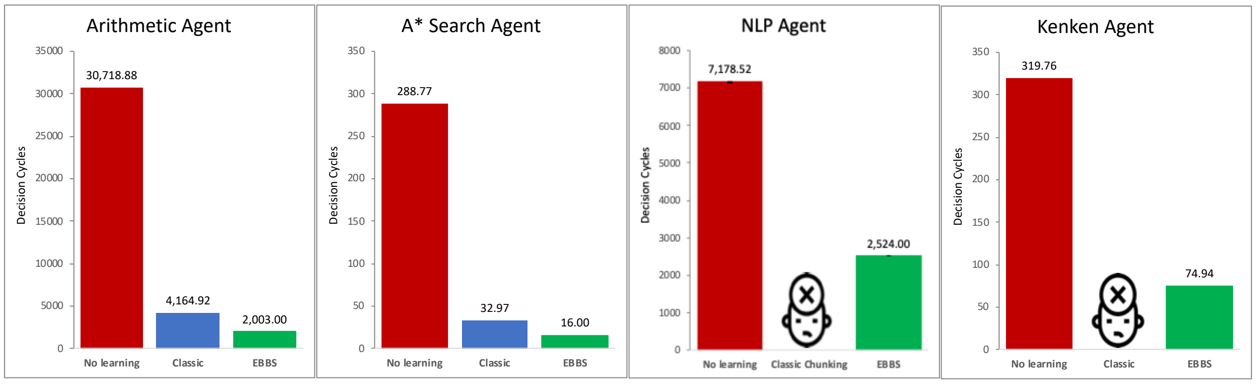


Figure 7.12: The number of decision cycles needed to solve the tasks of the agents in group one shows that EBBS reduces the number of decision cycles needed to solve a problem more than classic chunking does.

7.3.2.2 How Learning Rate Affects Task Performance

One of the reasons that we argued that learning optimally general rules may provide benefits over more specific rules is that they are more likely to apply in future situations because they describe a larger space of situations. But consider what happens when classical chunking is given a sufficient number of varied training examples that it also learns a set of rules that cover the entire space of situations that the original reasoning applies to. At that point, having more general rules may no longer provide any benefit. Both algorithms possess a set of rules that replace the same space of reasoning. Consequently, we must examine an agent's performance before it has learned that full set. Specifically, we examine how our metrics are impacted by an online procedural learning algorithm's learning rate, *how quickly it learns a set of rules that can be used in the full space of* *situations that the original rules they summarize can be used in.* This allows us to examine whether there are benefits to using a procedural learning algorithm with a faster learning rate.

Unfortunately, most of our agents do not lend themselves to this analysis because the limited complexity of their tasks and the representations they use allow both EBBS and classical chunking to learn all of the rules that cover the space of original reasoning during only a single (or few) examples. The arithmetic agent, though, is different and perfect for this experiment because we have fine-grained control over how much learning the agent performs. As discussed in Section 7.2.9, the arithmetic agent has Soar rules that generate a number of math problems at the beginning of a run that the agent then solves. And, as we show in the figures of this section, this is needed because the arithmetic agent requires far more than one example, especially when using classical chunking, to learn a set of rules that can replace its deliberate reasoning.

In the previous experiment, which was shown in Figure 7.12, we gave the arithmetic agent 1000 practice arithmetic problems to solve with learning on, and then tested the effectiveness of the learned rules by having the agent solve 50 different sets of 1000 problems with learning off. We now present an incremental version of that experiment where we gradually increase the amount of learning the arithmetic agent performs each run. So, in the first run, the agent solves exactly one problem with learning on and then solves 50 different sets of 1000 problems with learning off. (From this point on, we refer to the problems an agent solves with learning on as its *practice problems.*) In the second run, it solves two practice problems before solving 50 different sets of 1000 problems with learning off, and so forth. Figure 7.13 shows how many decision cycles an agent uses on average to solve 1000 math problems when given 1 to 200 practice problems.

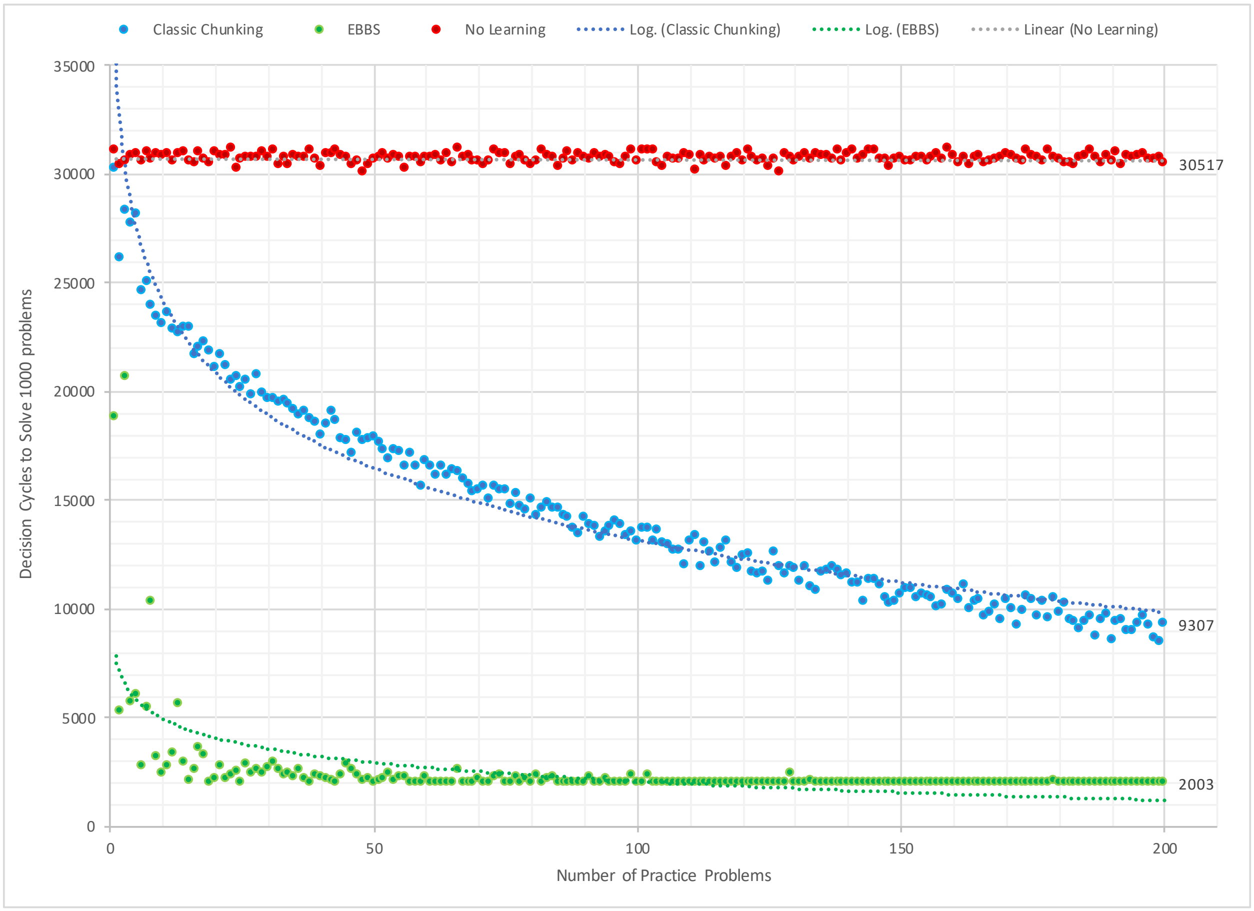


Figure 7.13: How the number of practice problems solved (1-200) affects number of decision cycles needed to solve 1000 problems.

EBBS needs only about a dozen practice problems to achieve its maximal performance. In contrast, an agent that learn classic chunks take roughly between 9k to 25k more decision cycles, depending on the number of practice problems it was given, to solve the same set of problems. And, if we extend this experiment out to 25k practice problems, we see that it takes thousands of practice problems before classic chunking can decrease the number of decision cycles to solve a set of 1000 math problems as much as EBBS.

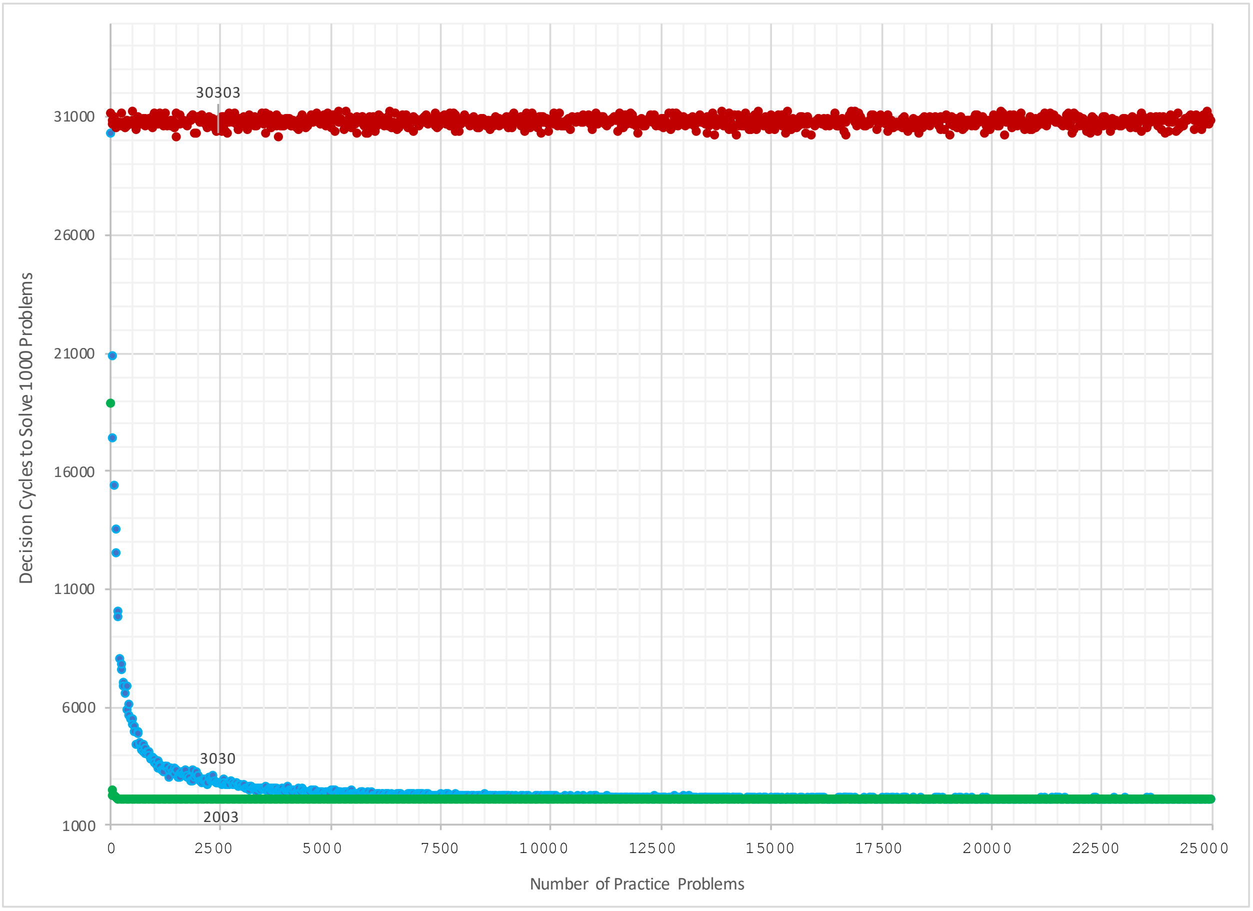


Figure 7.14: How the number of practice problems solved (1-25000) affects number of decision cycles needed to solve 1000 problems.

7.3.2.3 Evaluation of Agent Group 2

In our second group of agents, some agents that learned EBBS rules solved their tasks in fewer decision cycles than the ones that learned classic chunks, but not by a significant amount. As we can see in Figure 7.15, EBBS rules and classic chunks perform comparably for all of these agents.[4]

[4]Some of these agents, particularly the look-ahead agents, have a lot of noise in their results because of the variable nature of their search strategy.

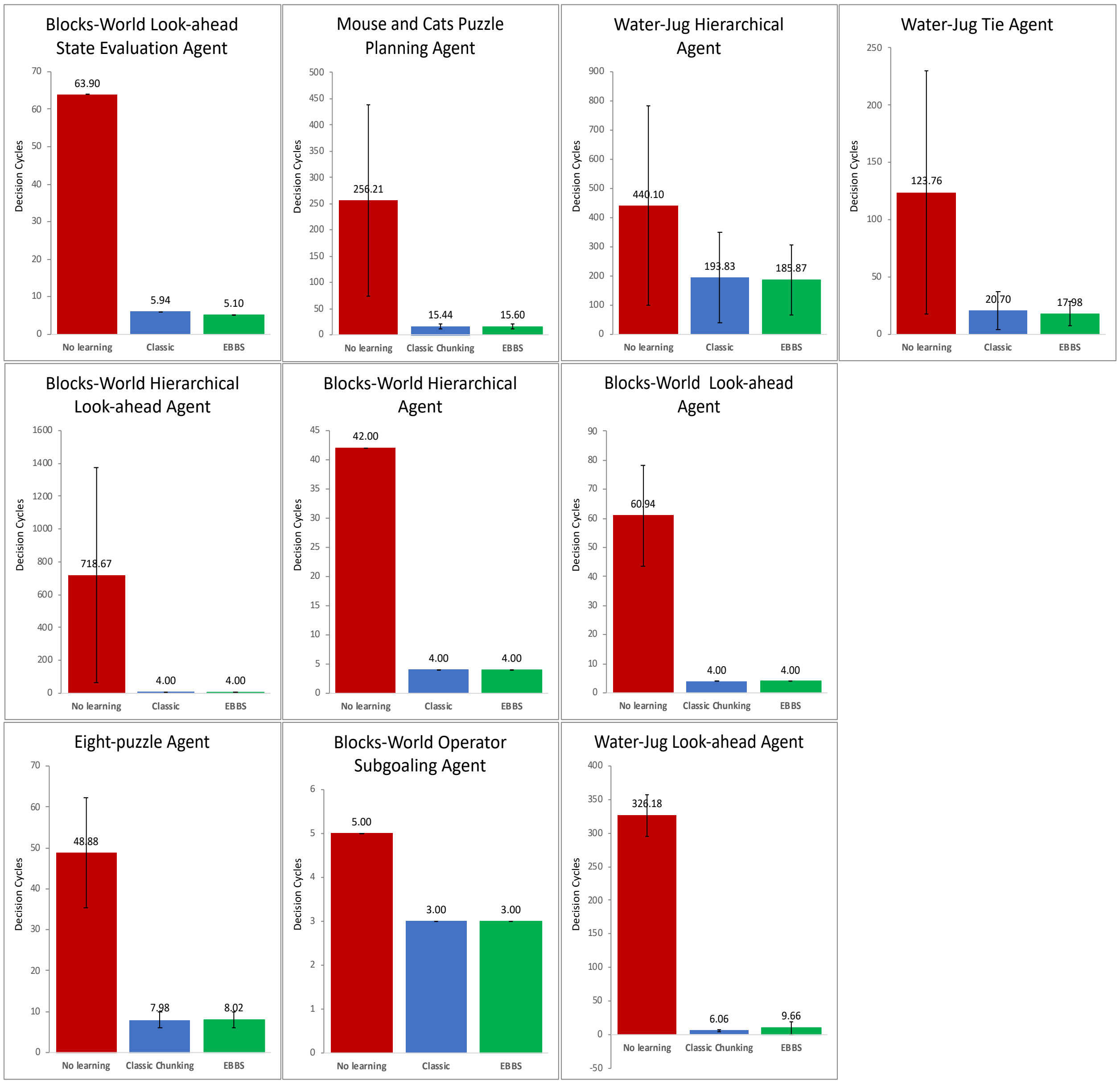


Figure 7.15: The number of decision cycles needed to solve classic AI problems show that both EBBS and classical chunking decrease the number of decision cycles to solve a task by approximately the same amount.

As shown in Figure 7.16, EBBS's faster learning rate does provide some task performance benefits to a few of the agents in group two. In the initial learning run, the run in which the agent starts with no learned rules and learns rules as it solves a problem, EBBS is able to solve the problem in fewer decision cycles than classic chunking in two of these agents. In BlocksWorld Hierarchical, the EBBS agent solves the task in 20% fewer decision cycles, and, in BlocksWorld Hierarchical Look-Ahead, it solves its task in approximately half the decision cycles. (The other agents completed their tasks in their learning run in the same number of decision cycles as they did in their post-learning runs.)

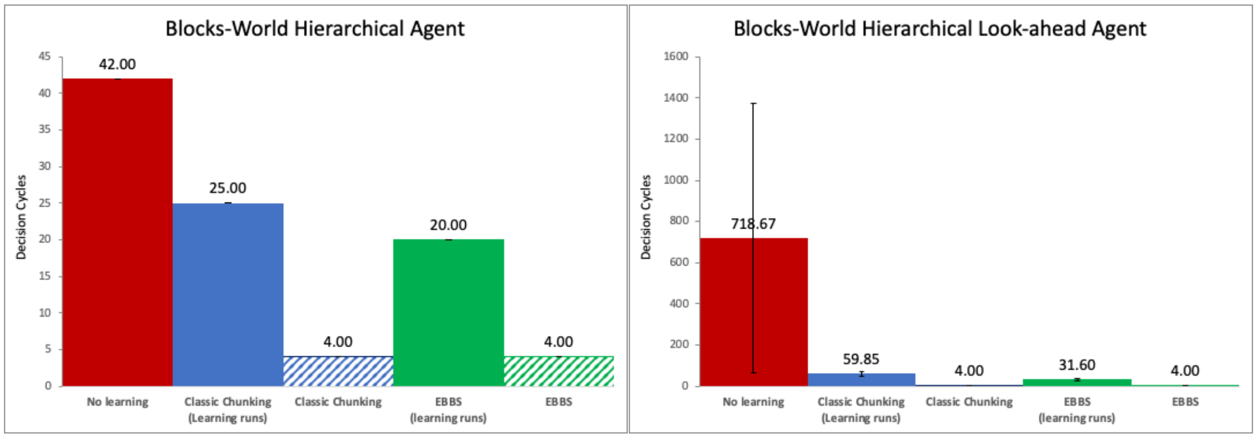


Figure 7.16: This figure shows the number of decision cycles needed to solve a task for both the learning and post-learning runs. EBBS reduces the number of decision cycles to solve a problem during the learning run more than classical chunking does for these two agents.

7.3.3 Conclusions on our Evaluation of Task Performance

Our experiments have shown strong evidence that EBBS improves task performances as much as classical chunking, and in some cases more. In domains or tasks where learning rate is important, the improvements can be dramatic.

7.4 Evaluation of Optimal Generality (E2)

In this section, we examine one of the most important of our evaluation criteria, whether the EBBS approach learns rules that capture more of the generality of the underlying reasoning than the current state of the art online procedural learning system, chunking.

7.4.1 Types of Knowledge That Learned Rules Reason Over

The key reason that classical chunking learns overly specialized rules is that it can only capture the generality of reasoning that involves Soar identifiers. It cannot capture any of the generality found in reasoning that involve other knowledge types; for example, classic chunking cannot capture reasoning that tests that two objects or concepts have the same string feature or that a feature is a number within a certain value range. Consequently, the rules that it learns will always use specific values instead of variables for those tests. This means that when a classic chunk fires, the variables it does include will always match Soar identifiers and never match other knowledge types. If EBBS and DIGU are indeed capturing generality from reasoning over other data types that classic chunking cannot, some of the variables in the EBBS rules should also match those other data types.

So, the first metric *tracks all learned rule firings and calculates how many times each type of knowledge is reasoned about generally, i.e. tested by a variable in a learned rule firing.* Whenever a learned rule fires, EVAL-SOAR examines each condition of the rule and compares it with the WME that matched. If an element is a variable and it matches a Soar identifier in a WME, then the variable match metric for identifiers is incremented. Similarly, if an element in the rule is a variable and it matches a string or number in a WME, then the variable match metric for a string match or numerical match is incremented.

7.4.1.1 Evaluation of Agent Group 1

Figure 7.17 shows the types of knowledge that were tested when the learned rules of group one fired.

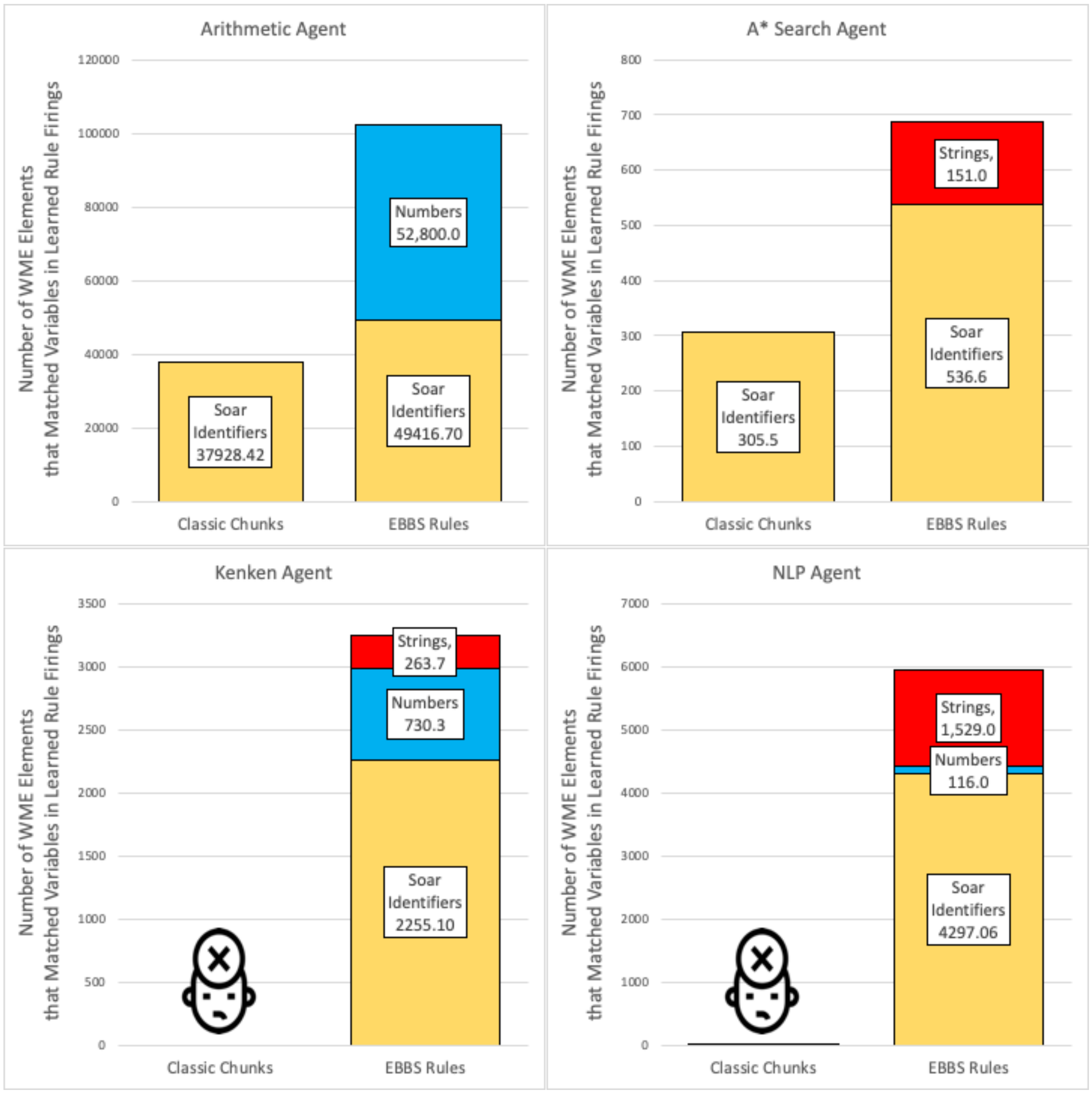


Figure 7.17: The types of knowledge that chunks and EBBS rules match against. When chunks fire, they match only against Soar identifiers; when EBBS rules fire, their variables match against multiple types of knowledge, evidence that EBBS captures and uses generality from an agent trace that chunking does not.

In all four runs that used classic chunking, the variables always match Soar identifiers, as expected. In contrast, the EBBS rules learned by the A* Search, Kenken and NLP agents contained variables that test a combination of Soar identifiers, strings and numbers. And, in the arithmetic agent, more variables in the EBBS rules matched numerical features than matched Soar identifiers.

If we examine two rules learned by the arithmetic agent, as shown in Figure 7.18, we see how the

EBBS rule on the left represents a more general description of the same reasoning summarized by the classical chunk on the right. If we examine the elements of the classic chunk that are highlighted in blue, we see that they each must match a specific value. In contrast, the corresponding elements in the EBBS rules, which are highlighted in green, can match any value that allows the relationships between the variables and any constraints on their values to hold. This is the generality in the underlying reasoning that EBBS is able to capture and classic chunking cannot.

```
sp {process-column*apply*result*EBBS
    (state <s1> ^operator <o1>
                ^ problem <a1>
                ^one-fact <o2>
                ^one-fact <o3>
                ^top-state <t1>
                ^arithmetic <a2>
                ^arithmetic <a3>)
    (<o1> ^name process-column)
    (<a1> ^operation subtraction
          ^current-column <c1>)
    (<c1> -^new-digit1 <n1>
          ^digit2 <d1>
          ^digit1 { <d2> < <d1> }
          ^next-column <n2>)
    (<n2> ^new-digit1 <n3>
          ^digit1 { <d3> < <o3> }
          ^next-column <n4>)
    (<n4> ^new-digit1 <n5>
          ^digit1 { <d4> >= <o2> })
    (<a3> ^add10-facts <a4>
          ^add10-facts <a5>)
    (<a4> ^digit1 <d2>
          ^digit-10 { <d5> >= <d1> })
    (<a5> ^digit1 <d3>
          ^digit-10 { <d6> >= <o3> })
    (<a2> ^subtraction-facts <s2>
          ^subtraction-facts <s3>
          ^subtraction-facts <s4>)
    (<s2> ^digit1 <d5> ^digit2 <d1>
          ^result <r1>)
    (<s3> ^digit1 <d6> ^digit2 <o3>
          ^result <n3>)
    (<s4> ^digit1 <d4> ^digit2 <o2>
          ^result <n5>)
    -->
    (<c1> ^result <r1>)
}
```

```
sp {process-column*apply*result
    (state <s1> ^operator <o1>
                ^problem <a1>
                ^one-fact 1

                ^top-state <s1>
                ^arithmetic <a2>
                ^arithmetic <a3>)
    (<o1> ^name process-column)
    (<a1> ^operation subtraction
          ^current-column <c1>)
    (<c1> -^new-digit1 <n1>
          ^digit1 0
          ^digit2 7
          ^next-column <n2>)
    (<n2> ^new-digit1 9
          ^digit1 0
          ^next-column <n3>)
    (<n3> ^digit1 5
          ^new-digit1 4)
    (<a3> ^add10-facts <a4>)

    (<a4> ^digit1 0
          ^digit-10 10)


    (<a2> ^subtraction-facts <s2>
          ^subtraction-facts <s3>
          ^subtraction-facts <s4>)
    (<s2> ^digit1 10  ^digit2 1
          ^result 9)
    (<s3> ^digit1 5  ^digit2 1
          ^result 4)
    (<s4> ^digit1 10  ^digit2 7
          ^result 3)
    -->
    (<c1> ^result 3)
}
```

Figure 7.18: The EBBS rule on the left captures generality of the agent's reasoning, shown in green, that the chunk on the right does not.

Note that the classic chunk also has fewer conditions than the EBBS rule. This is because, as previously discussed in Section 5.7.3, classic chunking collapses multiple conditions that test the same supergoal WME into a single condition, which is a consequence of learning from a WME trace instead of a trace of the underlying rules that fired.

7.4.1.2 Evaluation of Agent Group 2

In Figure 7.19, we see that seven agents in group two also learn and use EBBS rules that reason generally over knowledge types that the classic chunks do not reason over.

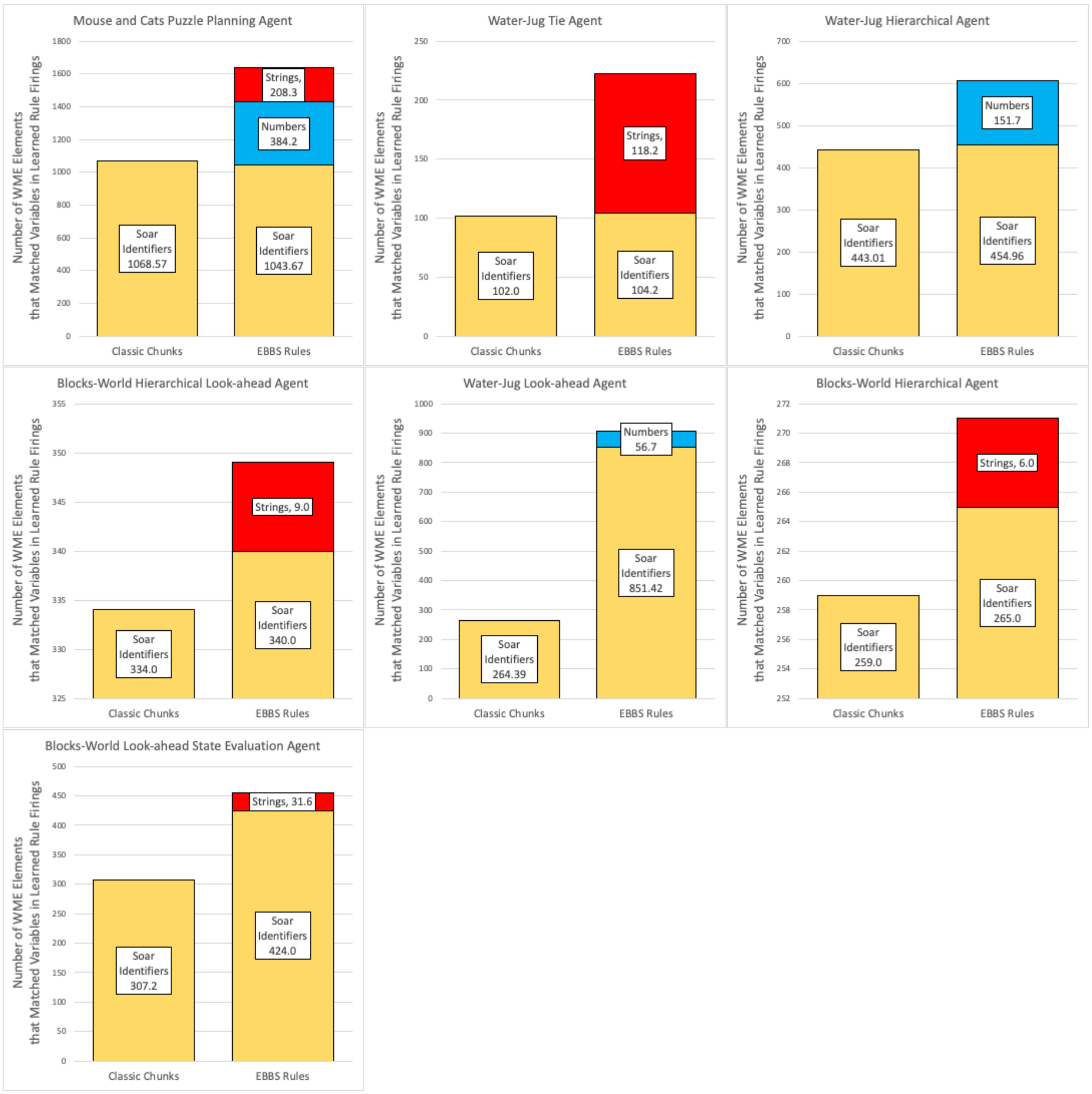


Figure 7.19: The types of knowledge that chunks and EBBS rules match against in seven of the agents of group two. We see evidence that EBBS captures and uses generality that chunking cannot in

In the final three agents, as shown in Figure 7.20, we see that both EBBS rules and classic chunks only match Soar identifiers. This shows that these agents' rules only use variables to match against elements known to be Soar identifiers. Because these agents do not reason generally about other knowledge types, there is no additional generality for DIGU to capture that classical chunking cannot already capture. This leads both classical chunking's heuristic-based variablization and EBBS's identity-based variablization to produce similar if not identical rules.

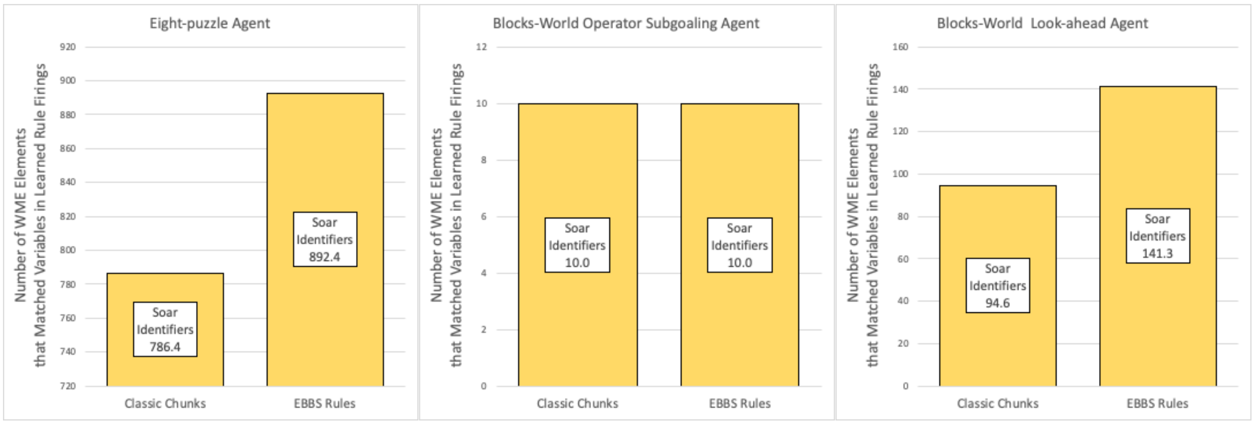


Figure 7.20: The types of knowledge that chunks and EBBS rules match against are the same in these three agents; both EBBS rules and classic chunks only reason generally about Soar identifiers.

Note that we could have also generated a similar metric for the types of knowledge used *by analyzing the match that gave rise to the rule*. Such a metric would likely have shown that EBBS captures additional generality that this metric does not show; this metric ignores generality captured by rules that are learned but do not find an opportunity to fire, a common occurrence. We chose to only include learned rule firings, though, because we felt it was more interesting and informative. It not only shows that generality is being captured by EBBS rules, but it also shows that that generality is being *used* by the agent.

In conclusion, this metric has shown clear evidence that EBBS captures and uses generality from the reasoning of all four of the agents in group one and most of the agents in group two that chunking does not.

7.4.2 Number of Rules Learned

The next metric we use to evaluate whether the rules that are learned by EBBS are more optimally general than those learned by chunking is the *number of rules learned*. While the absolute value of this metric is not meaningful in and of itself, it can be used to compare two procedural learning algorithms. If EBBS is able to capture more of the generality when summarizing the same reasoning traces, it should learn fewer rules.

One limitation of this metric is that it is difficult to guarantee that two agents will actually summarize the same behavior traces, most notably in agents that are searching through a problem space. Even when given the same random seed and starting configuration, two agents may end up summarizing different traces. If so, it is not clear whether the agent is learning fewer rules because its search path happened to find a solution quickly or because it captured an agent's reasoning in fewer, more optimally general rules.

We also include data on whether the two approaches learn rules that are actually identical. To calculate this value, we take advantage of Soar's RETE network, which reports when a new rule added to the RETE is a duplicate of a rule already in procedural memory. After the learning run of each approach, we save the rules learned and then load them both into an agent to see how many rules that the RETE reports as duplicates. In the following figures, if this mechanism detects any identical rules, the count will be superimposed on the right-most bar, the ones that indicates the number of rules that EBBS learns. Note that this value should be considered a minimum value, because, unfortunately, the way we detect identical rules is not completely reliable; the RETE sometimes fails to detect duplicate rules that have superficial differences.

7.4.2.1 Evaluation of Agent Group 1

Figure 7.21 shows the number of rules that compose an agent and the number of rules learned after each of the agents solves a single problem under both learning approaches.

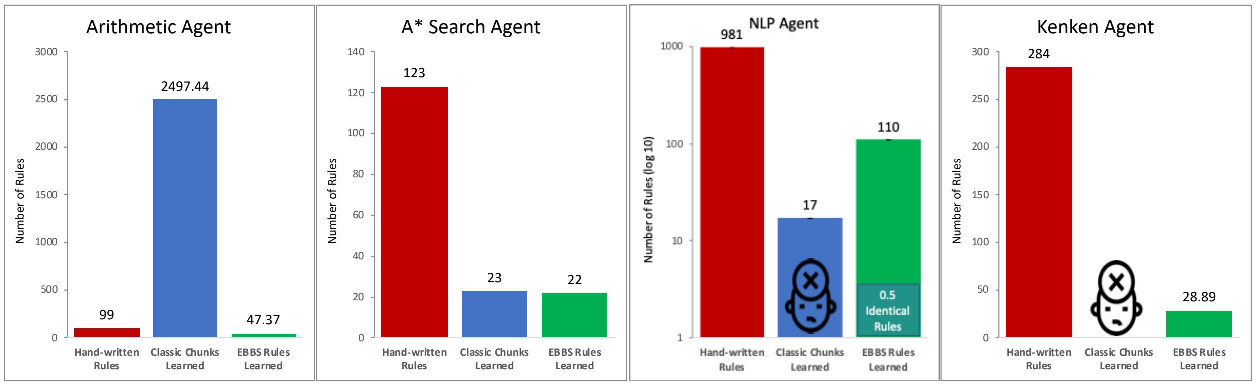


Figure 7.21: The number of hand-written rules and the number of rules learned using classical chunking and EBBS for each of the four agents of group one.

For the arithmetic agent, we find that EBBS summarizes the reasoning performed in many fewer rules (47) than classical chunking (2498). This is strong evidence that EBBS learns more optimally general rules than chunking in that agent. In the A* agent, though, EBBS learns only one fewer rules than classical chunking. Because our mechanism to detect identical rules did not indicate that any of the rules are identical, we gain no insight from this metric on whether the rules it learned are more general. Finally, because classical chunking is not able to learn correct rules that summarize the reasoning of the Kenken agent or the NLP agent, we trivially assert that EBBS is learning more optimally general rules from the traces of those agents than classical chunking.[5]

7.4.2.2 Evaluation of Agent Group 2

In seven agents, we find that EBBS summarizes the reasoning performed in fewer rules than classical chunking, as shown in Figure 7.22. This suggests that, in some situations, EBBS learns more optimally general rules than chunking, even in these simpler agents.

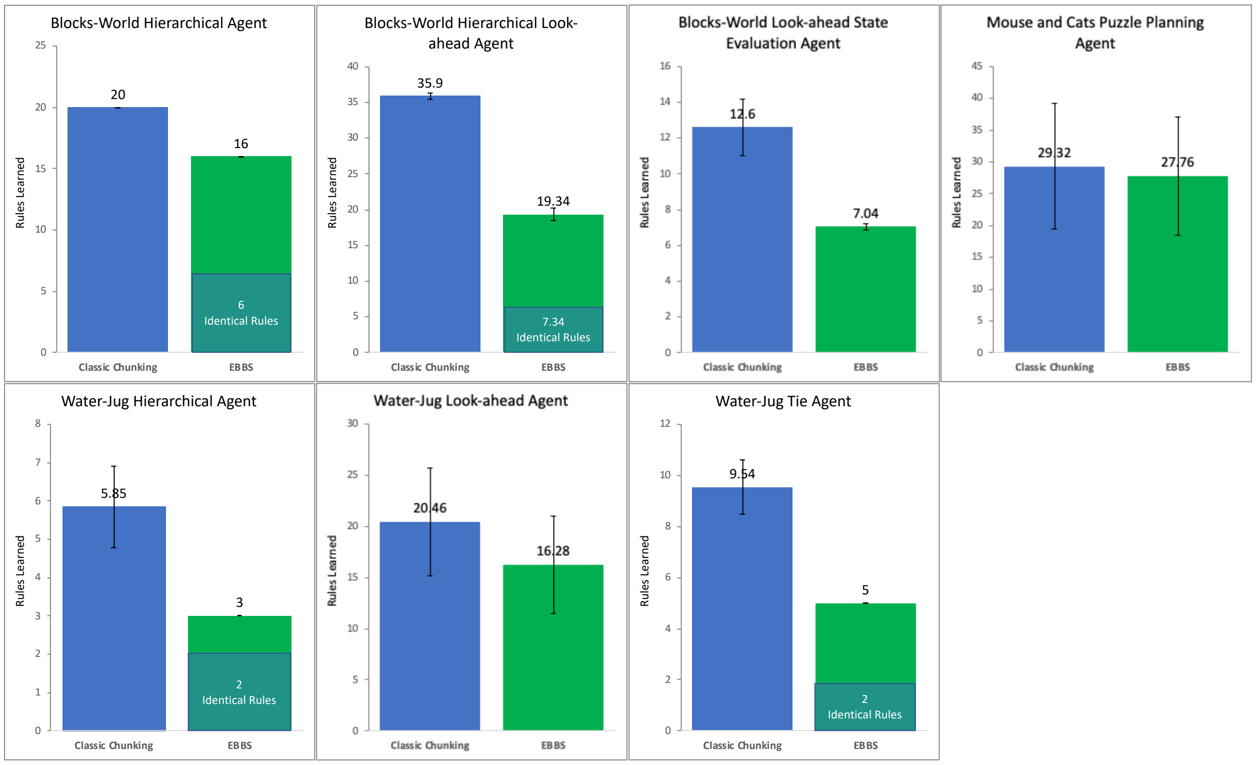


Figure 7.22: The number of rules learned for seven of the agents in group two. EBBS summarizes the reasoning performed by these agents in fewer rules than classical chunking.

In the other three agents, EBBS learns the same number of rules as classical chunking, as shown in Figure 7.23.

[5]While the NLP agent did learn fewer rules using classic chunking, our other metrics will show that this result was not meaningful.

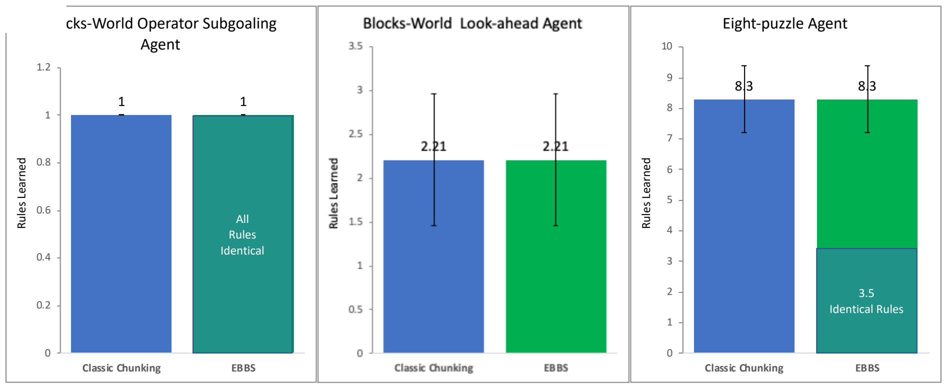


Figure 7.23: The number of rules learned for the final three agents of group two. EBBS learns the same number of rules from the reasoning of these agents.

This again suggests that these agents represent generality in their reasoning in a way that classical chunking can already capture.

7.4.2.3 How Learning Rate Affects the Number of Rules Learned

In this section, we revisit the arithmetic agent and examine how learning rate affects the number of rules learned by the agent. Figure 7.24 shows how many rules are learned from solving 1 to 200 practice problems.

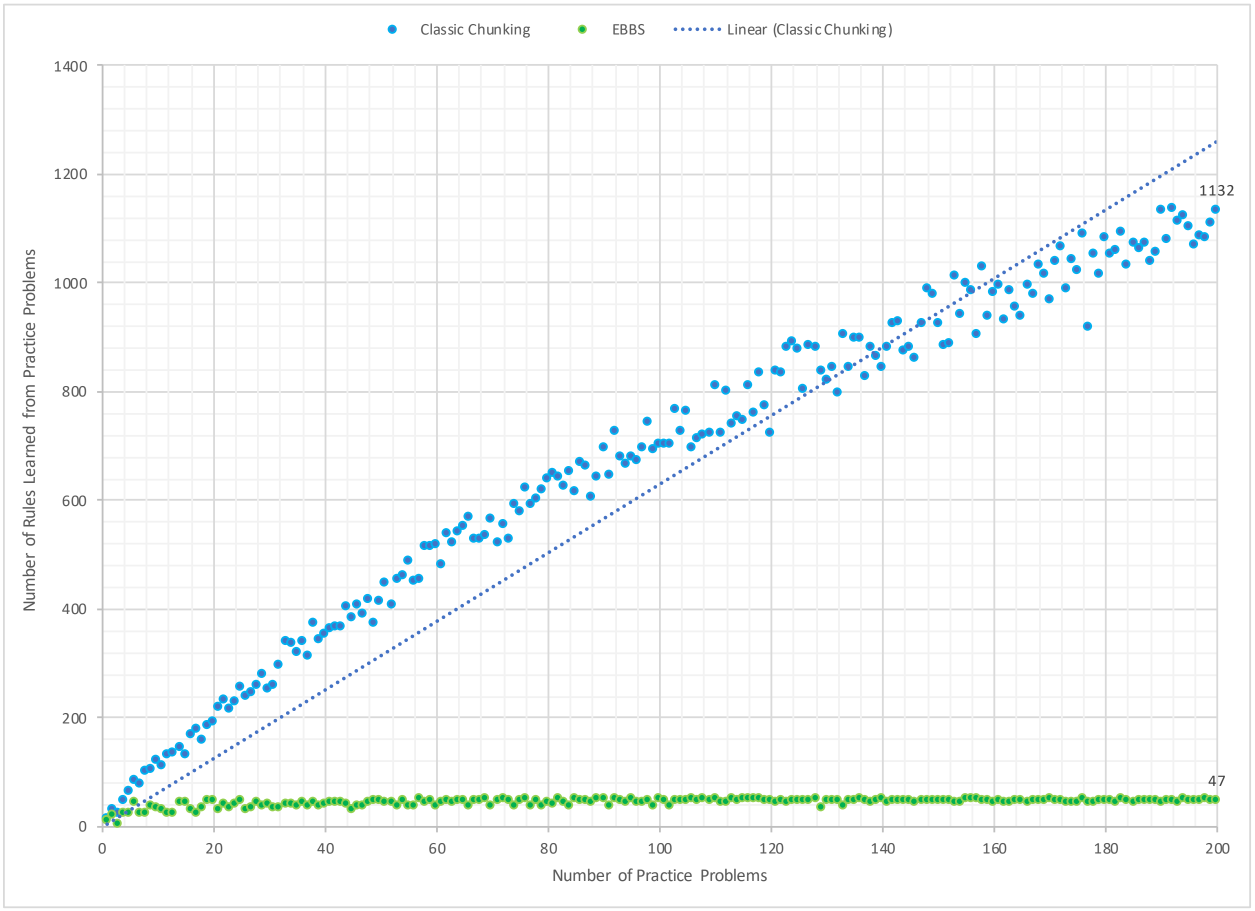


Figure 7.24: The number of rules learned by each learning approach after practicing a 1-200 problems. EBBS quickly learns all rules necessary to represents the reasoning of the arithmetic agent while classic chunking continually learns more rules.

As we can see, the EBBS agent never learns more than 47 rules. (The reason for this, as we show in the Section 7.4.3.1 is that this set of 47 rules eliminates most knowledge impasses, which eliminates the need to learn more rules.) If we look at the raw numbers underlying this figure, we see that EBBS reaches the point where it no longer learns new rules after solving about a dozen practice problems. In contrast, the classic chunking agent continually learns additional rules as it practices more problems. If we extend this experiment, as shown in Figure 7.25, we see that the classical chunking agent is still learning an increasing number of rules after 25,000 practice problems. (It does seem to be approaching an asymptotic value.)

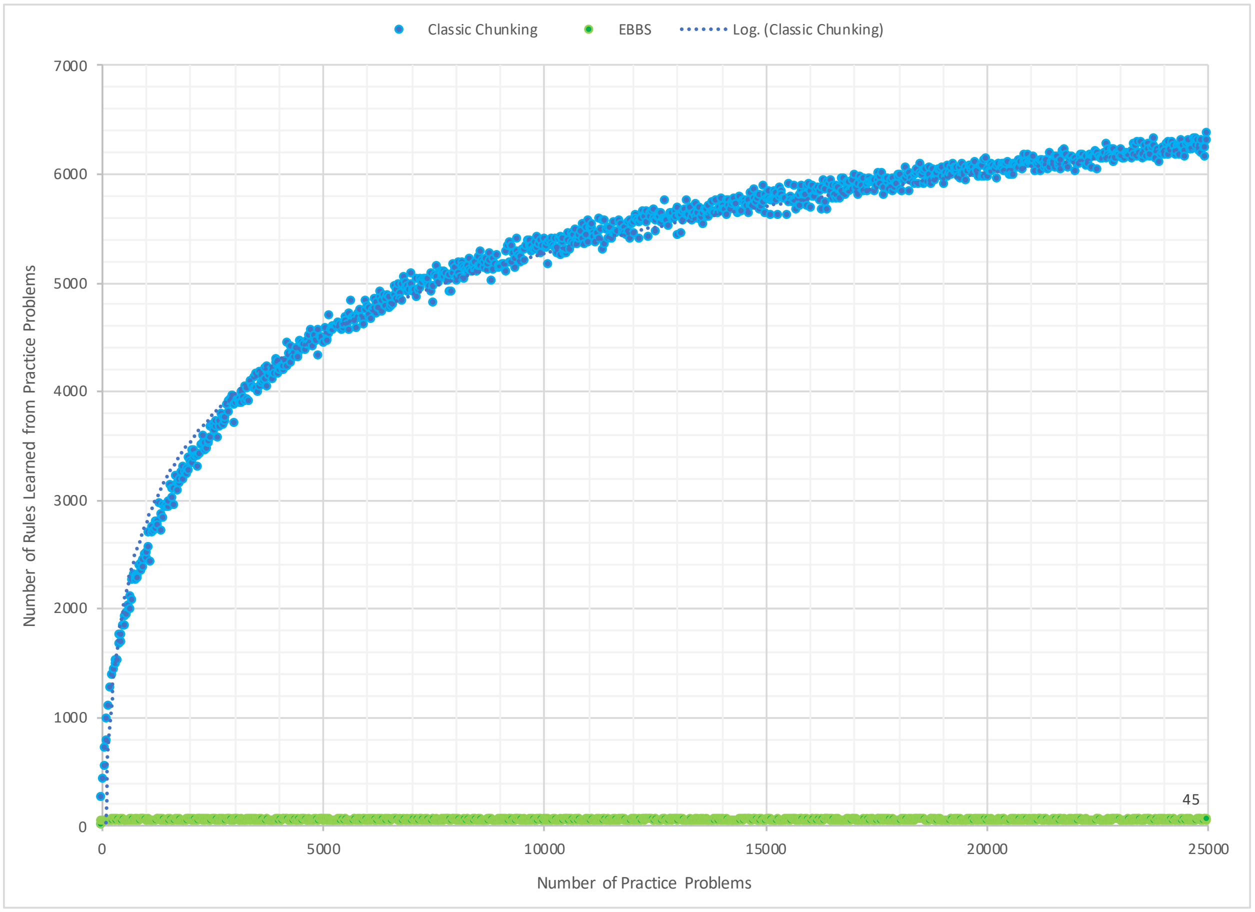


Figure 7.25: The number of rules learned by each learning approach after practicing 1-25k problems. Even after 25k practice problems, classic chunking continues to learn new rules.

7.4.3 Impasses Encountered

The fact that an agent learns fewer rules with a procedural learning algorithm does not necessarily imply that it learned rules that represent the same reasoning at a different level of generality. But if it is the case that the rules learned by EBBS do apply to a larger, more optimally general space of situations, then they should provide benefits in situations that the agent has not encountered more often than those learned by classic chunking. To get some measure of this, the third metric that we use to evaluate whether the rules that are learned by EBBS are closer to optimally general than those learned by chunking is *number of knowledge impasses encountered.* Because Soar's impasse-based learning mechanism is driven by knowledge insufficiencies, a learned rule that captures more of the generality of the subgoal reasoning that resolved a knowledge insufficiency, should apply to a larger space of situations, so it should more often provide the information that the agent needs to avoid a knowledge insufficiency in future similar situations.

7.4.3.1 Evaluation of Agent Group 1

As shown in Figure 7.26, both classical chunking and EBBS are quite effective at reducing impasses encountered in the Arithmetic and A* search agents, but EBBS decreases the number of impasses encountered by a larger amount. And, in the Kenken and NLP agents, EBBS dramatically decreases the number of impasses encountered while classical chunking does not and instead prevents task success.[6]

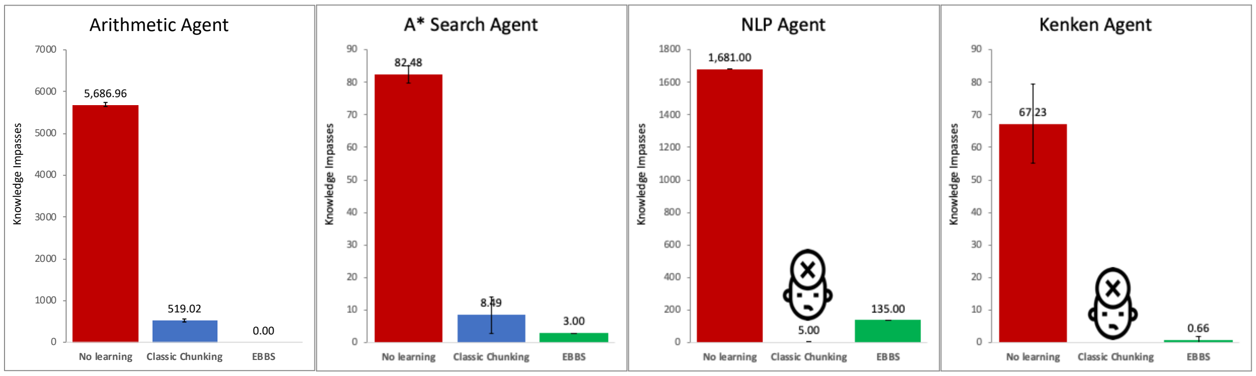


Figure 7.26: The number of impasses encountered by the agents of group one under each learning approach. EBBS prevents more knowledge impasses than classical chunking in all four agents.

In Figure 7.27, we re-visit the arithmetic agent and examine how each algorithm's learning rate affects the number of impasses the agent faces. As we can see, EBBS eliminates most impasses after only a few practice problems, while classical chunking still faces nearly two thousand impasses after 200 practice problems.

[6]While the NLP agent encountered fewer impasses using classic chunking, it did not successfully parse any sentences, so, like the number of learned rules, this is a generality metric that does not help us compare how well classic chunking and EBBS can capture the generality of the NLP agent.

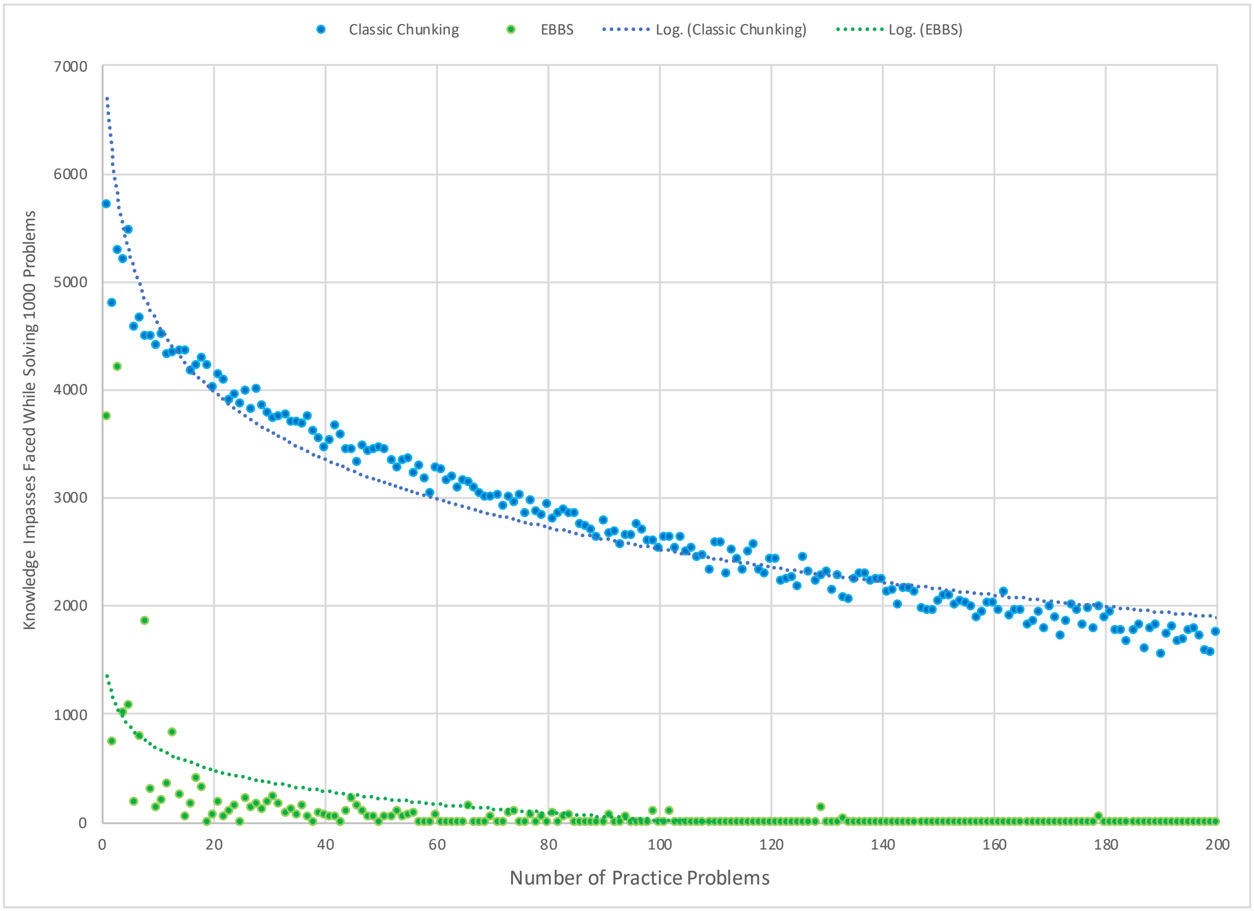


Figure 7.27: The number of impasses encountered by the arithmetic agent after practicing 1-200 problems. EBBS eliminates the vast majority of impasses encountered after only a few practice problems.

If we extend this experiment out to 25,000 practice problems, as shown in Figure 7.28, we see that it takes thousands of practice problems before classical chunking prevents a similar number of impasses as EBBS rules do.

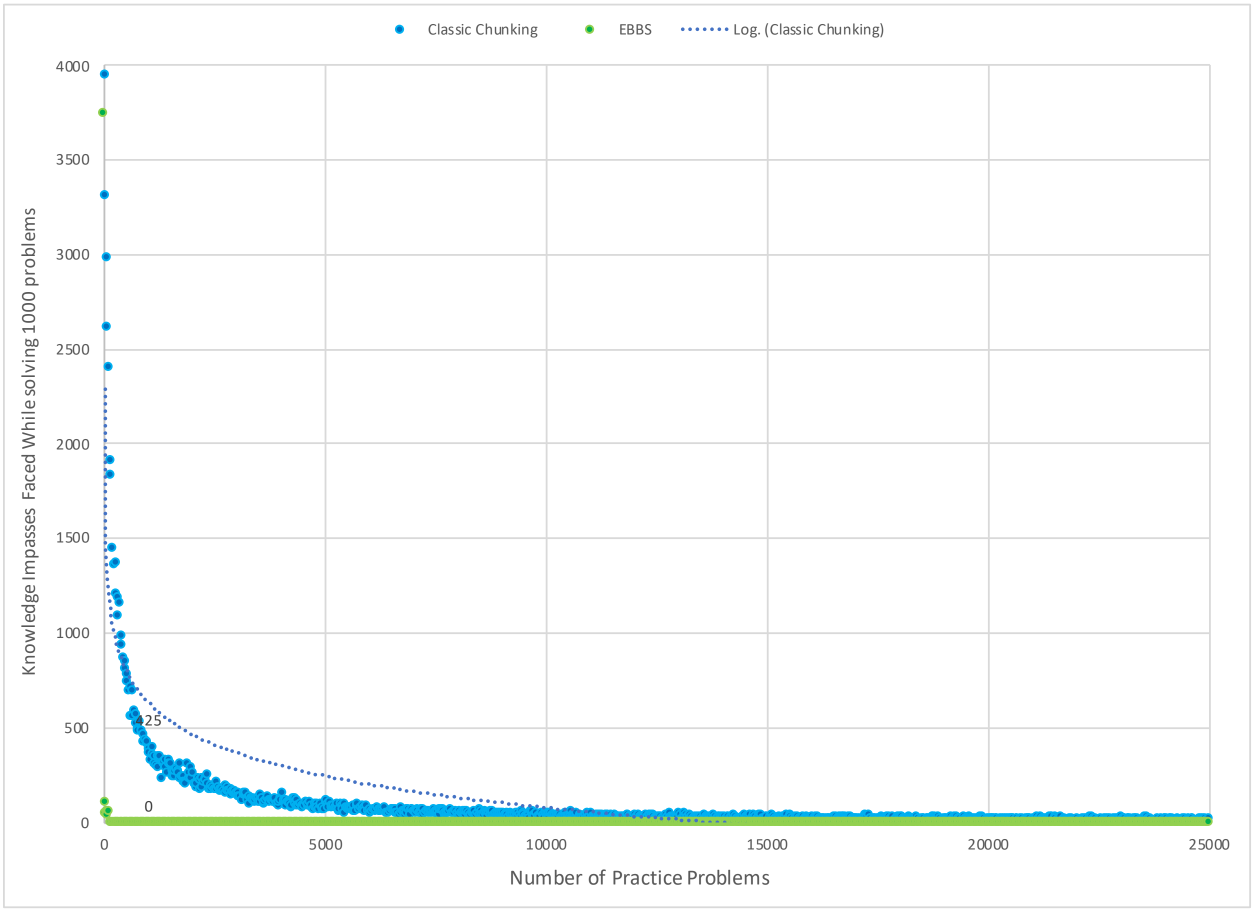


Figure 7.28: The number of impasses encountered by the arithmetic agent after practicing 1-25k problems. Classical chunking requires thousands of practice problems to eliminate as many impasses as EBBS.

7.4.3.2 Evaluation of Agent Group 2

In Figure 7.29, we can see that both EBBS rules and classic chunks provide the agent with the knowledge required to avoid many impasses. Learned rules allow the agent to avoid 76% of impasses in the BlocksWorld look-ahead agent, 20% of the impasses in the Mouse and Cat agent and nearly all of the impasses in the other seven agents. EBBS may be a little better at providing the knowledge necessary to avoid impasses in these particular agents but not by a significant amount. We see that six of the ten agents, the ones in the second and third row of Figure 7.29, face roughly the same number of impasses when using EBBS rules as they do when using classic chunks, which suggests that both approaches learned rules that cover the same space of reasoning that the underlying reason applies to. When the agent needed knowledge, both agents had learned rules ready. In the other four agents, the ones in the top row of Figure 7.29, we see only slight decreases in the number of impasses encountered. If these agents are indeed learning more general rules, they are not preventing significantly more impasses than the more specific rules of classical chunking.

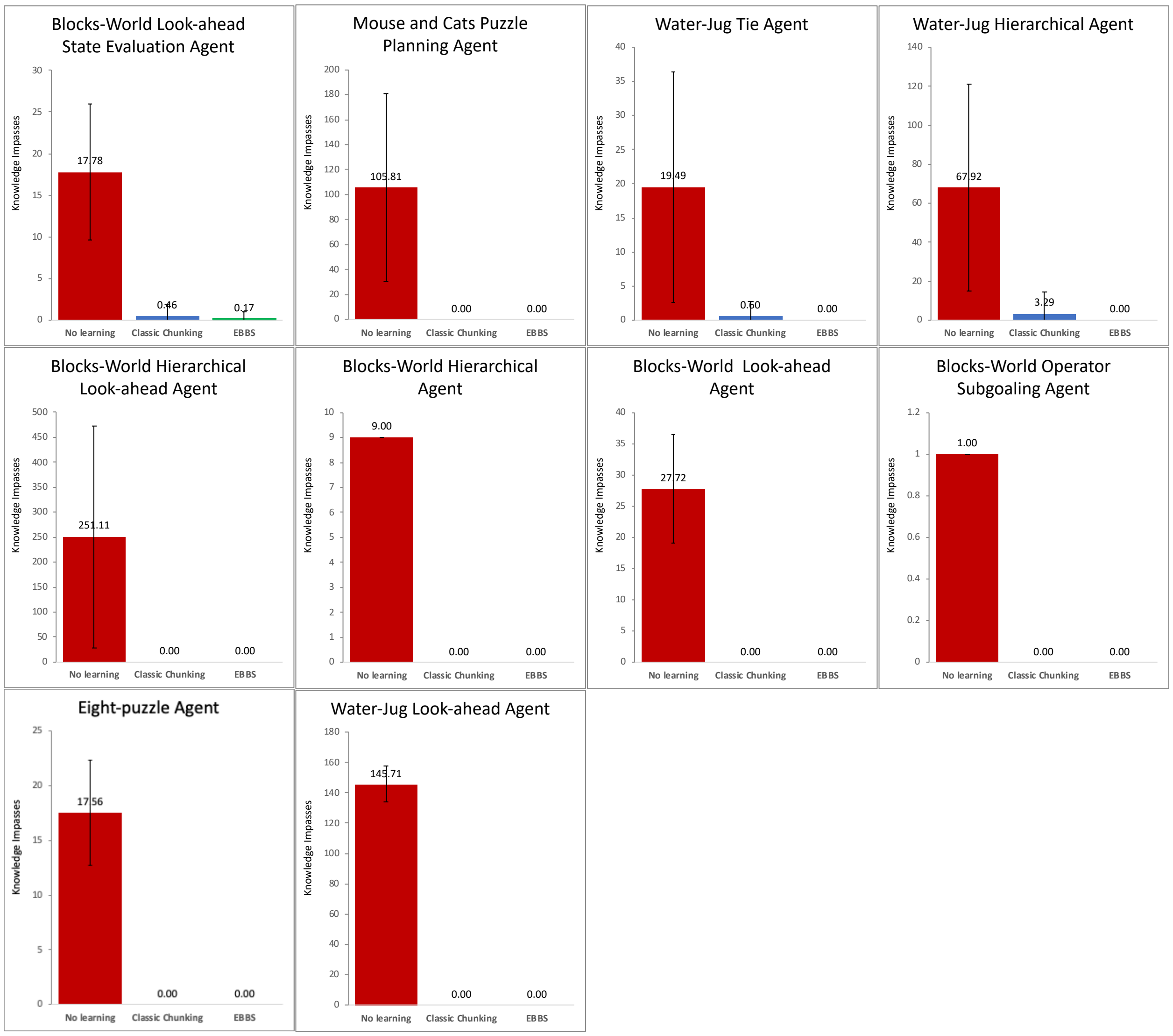


Figure 7.29: The number of impasses encountered by the agents of group two. EBBS and classical chunking both decrease the number of impasses encountered by these agent faces by roughly the same amount.

EBBS's faster learning rate does provides some benefits to these agents. If we examine the run in which the agent is actively learning rules, as shown in Figure 7.30, we find that EBBS reduces impasses in two of the agents of group two in the learning runs. In BlocksWorld Hierarchical, EBBS reduces the number of impasses from 7 to 5, while, in BlocksWorld Hierarchical Look-Ahead, it cuts the number of impasses encountered in half, from approximately 21 to 10.[7] Because both agents avoid all impasses in the post-learning runs, we see that they both learn a set of rules from the initial learning run that replaces all the deliberate subgoal reasoning in subsequent runs. This shows that the EBBS agents learn rules faster; the rules it learned were applied to new situations in

[7]In the other agents, the benefits during the learning run were the same as those during the post-learning runs.

the same run that they are learned in. In contrast, the rules that the classic chunking agents learns, which do cover the same space of situations, find no opportunities to fire until subsequent runs. So, the fact that EBBS learned usable rules more quickly leads to the differences in performances shown in Figure 7.30.

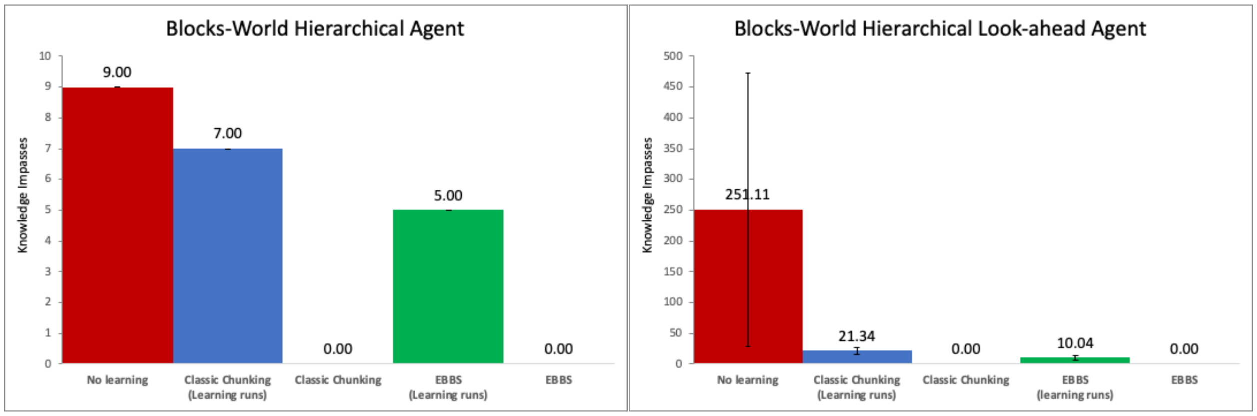


Figure 7.30: The number of impasses encountered in both the learning run and post-learning run. In these two agents, EBBS reduces the number of impasses encountered during the learning run more than classical chunking does.

7.4.4 Firing Count of Learned Rules Per Decision Cycle

One weakness of the impasses-encountered metric is that it is highly dependent on the way the agent decomposes a problem and the strategy it uses. For example, consider an agent that uses search. In such an agent, the number of impasses encountered may be as much a function of how deep the search path tended to be is as it is a function of how often learned rules replace deliberate reasoning. Another weakness of this metric is that it does not reflect when a learned rule fires and creates some of the working memory structures that the agent needs but not enough of them to prevent an impasse. (Sometimes multiple results that previously arose from a subgoal are necessary to avoid an impasse.) In other words, the impasses-encountered metric ignores how much learned rule firings contribute to resolving the *impasses that do still occur*.

To help compensate for these weaknesses, the final metric that we will use to evaluate whether the rules that are learned by EBBS can be applied across a more optimal space of situations than those learned by chunking is the frequency of learned rule firings, i.e. the *firing count of learned rules per decision cycle*. If EBBS is able to learn more optimally general rules than classical chunking, its rules should find more opportunities to fire. The limitation of this metric is that it will not show any relative benefits once both approaches learn rules that cover the same space of situations that the original reasoning could apply to.

7.4.4.1 Evaluation of Agent Group 1

As we can see in Figure 7.31, EBBS chunks find significantly more opportunities to fire than classic chunks in both the arithmetic, A* search and NLP agents.[8] We now see that the fact that the NLP agent learned fewer rules with classic chunking than EBBS is not meaningful. The classic chunks learned by the NLP agent found almost no opportunities to fire.

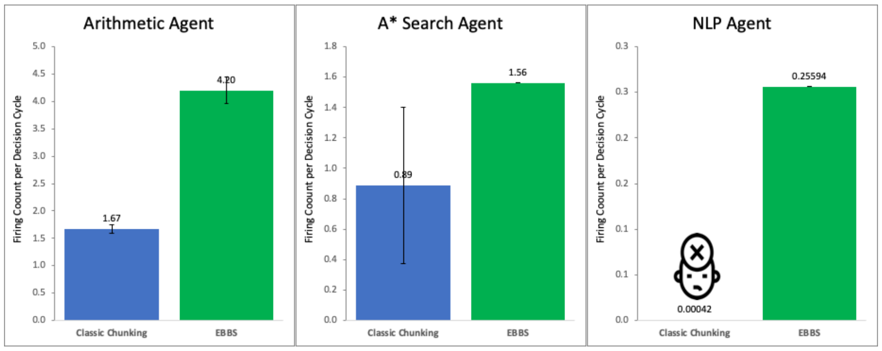


Figure 7.31: The firing count per decision cycle of the agents of group one. EBBS rules find more opportunities to fire every decision cycle than classic chunks in the arithmetic, A* and NLP agents.

If we re-visit the arithmetic agent, we see something interesting when we look at how EBBS's firing rate per decision cycle changes with the number of practice problems, as shown in Figure 7.32. EBBS's firing rate per decision cycle is an even distribution centered on a much larger value than classic chunking's. There seems to be little to no relationship between the effectiveness of EBBS rules and the number of practice problems. This is strong evidence that EBBS learns optimally general rules that summarize the original reasoning in only a few shots. In contrast, the firing rate of classic chunks starts low and gradually increases as the number of practice problems increases. The classic chunking agent must build up quite a large arsenal of rules before firing count increases significantly. And, after about 6000 practice problems, it approaches an asymptotic value that is still significantly lower than the firing rate of EBBS rules.

[8]The Kenken agent is not included in this section because this metric is only useful when comparing two approaches, and the Kenken agent cannot solve the problem using classical chunking.

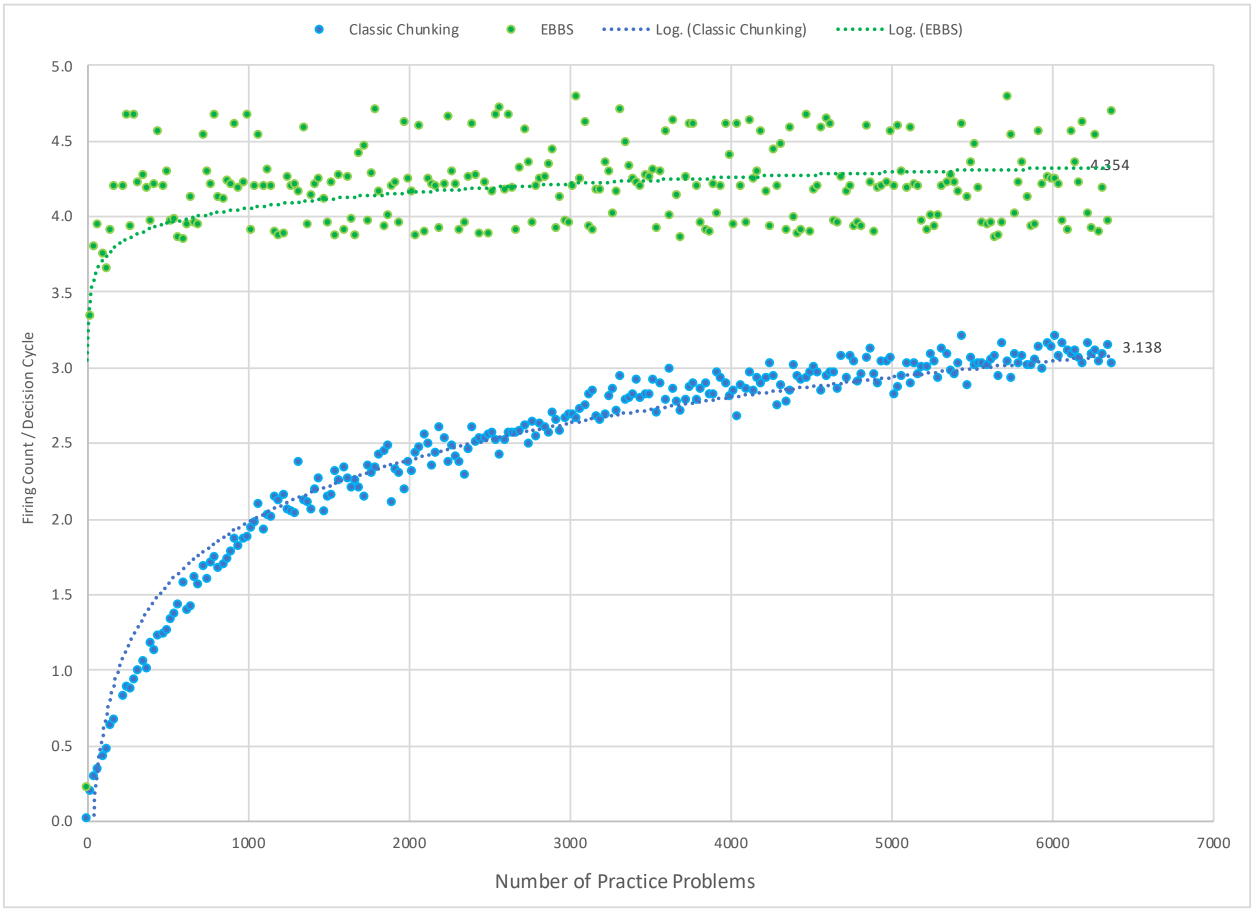


Figure 7.32: The firing count per decision cycle metric shows that the effectiveness of EBBS chunks has little to no relation with the number of practice problems. In contrast, the effectiveness of classic chunks gradually increases with more practice problems.

7.4.4.2 Evaluation of Agent Group 2

In these agents, both approaches quickly learn rules that cover the space of situations that the original rules cover, so we don't expect to see much differences in the firing count rate between EBBS rules and classic chunks. And, as we can see in Figure 7.33, for seven agents, that does seem to be the case.

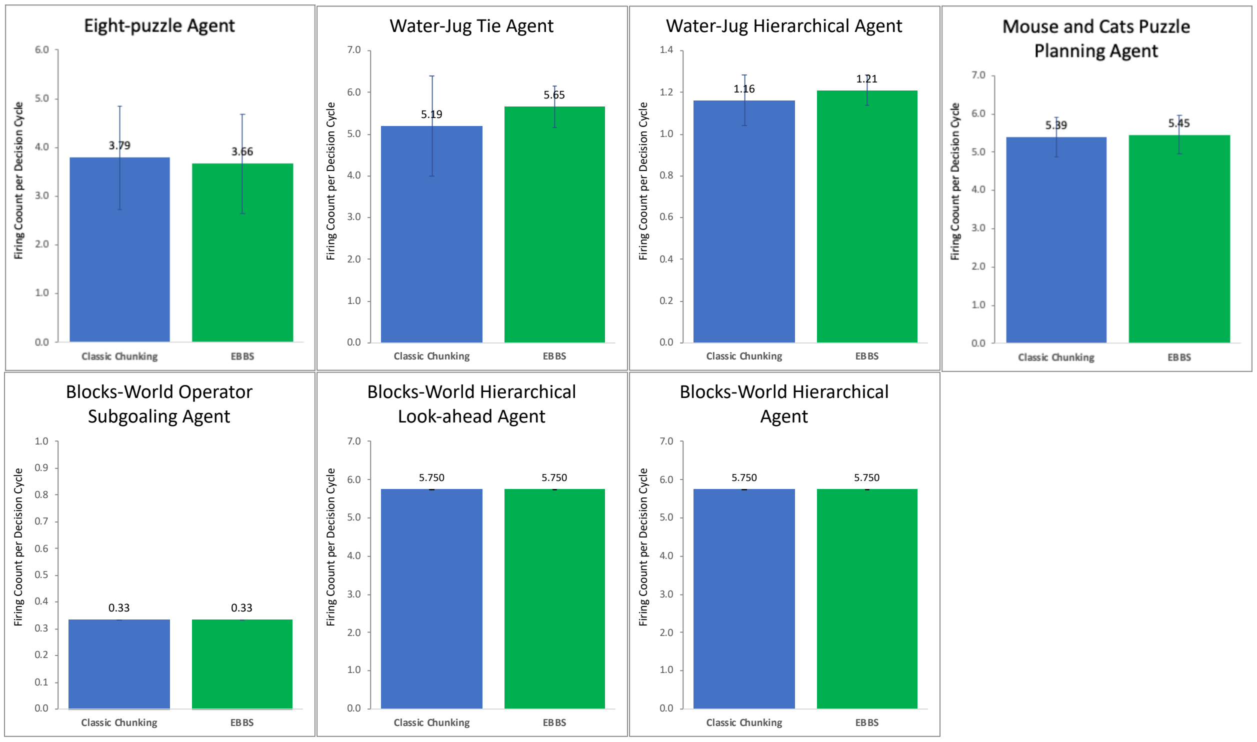


Figure 7.33: The firing count per decision cycle metric shows that, for these 7 agents, EBBS rules do not fire significantly more often than classic chunks.

In the other three agents, though, we find that EBBS rules find more opportunities to fire than their classic chunk counterparts, as shown in Figure 7.34, which suggests that some of their rules were more general and fired more often.

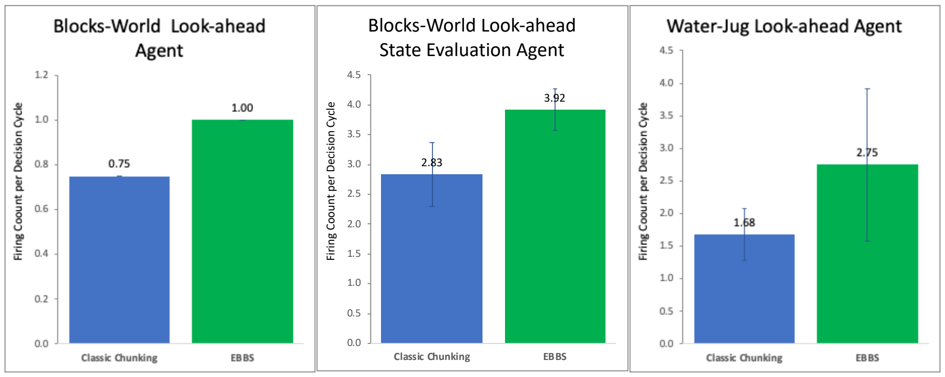


Figure 7.34: The firing count per decision cycle metric shows that, for these three agents, EBBS rules find more opportunities to fire than classic chunks.

7.4.5 Conclusions on our Evaluation of Optimal Generality

Our experiments have shown strong evidence that EBBS captures more of the generality of the underlying rules. In many agents, EBBS summarizes an agent's reasoning in fewer rules than classical chunking, and those rules prevent more impasses and find more opportunities to fire than the rules of classical chunking. We have also shown evidence that EBBS rules do indeed reason generally over knowledge types that classic chunks instead test specific values of, which is the key reason that classic chunking learns overly specialized rules.

7.5 Evaluating the Computational Cost of EBBS Approach (E3)

In this section, we attempt to evaluate whether the computational overhead added by the EBBS approach prevents an agent from maintaining reactivity.

Note that we do expect EBBS to have higher computational costs. Obviously, it performs far more analysis of the agent's subgoal's reasoning, and that is not free. Moreover, the more general rule is, the more expensive it can be to match against (Tambe and Newell, 1988), an issue that is examined more closely in Section 7.5.1.

As previously mentioned in Section 2.5, we ignore the average growth issue (Doorenbos, 1995), i.e. the performance decrease that occurs as procedural memory grows. Not only is it not currently a problem — modern rule matchers like RETE can efficiently detect matches in procedural memories with millions of rules — but EBBS learns fewer rules than the current procedural learning system, so this issue should be ameliorated not exacerbated.

7.5.0.1 Experiment Setup

In all of our previous experiments, we allow an agent to solve *only a single problem* with learning on[9] before we run the agent with those learned rules repeatedly with learning off, as described in Section 7.1.1. We focused on the post-learning runs because we wanted to control for the confounding factors that arise from problem-solving so that we can better examine the effects of the learned rules. Even if we were not focused on that, the comparison between the non-learning run and the post-learning run may *still* be more important than the learning runs. The reasoning behind this argument is that an agent, if given enough learning runs, is able to learn a set of rules that cover the entire space of reasoning that the original rules cover. When the agent reaches that point, it's seldom needs to analyze subgoals, which means its processing will be similar to that of the post-learning agent. So, as the agent gets closer to learning the full space of rules that summarize the agent's deliberate reasoning, its learning run performance becomes more and

[9]Or, in the case of the Arithmetic agent, we let it solve 1000 math problems.

more like its post-learning performance. Because the speed at which an agent can learn the full space of reasoning is a function of the task/domain, the post-learning run is a much more reliable point of comparison. *Nonetheless, if learning costs when actively learning prevents an agent from maintaining reactivity, that is a problem. So, we must examine learning runs as well.*

In the experiments of this section, each agent will perform both a learning run where it solves a single problem, which we will refer to as the initial learning run, and a learning run where it solves 50 problems, which we will refer to as the learning run.[10] Note that in these experiments, the agent's post-learning runs use the rules learned from the agent that solves 50 problems with learning on, not the rules learned from the agent that solves a single problem.

- **100 trials**
  1. **Non-Learning runs**
     (a) Baseline runs (Agent solves 50 problems with all procedural learning disabled.)
  2. **Classical chunking runs**
     (a) Initial learning run (Agent solves 1 problem with classical chunking enabled.)
     (b) Learning runs (Agent solves 50 problems with classical chunking enabled.)
     (c) Post-learning runs: (Agent solves 50 problems with learning disabled. Agent uses classic chunks learned in learning runs.)
  3. **EBBS runs**
     (a) Initial learning run (Agent solves 1 problem with EBBS enabled.)
     (b) Learning runs (Agent solves 50 problems with EBBS enabled.)
     (c) Post-learning runs: (Agent solves 50 problems with learning disabled. Agent uses EBBS rules learned in learning runs.)

7.5.1 Computational Costs of Determining Which Rules Match

The utility problem (Mooney, 1989) is a term that describes a phenomenon where learned rules decrease performance because the persistent cost of determining whether rules match outweighs the periodical computational benefits that are gained when the rules fire. While exploring the general utility problem is outside the scope of this work, we perform a limited examination of how it affects our agents by showing cases where *expensive chunks* arise and how they influence the time it takes for our agents to solve a problem.

[10]For practical reasons, the agent performs 50 learning runs instead of solving 50 problems in one run. Similarly, the Arithmetic agent solves 50 sets of 1000 problems in this experiment.

We now examine the amount of time spent matching rules per decision cycle. To determine whether expensive chunks arise more often with EBBS than with classic chunking, Figure 7.35 compares how much time is spent matching rules each decision cycle for both EBBS and classic chunking. This figure also includes the percentage increase in match cost that occurs when that agent uses EBBS rules instead of classic chunks, which is shown as a percentage number in between and to the right of each pair of bars. (Negative values indicate that an agent spends less time matching with EBBS than it does with classic chunks.) Any values in red indicate agents where the lower bound of the match cost of EBBS rules is always higher than the upper bound of the match cost of classic chunks. Note that the metrics in this figure are generated using the post-learning runs, where the agent's learning rate does not play a role

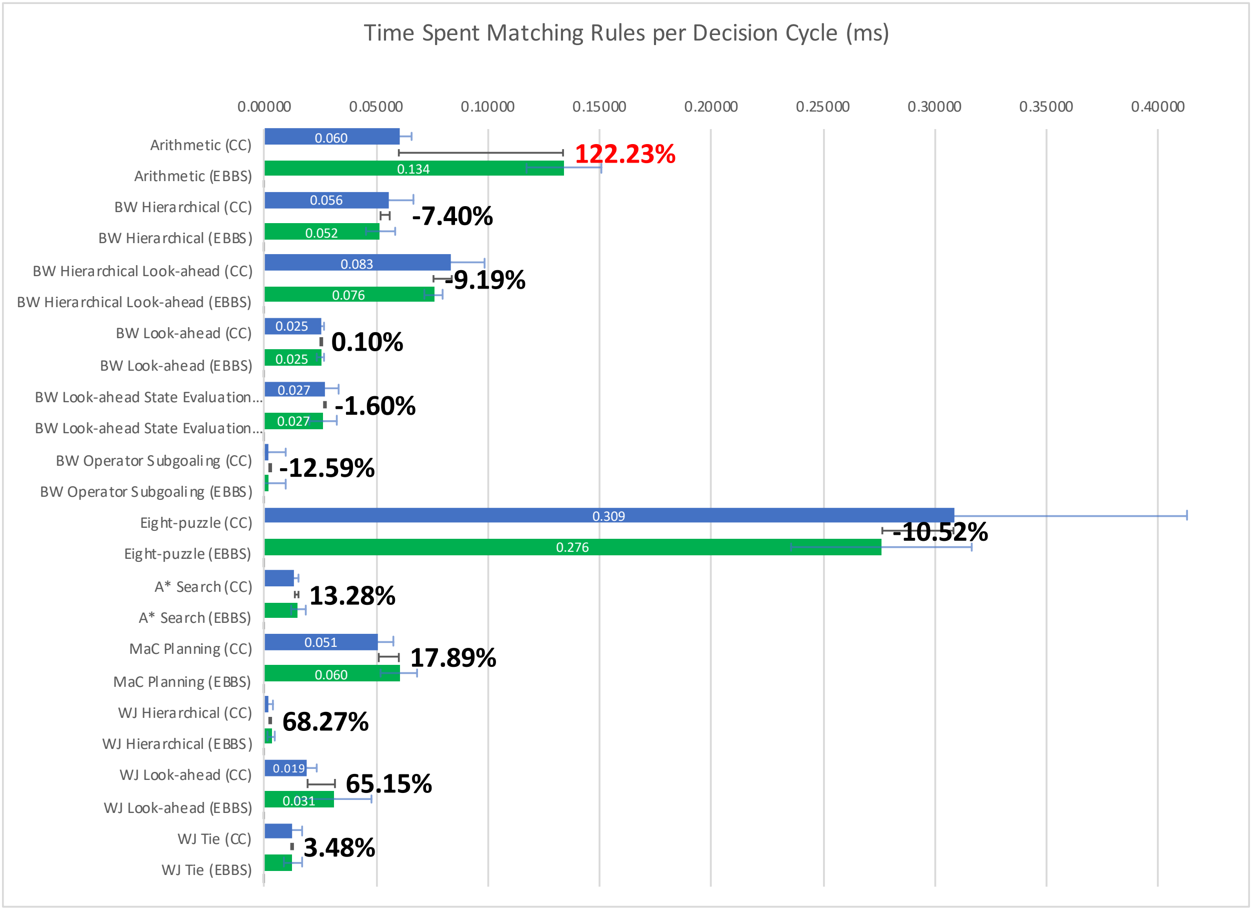


Figure 7.35: The amount of time spent matching rules each decision cycle and the percentage increase in match cost that occurs when that agent uses EBBS rules instead of classic chunks.

The agent with the highest percentage increase in match cost is the Arithmetic agent, where EBBS rules increase the amount of time spend matching rules each decision cycle by 122%. This is also the only agent where the lower bound of the match cost of EBBS rules is always higher, by a significant margin, than the upper bound of the match cost of classic chunks. This is clearly an agent whose match costs need closer examination.

For the rest of the agents in our evaluation, though, matching against EBBS rules does not seem to be systematically more computationally expensive than matching against classic chunks. The variability in match cost, as depicted by the error bars in Figure 7.35, is greater than differences in values in all of them,[11] which suggests that the differences in average time may not indicate anything meaningful. Moreover, the direction of improvement is split. In 5 of the agents, matching against EBBS rules is faster, and in 6 of them, matching against classic chunks is faster.

It is interesting that EBBS has lower match costs than classic chunking in some of these agents, even if the difference is small. While exploring the utility problem and the underlying reason for match differences is outside of the scope of this project, this does show that learning more general rules does not always lead to higher match costs, which is not unexpected. While EBBS rules are able to encode generality that classic chunks cannot, which may open up additional opportunities for match cost issues, that fact alone does not make them more expensive. Previous research has shown that multiple factors play a role in match cost, for example, the number of WMEs in memory that can match the variables in the learned rules and how many of them are multi-valued (Tambe and Newell, 1988). Those other aspects play a role in the match cost of both EBBS rules and classic chunks.

[11] The error bars overlap.

We now examine how the increased match cost of the arithmetic agent affects the total time of execution, as shown in Figure 7.36. Please remember that the classic chunking agent is given 50 learning runs, which translates into 50,000 practice problems. Consequently, it learns the full space of rules and solves its problems, in its post-learning runs, in the same number of decision cycles as the agent that learns EBBS rules.

As we can see, EBBS introduces computational costs that are significant enough that the average duration of the EBBS runs is greater than that of the classic chunking runs. It is worth noting that the agent still solved the problem faster than the non-learning agent.

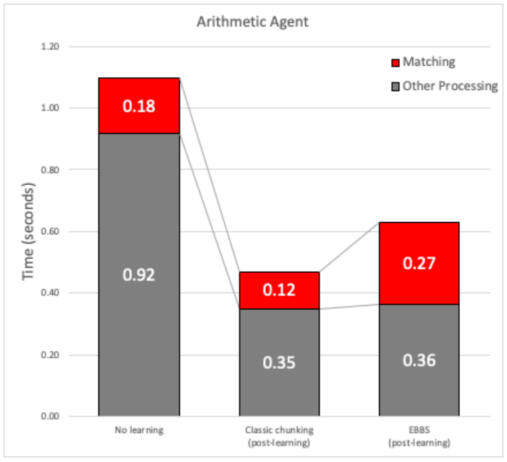


Figure 7.36: Execution time and the number of decision cycles to solve a task for the Arithmetic agent. The additional computational costs of EBBS outweigh the amount saved by decreases in decision times.

In conclusion, while the match cost of EBBS rules do generally seem higher than the match cost of classic chunks, we do not see any evidence that expensive chunks are more pervasive when learning EBBS rules. The Arithmetic agent is the sole agent that shows high match costs that may be caused by expensive chunks. And, if we use a strict definition of expensive chunks that only includes agents where the persistent cost of determining whether rules match outweighs the periodical computational benefits that are gained when the rules fire, the EBBS rules learned by the arithmetic agent are technically not expensive chunks either. As shown in Figure 7.36, they still provide a computational benefit over the non-learning agent.

7.5.1.1 Impact of Increased Match Costs in Real-World Agents

Real-world agents are often connected to external environments or embodiments, which means they must spend time updating their percepts, potentially expensive processing that is performed periodically, often every decision cycle. As a result, match cost may be a relatively smaller portion of the overall decision cycle cost than we see in our evaluation agents. This would also mean that *reducing the number of decision cycles to solve a problem may lead to bigger improvements in the total time to execute the task than we are seeing in this evaluation*. Because updating input is generally a constant cost each cycle, any decrease in the number of decision cycles needed to solve a problem also avoids the cost of that many percept updates. So, we expect that the benefits EBBS provides to real-world agents, which may have embodiments or environments where that cost is high, to be much more consequential, in terms of the total time to solve a task, than what we would see in these test agents.

In our experiments, we found that percept processing costs are influential enough that even a small amount will dominate the effect that match cost has. To illustrate this, we show how our previous timing figures for the Arithmetic agent, the agent most affected by high match cost, would be impacted by the inclusion of percept updating. Figure 7.37 shows how much time that the Arithmetic agent took to solve 1000 problems given a set of practice problems without any adjustment for percept updating. We can see that the additional computational costs of EBBS is larger than the speed-up provided by the decrease in the number of decision cycles used, regardless of the number of practice problems.

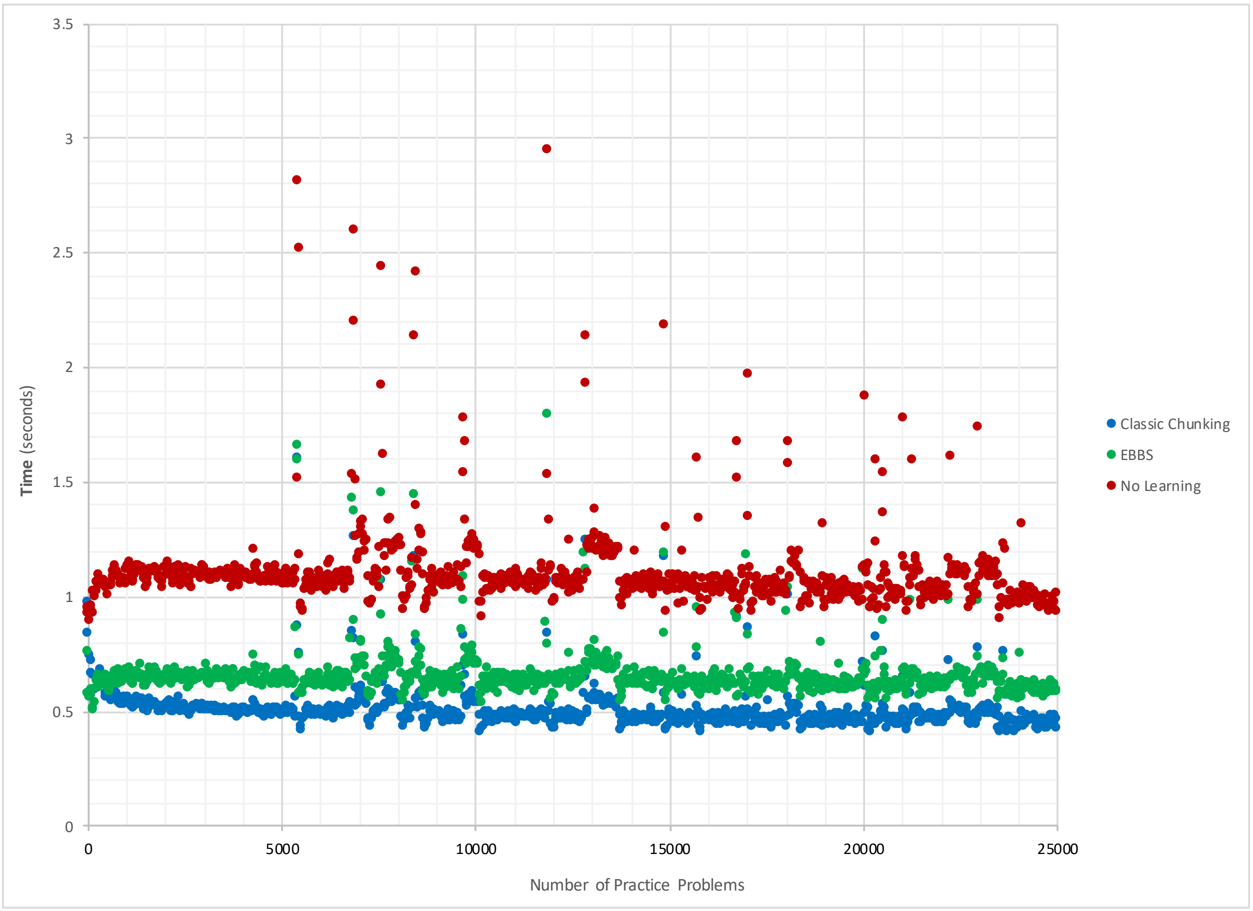


Figure 7.37: The amount of time spent to solve 1000 math problems after practicing 1-25k practice problems.

In comparison, Figure 7.38 shows the same data *but also includes of 0.001 ms of percept updating time every decision cycle*. As we can see, adding some percept updating costs decreases the total time to solve the problem in a way that it once again correlates closely to the decreases in decision cycles to solve a problem, as previously shown in Figure 7.14.

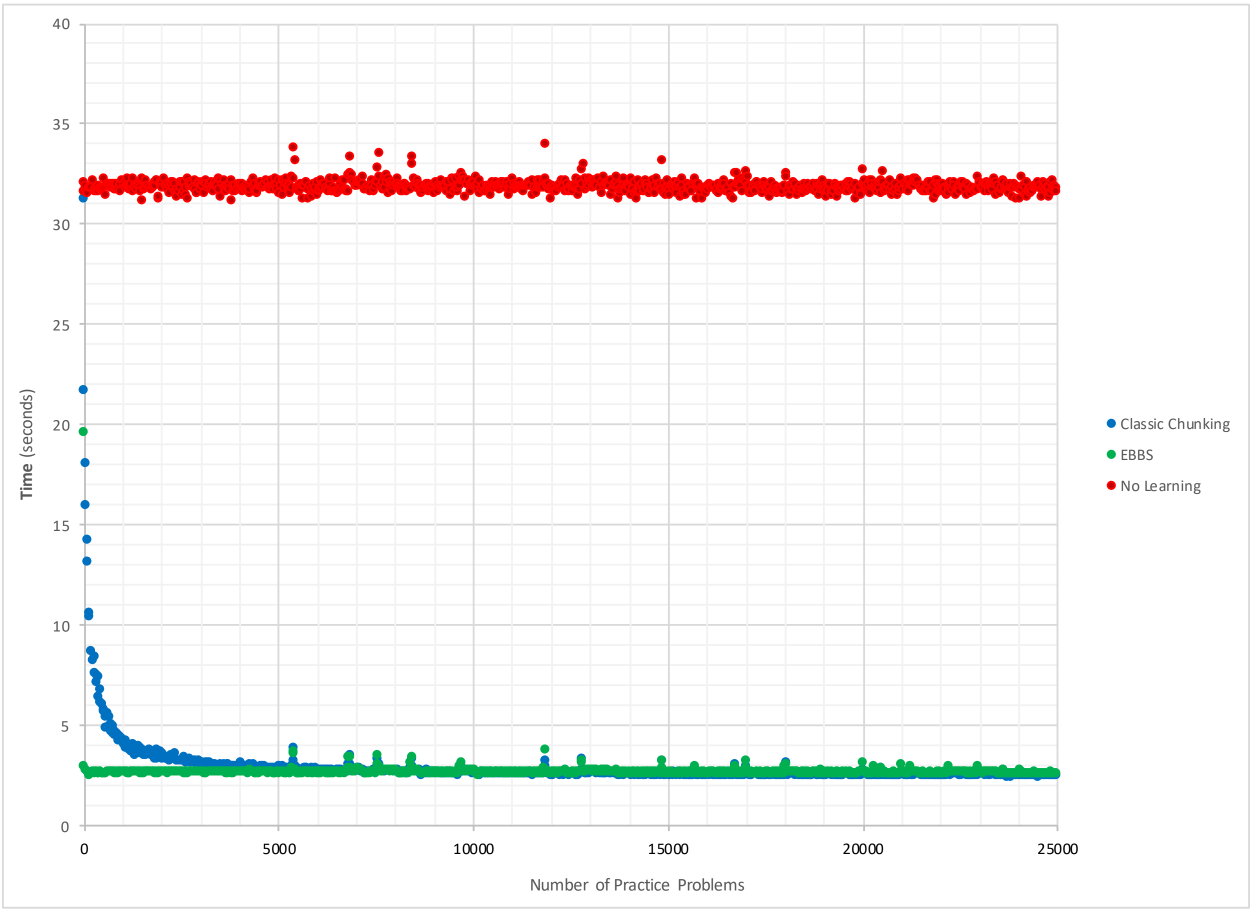


Figure 7.38: With the inclusion of minimal percept processing time (0.001 ms/dc), the relative amount of time needed for the arithmetic agent to solve 1000 problems under each learning approach changes dramatically. It now mirrors how the number of decision cycles to solve the problem decreases with number of practice problems.

Obviously, if you add enough time to a decision cycle for perceptual processing, you can mask the effects of other contributing factors. So, it's important that we point out that we chose a conservative value for percept updating, 0.001 ms. Even toy domains have much higher values. For example, in the Eaters domain, a Pac-Man like game environment used in the Soar tutorial, percept updating takes approximately 0.12 ms every decision cycle. And, in embodied real-world agents, percepts with vision often take on the order of 10-100 ms or more.

7.5.2 Computational Costs of Analyzing Subgoals and Building Rules

In Section 5.7, we discussed the processing that occurs after a rule matches in a subgoal and returns a result to a supergoal, processing that we referred to as the learning episode. In this section, we examine the cost of this processing, which we will refer to as learning costs.

One interesting phenomenon we noticed is that a significant portion of the speed-up benefits that online procedural learning provides to these agents *is a result of the agents spending less time learning*. While it may be counter-intuitive that enabling procedural learning would produce runs with lower learning costs, that is often the case. This is because an architecture that is based on universal subgoaling is never able to fully disable learning; even a non-learning agent needs to perform similar analysis of subgoals to build justifications. And, as we previously saw, agents that are learning more specific rules face more subgoals, and agents that disable procedural learning face the most subgoals of all, every one of which needs to be analyzed. So, even though a non-learning or classic chunking agent may be doing less work when analyzing a single subgoal, it may be analyzing more of them.

In the next section, we present figures that show the amount of time spent in the following categories of Soar's internal processing:

- **Learning**: Architectural processing that creates rules or justifications by analyzing how reasoning in a subgoal relates to knowledge in the supergoal. This metric is the focus of this discussion and is highlighted in red in all of the figures.
- **Matching**: Processing that determines which rules in procedural memory match the state of working memory.
- **WM updating**: Processing that annotates each WME with meta-information that indicates which goals have access to it. Note that WME updating is a function of the number of working memory changes that occur and is not directly impacted by learning.
- **Other processing**: This metric is calculated by taking the total time of execution and subtracting the above three timers. Because this a bucket for all miscellaneous processing, it is not particularly meaningful, but we include it in our figures so that the bar shows the total time of execution.

Note some of the agents have learning and post-learning runs that are so much faster than the non-learning run that they are difficult to visually depicted clearly; consequently, we provide a support figure that zooms in on the bars that are too small to read.

7.5.2.1 Results

Our experiments show evidence that the following three observations hold consistently:

1. **(O1)**: Learning costs of the *non-learning run* are always higher than that of the *post-learning run*.

2. **(O2)**: When either classical chunking or EBBS is enabled, learning costs decrease with more learning. In other words, the learning costs of the *initial learning run* are higher than the learning costs of the *full learning runs*, which are higher than learning costs of the post-learning runs. This is evidence for our argument that, as the agent gets closer to learning the full space of rules that summarize the agent's deliberate reasoning, the performance of its learning runs approach the performance of its post-learning runs.

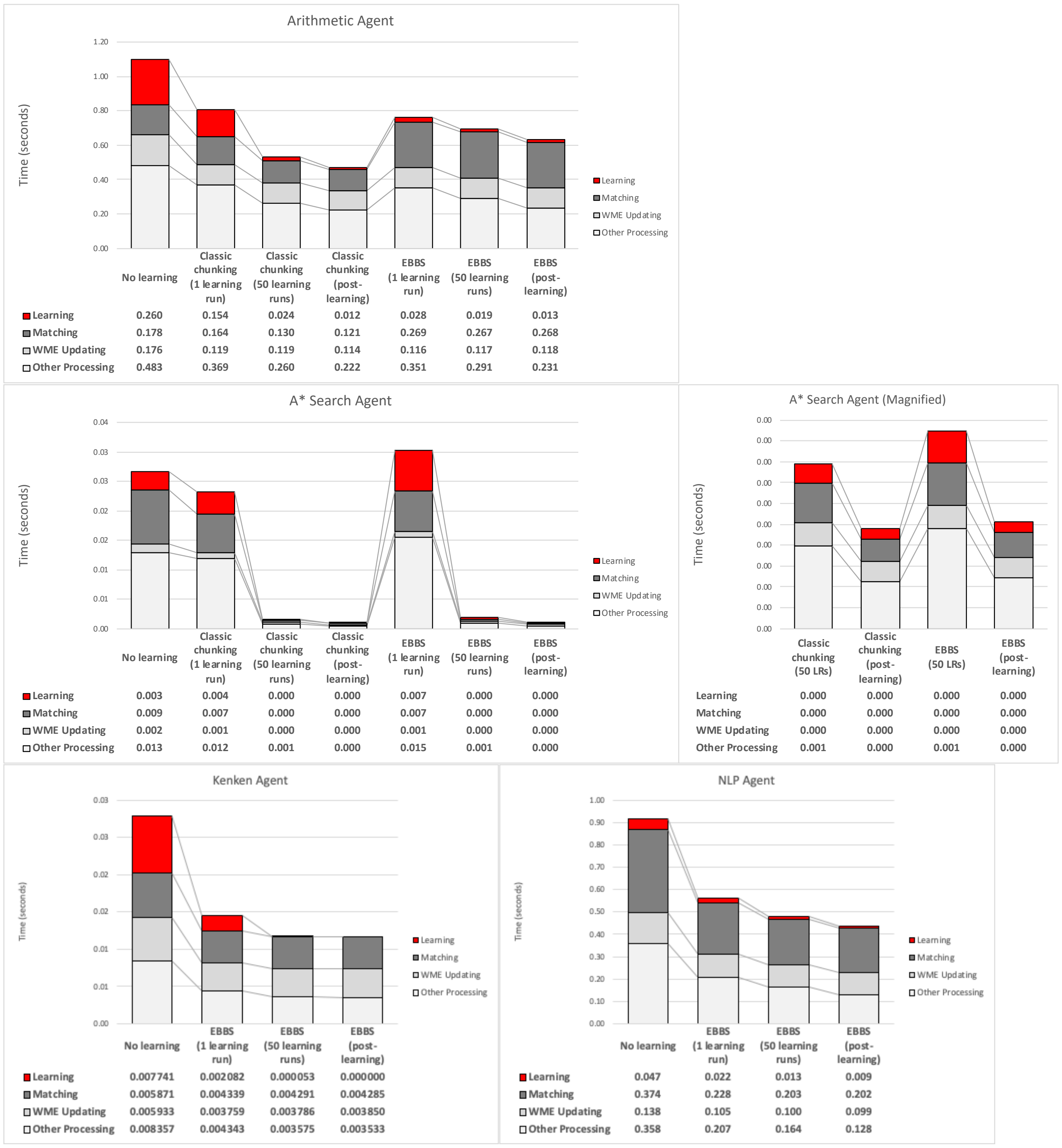
3. **(O3)**: Learning costs of both EBBS and classic chunking in the post-learning runs are comparable to each other.

They also show that the following two observations are sometimes true:

1. **(O4)**: Learning costs of the initial learning run are sometimes greater than that of the non-learning run. In half of the agents, it takes more than a single learning run before the benefits of learned rules overcome the cost of learning them.

2. **(O5)**: Learning costs of the initial EBBS learning run are higher than that of the initial classic chunking learning run in eight out of fourteen evaluation agents. While the differences are fairly minor, this confirms that it can be more computationally expensive to learn EBBS rules than it is to learn classic chunks.

Figure 7.39 shows how the agents of group one spend their time. As we can see, O1 holds true across all agents: the learning cost of the non-learning run, the red portion of the left-most bar, is higher than the learning costs of the post-learning runs, the red portion of the middle and right-most bars. (For the Arithmetic, Kenken and NLP agents, it is higher than all other runs.) We also see evidence of O2, that learning costs decrease with more learning, in all agents. Both the EBBS and classic chunking spend less time building rules in their full learning run than they do in their initial learning run. While O3 is not applicable to the Kenken or NLP agents because they cannot be run with classic chunking, we do find that O3 holds true for both the Arithmetic agent and the A* agent: both EBBS and classic chunking have almost the same learning cost in the post-learning runs. We find O4 is true only for the A* Search agent, in which the learning costs of the initial learning run of both EBBS and classic chunking are higher than the learning costs of the non-learning run. Finally, O5 also only holds true for the A* search agent, where the initial EBBS learning run is more expensive than the initial classic chunking learning run. (In the arithmetic agent, though, the classic chunking arithmetic agent has higher learning costs during its initial learning run than the EBBS agent. The reason for this, as we learned in Section 7.4.3.1, is that the arithmetic agent that learns classic chunks always faces thousands of impasses that the agent that learns EBBS rules avoids.)



Arithmetic Agent

	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.260	0.154	0.024	0.012	0.028	0.019	0.013
▨ Matching	0.178	0.164	0.130	0.121	0.269	0.267	0.268
□ WME Updating	0.176	0.119	0.119	0.114	0.116	0.117	0.118
□ Other Processing	0.483	0.369	0.260	0.222	0.351	0.291	0.231

A* Search Agent

	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.003	0.004	0.000	0.000	0.007	0.000	0.000
▨ Matching	0.009	0.007	0.000	0.000	0.007	0.000	0.000
□ WME Updating	0.002	0.001	0.000	0.000	0.001	0.000	0.000
□ Other Processing	0.013	0.012	0.001	0.000	0.015	0.001	0.000

A* Search Agent (Magnified)

	Classic chunking (50 LRs)	Classic chunking (post-learning)	EBBS (50 LRs)	EBBS (post-learning)
Learning	0.000	0.000	0.000	0.000
Matching	0.000	0.000	0.000	0.000
WME Updating	0.000	0.000	0.000	0.000
Other Processing	0.001	0.000	0.001	0.000

Kenken Agent

	No learning	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.007741	0.002082	0.000053	0.000000
▨ Matching	0.005871	0.004339	0.004291	0.004285
□ WME Updating	0.005933	0.003759	0.003786	0.003850
□ Other Processing	0.008357	0.004343	0.003575	0.003533

NLP Agent

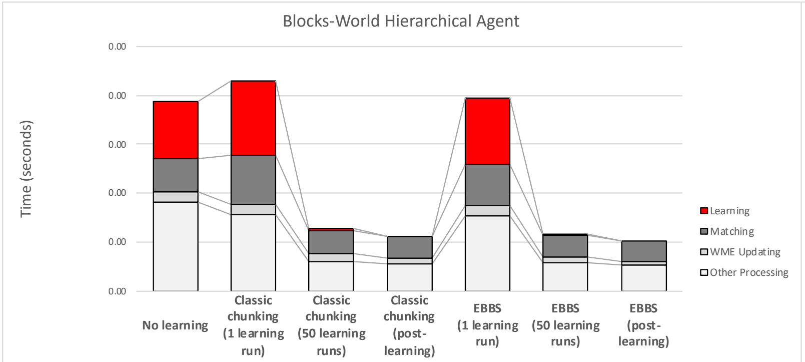
	No learning	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.047	0.022	0.013	0.009
▨ Matching	0.374	0.228	0.203	0.202
□ WME Updating	0.138	0.105	0.100	0.099
□ Other Processing	0.358	0.207	0.164	0.128

Figure 7.39: A breakdown of how the agents of group one spend their time.

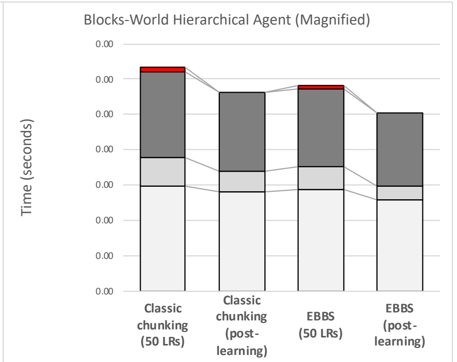
Figures 7.40 and 7.41 show how the BlocksWorld agents spend their time. Again, O1, O2 and O3 hold for all agents.[12] Learning costs of the non-learning run are always higher than those of the post-learning runs; when learning is enabled, learning costs decrease with more learning; and, the learning costs of the EBBS and classic chunking post-learning runs are comparable. O4, which

[12]Note that BlocksWorld Operator Subgoaling agent is such a simple agent that our timers do not have the resolution to measure how long learning takes in several of the runs.

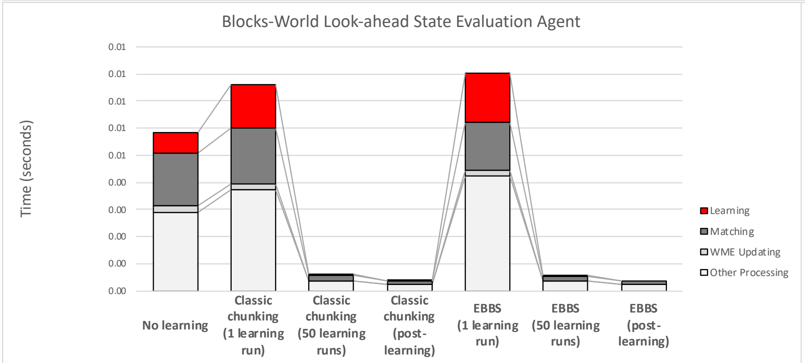
states that learning cost of the initial learning run may be greater than that of the non-learning run, is true for 4 out of the 5 agents. The one exception is the BlocksWorld Hierarchical Look-ahead agent, which is able to eliminate all subgoaling in its initial learning run. Finally, O5 holds for 3 out of the 5 agents: in the BlocksWorld Look-Ahead State Evaluation, BlocksWorld Look-Ahead and the BlocksWorld Operator Subgoaling agents, the learning cost of the initial EBBS learning run is greater than that of classical chunking.



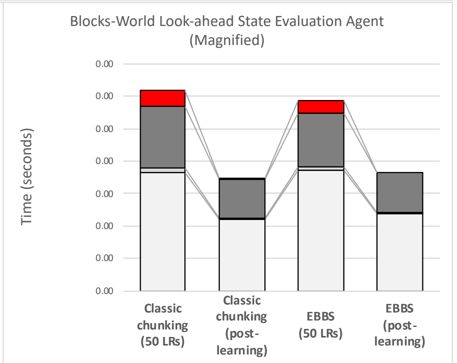
	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.000584	0.000763	0.000013	0.000000	0.000675	0.000011	0.000000
■ Matching	0.000340	0.000498	0.000245	0.000224	0.000432	0.000220	0.000207
□ WME Updating	0.000101	0.000101	0.000081	0.000059	0.000100	0.000066	0.000039
□ Other Processing	0.000914	0.000785	0.000295	0.000280	0.000768	0.000286	0.000259



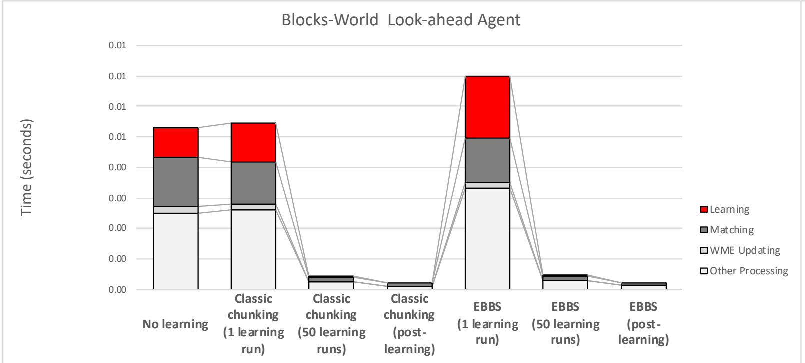
	Classic chunking (50 LRs)	Classic chunking (post-learning)	EBBS (50 LRs)	EBBS (post-learning)
Learning	0.000013	0.000000	0.000011	0.000000
Matching	0.000245	0.000224	0.000220	0.000207
WME Updating	0.000081	0.000059	0.000066	0.000039
Other Processing	0.000295	0.000280	0.000286	0.000259



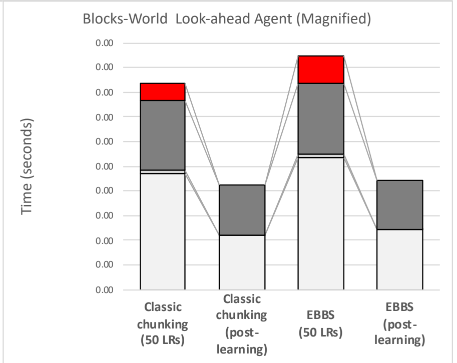
	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.000760	0.001596	0.000047	0.000001	0.001803	0.000038	0.000000
■ Matching	0.001912	0.002057	0.000192	0.000121	0.001776	0.000165	0.000121
□ WME Updating	0.000240	0.000215	0.000012	0.000002	0.000212	0.000010	0.000005
□ Other Processing	0.002910	0.003725	0.000366	0.000222	0.004224	0.000373	0.000238



	Classic chunking (50 LRs)	Classic chunking (post-learning)	EBBS (50 LRs)	EBBS (post-learning)
Learning	0.000047	0.000001	0.000038	0.000000
Matching	0.000192	0.000121	0.000165	0.000121
WME Updating	0.000012	0.000002	0.000010	0.000005
Other Processing	0.000366	0.000222	0.000373	0.000238

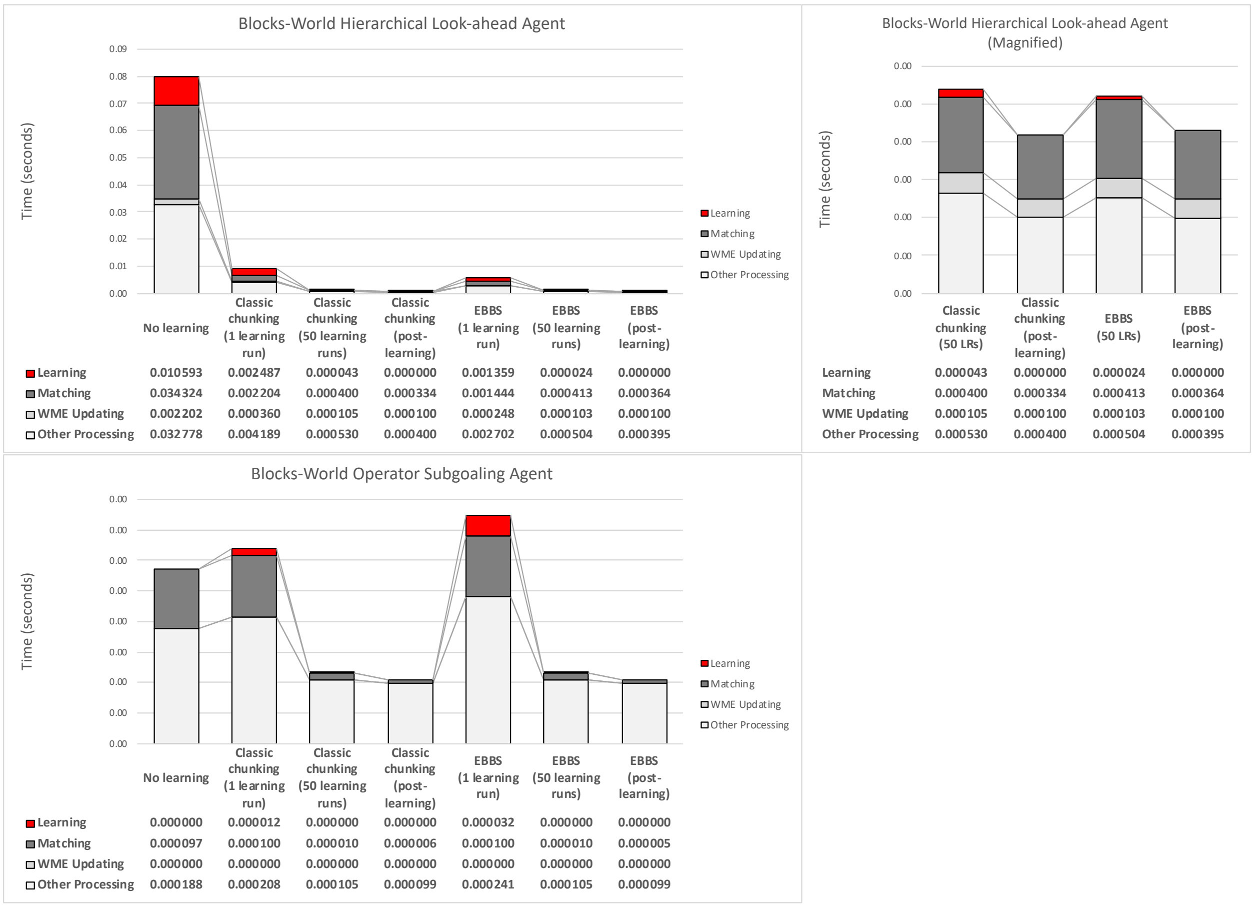


	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.000940	0.001270	0.000036	0.000000	0.002020	0.000056	0.000000
■ Matching	0.001618	0.001361	0.000140	0.000100	0.001441	0.000143	0.000100
□ WME Updating	0.000237	0.000186	0.000007	0.000000	0.000202	0.000008	0.000000
□ Other Processing	0.002492	0.002622	0.000235	0.000111	0.003316	0.000267	0.000122



	Classic chunking (50 LRs)	Classic chunking (post-learning)	EBBS (50 LRs)	EBBS (post-learning)
Learning	0.000036	0.000000	0.000056	0.000000
Matching	0.000140	0.000100	0.000143	0.000100
WME Updating	0.000007	0.000000	0.000008	0.000000
Other Processing	0.000235	0.000111	0.000267	0.000122

Figure 7.40: A breakdown of how the BlocksWorld Hierarchical, BlocksWorld Look-Ahead State Evaluation and BlocksWorld Look-Ahead agents spend their time.



Blocks-World Hierarchical Look-ahead Agent

	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.010593	0.002487	0.000043	0.000000	0.001359	0.000024	0.000000
■ Matching	0.034324	0.002204	0.000400	0.000334	0.001444	0.000413	0.000364
□ WME Updating	0.002202	0.000360	0.000105	0.000100	0.000248	0.000103	0.000100
□ Other Processing	0.032778	0.004189	0.000530	0.000400	0.002702	0.000504	0.000395

Blocks-World Hierarchical Look-ahead Agent (Magnified)

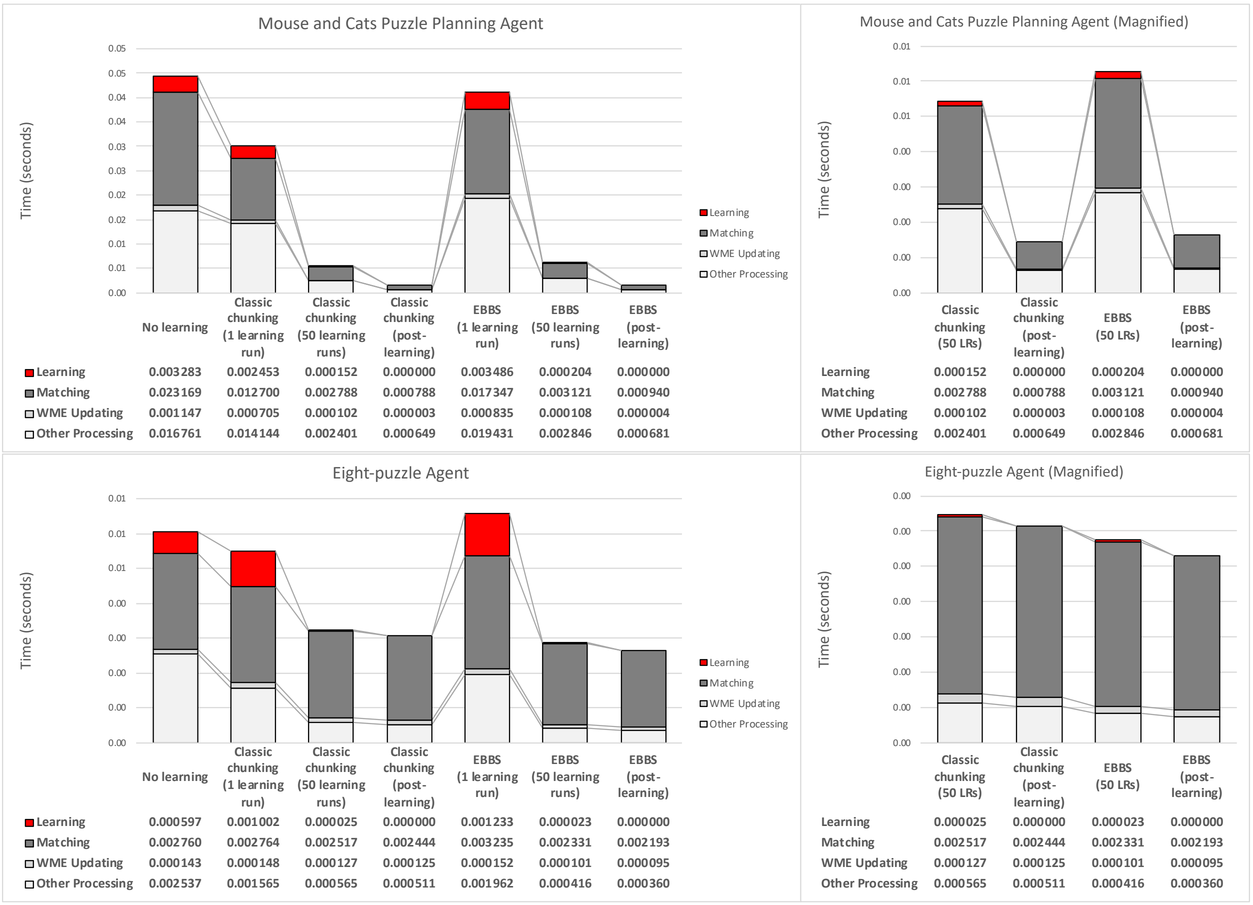
	Classic chunking (50 LRs)	Classic chunking (post-learning)	EBBS (50 LRs)	EBBS (post-learning)
Learning	0.000043	0.000000	0.000024	0.000000
Matching	0.000400	0.000334	0.000413	0.000364
WME Updating	0.000105	0.000100	0.000103	0.000100
Other Processing	0.000530	0.000400	0.000504	0.000395

Blocks-World Operator Subgoaling Agent

	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.000000	0.000012	0.000000	0.000000	0.000032	0.000000	0.000000
■ Matching	0.000097	0.000100	0.000010	0.000006	0.000100	0.000010	0.000005
□ WME Updating	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
□ Other Processing	0.000188	0.000208	0.000105	0.000099	0.000241	0.000105	0.000099

Figure 7.41: A breakdown of how the BlocksWorld Hierarchical Look-ahead and BlocksWorld Operator Subgoaling agents spend their time.

Figure 7.42 shows how the Mouse and Cats Planning and Eight-puzzle agents spend their time. In these two agents, O1, O2, O3 and O5 hold true. O4, which states that learning cost of the initial learning run can sometimes be higher than that of the non-learning run only holds for the Mouse and Cats Planning agent, where the learning cost of the initial EBBS learning run is greater than that of the non-learning run.



	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.003283	0.002453	0.000152	0.000000	0.003486	0.000204	0.000000
■ Matching	0.023169	0.012700	0.002788	0.000788	0.017347	0.003121	0.000940
□ WME Updating	0.001147	0.000705	0.000102	0.000003	0.000835	0.000108	0.000004
□ Other Processing	0.016761	0.014144	0.002401	0.000649	0.019431	0.002846	0.000681

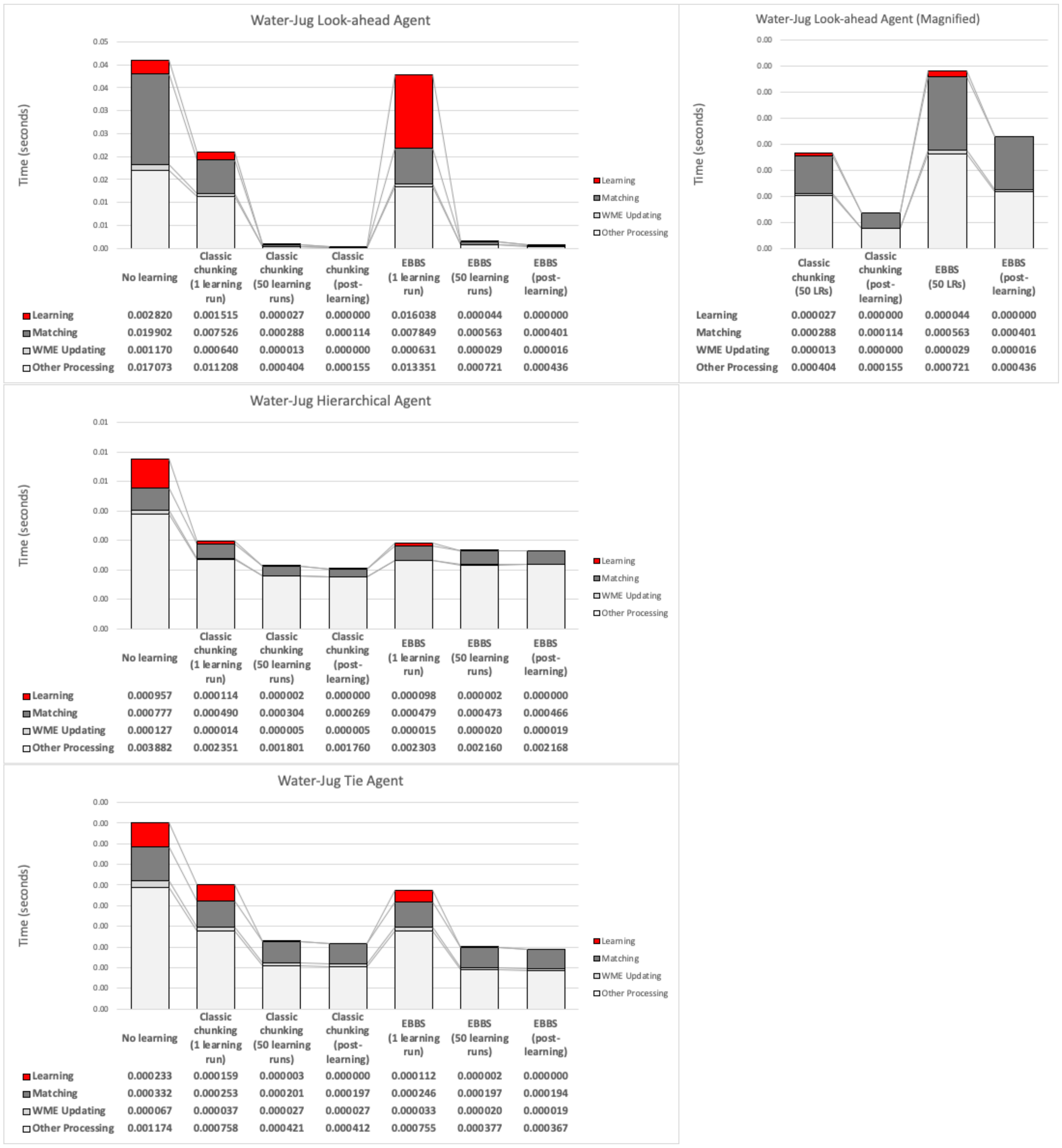
	Classic chunking (50 LRs)	Classic chunking (post-learning)	EBBS (50 LRs)	EBBS (post-learning)
Learning	0.000152	0.000000	0.000204	0.000000
Matching	0.002788	0.000788	0.003121	0.000940
WME Updating	0.000102	0.000003	0.000108	0.000004
Other Processing	0.002401	0.000649	0.002846	0.000681

	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.000597	0.001002	0.000025	0.000000	0.001233	0.000023	0.000000
■ Matching	0.002760	0.002764	0.002517	0.002444	0.003235	0.002331	0.002193
□ WME Updating	0.000143	0.000148	0.000127	0.000125	0.000152	0.000101	0.000095
□ Other Processing	0.002537	0.001565	0.000565	0.000511	0.001962	0.000416	0.000360

	Classic chunking (50 LRs)	Classic chunking (post-learning)	EBBS (50 LRs)	EBBS (post-learning)
Learning	0.000025	0.000000	0.000023	0.000000
Matching	0.002517	0.002444	0.002331	0.002193
WME Updating	0.000127	0.000125	0.000101	0.000095
Other Processing	0.000565	0.000511	0.000416	0.000360

Figure 7.42: A breakdown of how the Mouse and Cats and Eight-Puzzle agents spend their time.

Finally, Figure 7.43 shows how our last set of agents, the Water-Jug agents, spend their time. In these agents, again O1, O2 and O3 all hold true. O4, which states that learning cost of the initial learning run can sometimes be higher than that of the non-learning run, is true for the initial EBBS learning run of the Water-Jug Look-ahead agent but is not true for its classic chunking runs or any of the other agents. Similarly, O5 is only true for the Water-Jug Look-ahead agent; the learning cost of the initial EBBS run is greater than that of the initial classic chunking run. In the other two agents, the learning cost of the classic chunking initial learning run is greater.



Water-Jug Look-ahead Agent

	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.002820	0.001515	0.000027	0.000000	0.016038	0.000044	0.000000
▨ Matching	0.019902	0.007526	0.000288	0.000114	0.007849	0.000563	0.000401
□ WME Updating	0.001170	0.000640	0.000013	0.000000	0.000631	0.000029	0.000016
□ Other Processing	0.017073	0.011208	0.000404	0.000155	0.013351	0.000721	0.000436

Water-Jug Look-ahead Agent (Magnified)

	Classic chunking (50 LRs)	Classic chunking (post-learning)	EBBS (50 LRs)	EBBS (post-learning)
Learning	0.000027	0.000000	0.000044	0.000000
Matching	0.000288	0.000114	0.000563	0.000401
WME Updating	0.000013	0.000000	0.000029	0.000016
Other Processing	0.000404	0.000155	0.000721	0.000436

Water-Jug Hierarchical Agent

	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.000957	0.000114	0.000002	0.000000	0.000098	0.000002	0.000000
▨ Matching	0.000777	0.000490	0.000304	0.000269	0.000479	0.000473	0.000466
□ WME Updating	0.000127	0.000014	0.000005	0.000005	0.000015	0.000020	0.000019
□ Other Processing	0.003882	0.002351	0.001801	0.001760	0.002303	0.002160	0.002168

Water-Jug Tie Agent

	No learning	Classic chunking (1 learning run)	Classic chunking (50 learning runs)	Classic chunking (post-learning)	EBBS (1 learning run)	EBBS (50 learning runs)	EBBS (post-learning)
🟥 Learning	0.000233	0.000159	0.000003	0.000000	0.000112	0.000002	0.000000
▨ Matching	0.000332	0.000253	0.000201	0.000197	0.000246	0.000197	0.000194
□ WME Updating	0.000067	0.000037	0.000027	0.000027	0.000033	0.000020	0.000019
□ Other Processing	0.001174	0.000758	0.000421	0.000412	0.000755	0.000377	0.000367

Figure 7.43: How the Water-Jug agents spend their time.

In conclusion, while online procedural learning can lead to higher learning costs in early learning runs, they play an increasingly minor role in overall timing after the agent is exposed to enough problems that it has summarized the full space of reasoning that its underlying rules are capable of. EBBS does have higher learning costs than classical chunking in some agents, but that cost difference is also reduced with further learning.

7.5.3 Average and Maximum Decision Cycle Time

The goal of our computational overhead desiderata is to ensure that an agent maintains reactivity. Because an agent is given the opportunity to react once every decision cycle, an agent's reactivity is a function of the duration of a decision cycle. Consequently, the most important metric we use to evaluate computational overhead is the average and maximum decision cycle time.

Figure 7.44 shows the average decision cycle times for all of our evaluation agents across all three learning approaches. Note that the figure uses log 10 scale and includes a data point that corresponds to our maximum decision cycle time of 50ms.

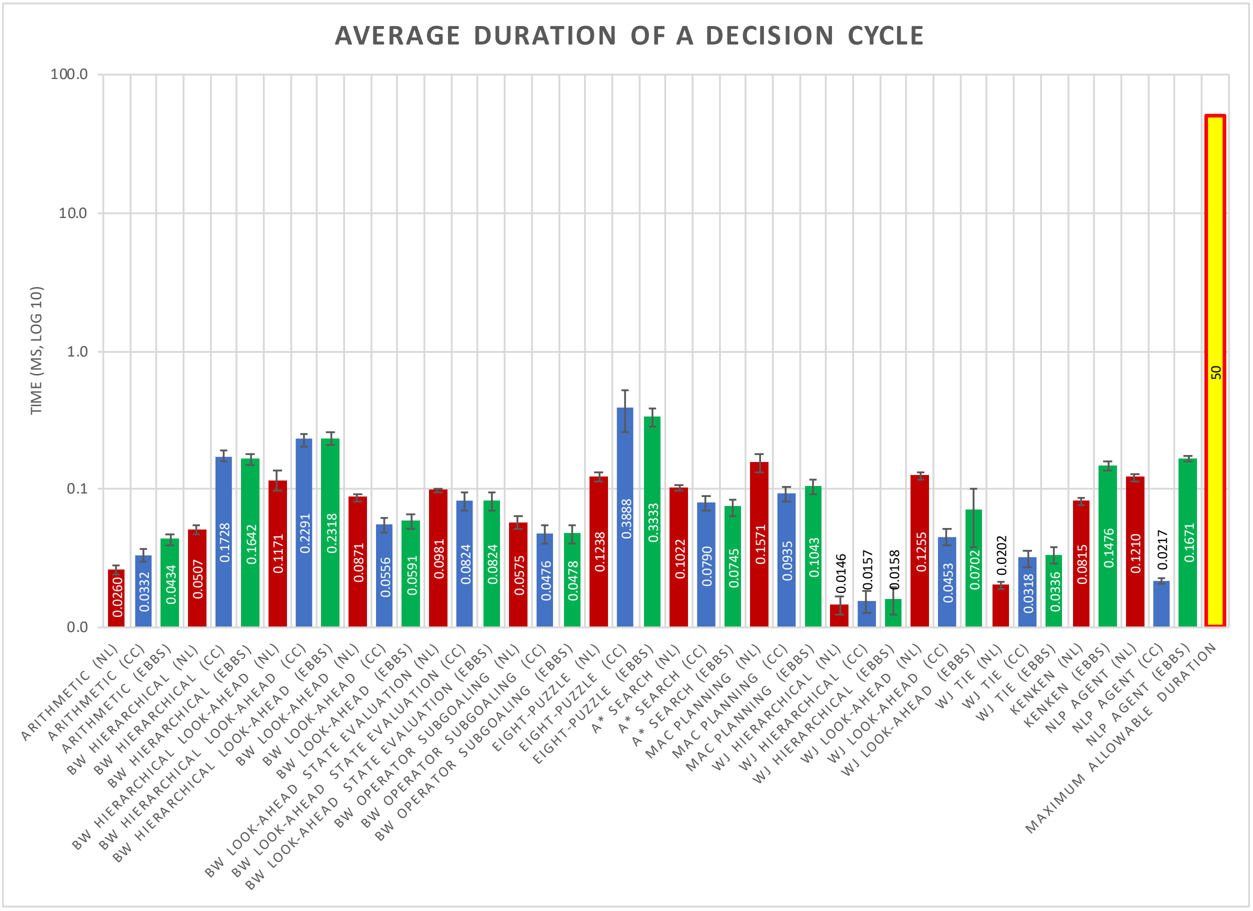


Figure 7.44: The average duration of a decision cycle. All agents are well below our target level of 50 ms/decision.

As we can see, the average decision cycle time is small for all of the agents. For the majority of them, it is under 0.1ms. The largest value is 0.38 ms, which is over two orders of magnitude less than our maximum decision cycle duration of 50 ms. The learning approach does have some effect on the value of this metric, but, in absolute terms, the amount is negligible.

We see a similar pattern when we look at the maximum decision cycle type, as shown in Figure 7.45.

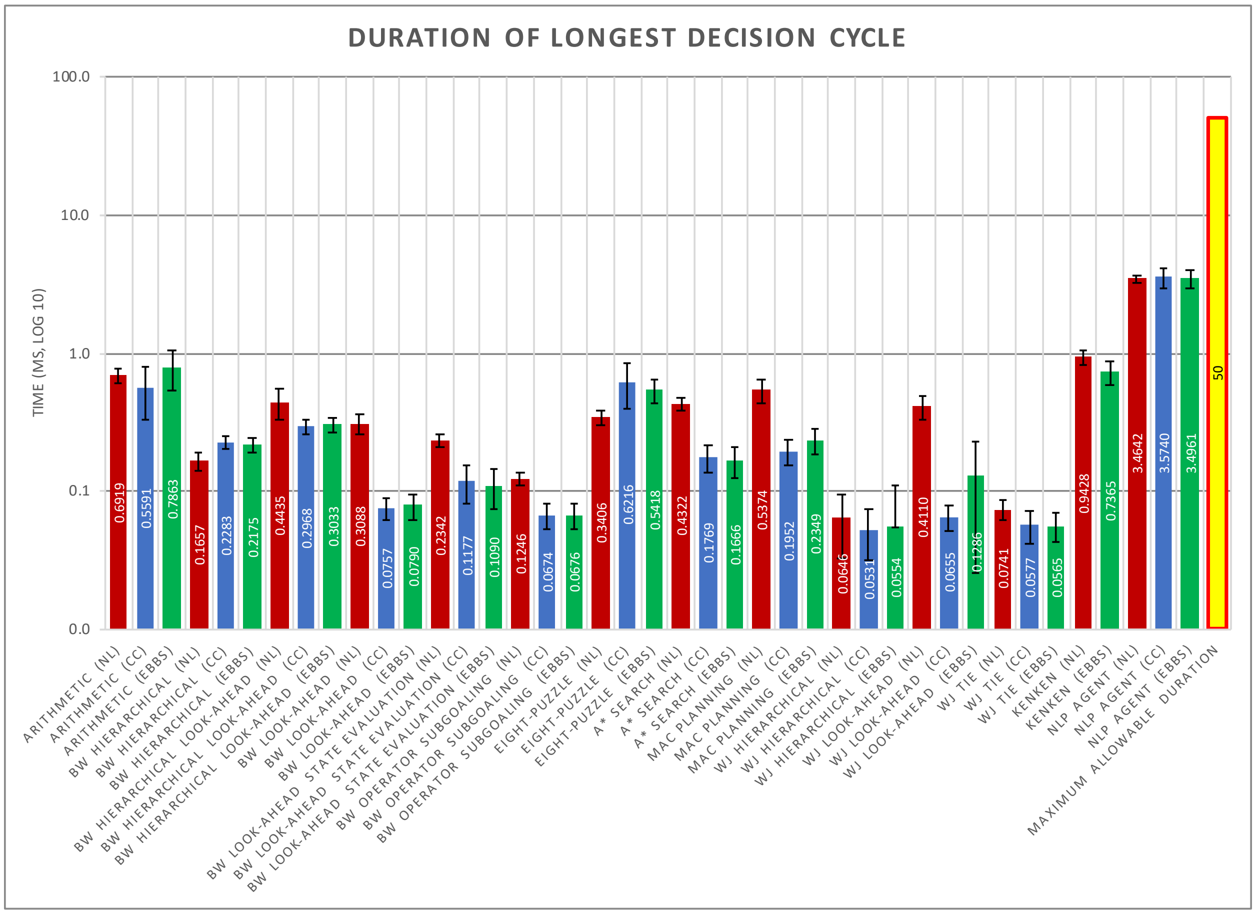


Figure 7.45: The duration of the longest decision cycle during a run. All agents are well below our target level of 50 ms/decision.

The learning approach does not seem to significantly affect the duration of the longest decision cycle time of an agent. There are some cases where EBBS has higher values compared to classic chunking,[13] most notable the arithmetic agent, but the differences are, again, negligible.

What is important is that the durations of the longest decision cycles *are under 50 ms*. As Figure 7.45 shows, the duration of the longest decision cycle is under 1ms for all agents and learning approaches except one, the NLP agent, which had at least one decision cycle that lasted 3.57 ms.

In conclusion, we can say the EBBS approach does not compromise agent reactivity; all of our decision cycle times, both average and maximum, are well below our target level of agent reactivity of 50 ms/dc.

[13]There are also a few agents where classic chunking has increased average or maximum decision times

7.5.4 Conclusions on our Evaluation of Computational Cost

We began this section by examining how the utility problem impacts our agents. While expensive chunks do still exist and are likely to be more common under EBBS, we found that there were no cases in our evaluation agents where they obviate the total speed-up provided by online procedural learning.

More importantly, our experiments have shown strong evidence that the additional costs of EBBS do not prevent an agent from maintaining reactivity. All of the agents had average decision cycle times under 0.39 ms/dc and all but one agent had a maximum decision cycle time under 1 ms. The slowest maximum decision cycle duration we saw in our evaluation were 3.57 ms/dc, which is more than a full order of magnitude below our target level of 50ms/dc.

7.6 Evaluating Correctness (E4)

In this section, we attempt to answer the most difficult of our evaluation questions: does the EBBS approach learns correct rules in situations that the current state of the art online procedural learning system cannot?

7.6.1 Empirical Evaluation of Correctness

While we were able to develop quantitative metrics to evaluate an algorithm's ability to capture generality, we found that it was much more difficult to evaluate correctness empirically.

To prove that rule is correct would be impractical. To review our definition of correctness from Section 3.5, *a learned rule is a correct summarization if it always generates the same knowledge structures that the original, deliberate reasoning would have in the same situation.* So, we would need to first explicate every possible context in which the original agent's deliberate subgoal reasoning could apply. We would then need to determine what inferences would arise from a subgoal that used deliberate reasoning to respond to that context, so that we could then compare them with the inferences that are generated from the rules that were learned based on the same underlying reasoning. If the two sets of inferences were identical, then we'd know the rule produces correct inferences in all the situations that the original reasoning produced inferences. At that point, we'd also have to examine all situations where the deliberate reasoning would not produce any inferences to make sure that the learned rules also don't produce any inferences. Obviously, it is not feasible to exhaustively list all situations where a set of rules could apply or could not apply, so an ideal version of this approach is not possible.

We did spend significant effort attempting to implement an ambitious tool that performed a more limited form of such meta-analysis. This tool, which we call *the chunk doctor*, monitors the subgoal reasoning of an agent and compares what happens in each subgoal with what would have happened had a learned rule fired. So, in some sense, the agent itself is used to create a meaningful space of situations for examining the correctness of an agent's learned rules. But, the real trick behind the chunk doctor is that it learns rules, keeps them in procedural memory, *but does not allow them to fire*, which forces a subgoal to be generated, which then leads to deliberate reasoning in the subgoal. Because the rules don't fully fire, the chunk doctor can perform meta-analysis and compare the deliberate reasoning of an agent with what would have happened had learning truly been enabled.

The first way we tried to detect whether a learned rule is correct is to have the chunk doctor verify that a duplicate rule would be learned from the same context that led the learned rule to match. So, whenever a learned rule matches, the chunk doctor inhibits the rule from firing and tracks what rules are learned from the subgoal. If no subgoal is generated or the same chunk is not re-learned, the chunk doctor concludes that the matched chunk fired in a situation that the original reasoning did not support, which would mean that the matched chunk was overgeneral and, hence, incorrect.

Unfortunately, this method was hampered by the fact that the RETE sometimes fails to detect that a rule is a duplicate of another previously added rule. Because every one of these failures was a false positive that required manual comparison of rules, this approach proved to be very time intensive. Eventually, we abandoned this idea until more work can be done to improve the RETE's ability to detect duplicate rules.

The second way the chunk doctor detects whether a learned rule is correct is to compare the WMEs generated by the subsequent subgoal and compare them to the WMEs that would have been generated by the learned rules that were ready to fire but were instead inhibited.[14] So, instead of monitoring the rules that are learned from the subgoal reasoning, the second method monitors the inferences that are generated by subgoal reasoning. To compare these WMEs to the WMEs that would have been generated by the inhibited learned rule, the chunk doctor simulates the firing of the inhibited learned rule. The firing is simulated in the sense that the WMEs that would arise from the rule will never be added to the agent's memory and instead are set aside to be used for comparison with the WMEs generated by the deliberate reasoning in the subgoal. This comparison occurs when the subgoal ends. If other chunks also match and are inhibited while reasoning in the subgoal is occurring, the rule and its simulated inferences are added for comparison.

For some agents, the chunk doctor did show that they were generating the same inferences using learned rules that they generate with deliberate reasoning. But for many agents, small differences

[14]In reality, the chunk doctor compares preferences for WMEs, but, for the sake of clarity, we'll ignore that level of processing.

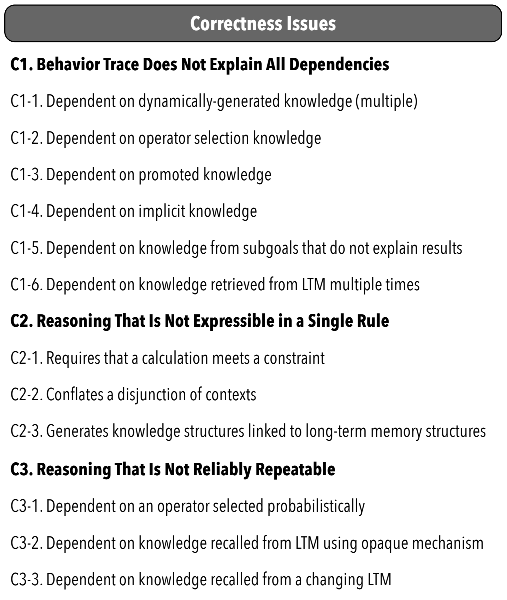
in the preferences that arose from each subgoal would flag a chunk as incorrect that proved to be correct. For example, because other elements of working memory may have changed, additional rules may fire in the subgoal that led to data structures being generated in the supergoal that weren't generated from the original subgoal; this would cause all learned rules that were currently inhibited to be flagged as a potentially incorrect. After time-consuming manual inspection, we would find an explanation for the discrepancy. So, like the previous technique, dealing with false positives proved to be too time intensive for this approach to be of use.

In conclusion, while the chunk doctor shows some limited promise as a verification mechanism, we did not have success using it as a correctness evaluation or a diagnostic tool. As an evaluation tool, the chunk doctor proved to be more labor intensive and less informative than our explicit correctness issue detectors. As a diagnostic tool, the chunk doctor would not be informative enough because, even if it did accurately detect a case where a chunk produced different inferences than deliberate reasoning, it would not be able to offer any explanation of why the rule was incorrect. While we certainly could have improved it further, we weren't confident that something robust could be built. So, after about a year of development and experimentation, we abandoned our efforts to empirically evaluate correctness using the chunk doctor.

7.6.2 Correctness Regression Test Agents

To evaluate correctness, we use a suite of *correctness regression test agents*. Each of these agents creates a subgoal and performs reasoning that is designed to produce a trace that will require the learning algorithm to deal with one of the correctness issue originally introduced in Section 3.5, as shown in Figure 7.46.

While these agents are simple, a detailed, description of every one would be taxing to follow and is more detail than is necessary to understand the outcomes of this experiment. So, we present a low-level description of operation for only one of the agents, which should allow the reader to understand how the regression tests agents generally work. For the remaining agents, descriptions are provided, but they are more abstract. The full Soar agent code for all of these agents are available for download and inspection in the evaluation agents package described in Section 7.1.3.


Correctness Issues

C1. Behavior Trace Does Not Explain All Dependencies

C1-1. Dependent on dynamically-generated knowledge (multiple)

C1-2. Dependent on operator selection knowledge

C1-3. Dependent on promoted knowledge

C1-4. Dependent on implicit knowledge

C1-5. Dependent on knowledge from subgoals that do not explain results

C1-6. Dependent on knowledge retrieved from LTM multiple times

C2. Reasoning That Is Not Expressible in a Single Rule

C2-1. Requires that a calculation meets a constraint

C2-2. Conflates a disjunction of contexts

C2-3. Generates knowledge structures linked to long-term memory structures

C3. Reasoning That Is Not Reliably Repeatable

C3-1. Dependent on an operator selected probabilistically

C3-2. Dependent on knowledge recalled from LTM using opaque mechanism

C3-3. Dependent on knowledge recalled from a changing LTM

Figure 7.46: Regression test agent were designed for most of these correctness issues.

Because we do not need to generate quantitative metrics for these tests, we do not use the simulated version of classical chunking and instead compare learning algorithms by first running a regression agent in EVAL-SOAR and then running the same agent in the last version of Soar with classical chunking, Soar 9.3.2.

For each regression agent, we describe three qualities of what we observe:

1. What happens when EBBS summarizes the agent's reasoning

2. What happens when classic chunking summarizes the agent's reasoning

3. When applicable, whether each of the learning algorithms can detect the correctness issue before the rule is added to procedural memory

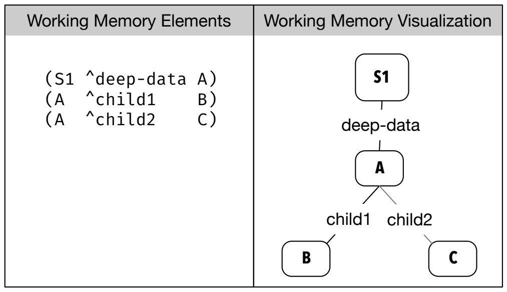
   This section is included if EBBS fails. Classical chunking has few correctness issue detectors, so the reader may assume that a detector does not exist in classical chunking unless otherwise noted.

The first eight tests involve correctness issues that arise when summarizing a behavior trace that does not explain all dependencies. These agents are designed to test as little information in the supergoal as possible. In most cases, this makes failures obvious; the agent learns a rule with no conditions, which is rejected by Soar. While this type of invalid rule may seem different from an incorrect rule, it really isn't; if we had designed the agent so that it directly tests an additional WME in the supergoal, it would have successfully learned a rule, one that was overgeneral and hence incorrect.

The agent that we describe in detail is the agent that uses a dynamic rule in its subgoal reasoning. To review, a dynamic rule is one that has some of its structure determined dynamically at run-time, for example, based on the contents of working memory.

7.6.2.1 Behavior Trace Dependent on a Dynamic Rule

- **Initial setup**: This agent will use the RHS function deep-copy to test whether an agent can learn from a dynamic rule. The knowledge that is deep-copied is a multi-level working memory structure that the agent creates on the top goal as shown in Figure 7.47. The actual items copied are thus determined at run-time, and that is the dynamic aspect of this rule.



Working Memory Elements	Working Memory Visualization
(S1 ^deep-data A) (A ^child1 B) (A ^child2 C)	

Figure 7.47: Initial state of WM of regression test agent C1-1(a).

- **Operation**: The agent subgoals, proposes and selects a subgoal operator. A rule then fires that uses the dynamic function deep-copy to create a copy (in the subgoal) of A and all of its children. Another rule fires and returns a copy of the top node A, a new WME D, as a result to the supergoal. (This makes the children accessible to the supergoal, so they are also results.) Figure 7.48 shows these two rules.

- **Desired Behavior**: A rule is learned that tests A, B and C and creates a structurally equivalent structure: D, E and F. In Figure 7.49, A, B and C would match <f1> <f2> and <f3>, while D, E and F would be created by the RHS unbound variables <f4>, <f5> and <f6>. Note that B and C are not tested in any of the conditions of the rules shown in Figure 7.48 but are implicit in the operation of deep-copy.

- **EBBS Behavior**: EBBS learns the expected rule and passes this regression test.

```
sp {perform-deep-copy
   (state <s> ^operator.name deep-copy
              ^superstate <ss>)
   (<ss> ^deep-data <a>)
   -->
   (<s> ^deep-copy-in-subgoal (deep-copy <a>))
  }

sp {add-deepcopy-to-supergoal
   (state <s> ^operator.name deep-copy
              ^superstate <ss>
              ^deep-copy-in-subgoal <a2>)
   (<ss> ^operator <so>)
   -->
   (<ss> ^deep-data2 <a2>)
  }
```

Figure 7.48: The dynamic rule (top) and the rule that produces a result in the supergoal (bottom.)

```
sp {desired-learned-rule
   (state <s> ^superstate nil
              ^deep-data <f1>)
   (<f1> ^child1 <f2>
         ^child2 <f3>)
   -->
   (<s>  ^deep-data2 <f4>)
   (<f4> ^child1     <f5>)
   (<f4> ^child2     <f6>)
  }
```

Figure 7.49: The correct rule that should be learned by regression test agent C1-1(a).

```
sp {EBBS-rule-learned
   (state <s> ^superstate nil
              ^deep-data <f1>)
   (<f1> ^child1 <f2>
         ^child2 <f3>)
   -->
   (<s>  ^deep-data2 <f4>)
   (<f4> ^child1     <f5>)
   (<f4> ^child2     <f6>)
  }
```

Figure 7.50: The EBBS rule learned from regression test agent C1-1(c).

- **Classic Chunking Behavior**: Classic chunking fails this regression test. The agent learns an incorrect rule that only matches A before it creates D, E and F. This rule is overgeneral because it does not also test A's children.

```
sp {classic-chunk-learned
   (state <s> ^superstate nil
              ^deep-data <f1>)
   -->
   (<s>  ^deep-data2 <f4>)
   (<f4> ^child1     <f5>)
   (<f4> ^child2     <f6>)
  }
```

Figure 7.51: The classic chunk learned from regression test agent C1-1(c).

7.6.2.2 Behavior Trace Dependent on Architectural Meta-data

We evaluate whether an online procedural learning algorithm can learn from a trace dependent on architectural meta-data, for example the architecturally created features in a subgoal that indicate which operators are tied in the supergoal, by using an agent that performs subgoal reasoning that only tests meta-data. This agent faces an operator tie impasse in the top-level goal, which causes the agent to subgoal. In the subgoal, a rule fires that tests supergoal operators A and B *via the architectural meta-data* and creates a preference in the supergoal to prefer A over B.[15] What's important is that this rule *does not directly test the operators in the supergoal*; instead, it tests the operators referred to by meta-knowledge that Soar creates in the subgoal that list all tied operators. The correct rule is one that tests that both operators A and B are proposed in the current state, and, if so, creates a preference preferring A over B.

- **EBBS Behavior**: EBBS learns the expected rule and passes this regression test.
- **Classic Chunking Behavior**: Classic chunking learns the expected rule and passes this regression test. As described in 6.1.1.1, both EBBS and classic chunking automatically create architectural instantiations for these subgoal features.

7.6.2.3 Behavior Trace Dependent on Knowledge Recalled from LTM

This agent tests whether an online procedural learning algorithm can learn correct rules from LTM memory recalls. As previously discussed in Section 6.1.1.2, summarizing traces that test LTM is a challenge because structures generated by long-term memory retrievals are dynamically-created by the architecture, so there is no computational accounting of any dependency between the context of the query and the knowledge retrieved. To test whether an agent can learn from such traces, this agent proposes an operator in a subgoal that will issue a semantic memory recall. The proposal

[15]Note that correctness issues involving learning from operator selection knowledge do not exist here. This agent is not summarizing reasoning that includes operator selection knowledge, which is where correctness issues lie, and is simply creating operator selection knowledge.

for the subgoal operator tests a feature A in the supergoal. The operator is selected and issues a semantic memory recall command. The semantic memory command creates a data structure B in the subgoal that corresponds to a long-term memory structure. An operator application rule then fires that returns that structure to the supergoal. The correct rule is one that tests feature A and creates a knowledge structure B in the same goal.

- **EBBS Behavior**: EBBS learns the expected rule and passes this regression test.
- **Classic Chunking Behavior**: Classic chunking passes this regression test.

7.6.2.4 Behavior Trace Dependent on Operator Selection Knowledge

This agent tests whether an online procedural learning algorithm can learn correct rules that behavior traces that include operators selected using operator selection knowledge. This agent proposes two operators in a subgoal. A rule fires that tests feature A in the supergoal and creates an operator preference that prefers one of the subgoal operators over the other. The preferred operator is selected and an operator application rule then fires that returns a result, feature B, to the supergoal. The correct learned rule tests feature A, which was originally tested by the OSK rule, and creates feature B.

- **EBBS Behavior**: EBBS learns the expected rule and passes this regression test.
- **Classic Chunking Behavior**: Classic chunking fails this regression test. Agent learns a rule that creates feature B but has no conditions that it rejects.

7.6.2.5 Behavior Trace Dependent on Promoted Knowledge

This agent tests whether an online procedural learning algorithm can learn from a behavior trace that inaccurately describes knowledge as subgoal knowledge that is actually supergoal knowledge. As discussed in Section 6.1.3, this occurs when another rule previously fired in the subgoal and returned the knowledge structure as a result, effectively promoting the knowledge to supergoal knowledge.

This agent creates a WME A in the top goal. The agent then subgoals, after which it proposes and selects an operator in the subgoal. An operator application rule then fires and creates subgoal WME B. Another rule fires and adds a child WME to A that points to B. Because A is in the supergoal, this promotes B to supergoal knowledge. Finally, a rule fires in the subgoal that references B relative to the subgoal and adds a child WME C to B. Because B was promoted to supergoal knowledge, C is also being added to the supergoal, so a rule should be learned. The correct rule is one that tests A and B and creates C as a child.

- **EBBS Behavior**: EBBS learns the expected rule and passes this regression test. Note that this agent initially learns an invalid rule that tests for B and creates C. EBBS uses rule repair to find a path from the supergoal to A to B. This allows it to add a condition that tests A, which provides a path to the goal and makes the rule valid.

- **Classic Chunking Behavior**: Classic chunking fails this regression test. Soar reports a critical error and exits, which is a problematic response, given that this is one of the most common correctness issues agents encounter. (We assume that the reason that it exits is that the researchers who implemented the system felt that stability of Soar would be compromised if they added unjustified knowledge to the agent.)

7.6.2.6 Behavior Trace Dependent on Implicit Knowledge

This agent tests the one detectable case of an incorrect rule that arises from the use of implicit knowledge: behavior traces that tests that a knowledge structure *does not exist* in the subgoal. As previously discussed, these traces only produce incorrect rules if the agent has other rules that both create the prohibited structure in the subgoal and test supergoal knowledge. If the agent designer has implicit knowledge that such rules do not exist, then the rule is correct.

This agent is fairly simple. It creates A in the top goal, subgoals, tests that some hypothetical WME B does not exist in the subgoal and then returns B as a result. We also add another rule, which never fires, which tests for C in the topgoal and adds B to the subgoal. While this rule never actually fires[16], its existence means that the implicit knowledge used by the "designer" was wrong, so any rule learned from a trace that tests that B doesn't exist in the subgoal is incorrect.

- **EBBS Behavior**: EBBS fails this regression test.

- **Classic Chunking Behavior**: Classic chunking fails this regression test.

- **Detection**: Neither OPL algorithm can verify the implicit assumption that no rules exist that can fire in a supergoal and create the prohibited subgoal WME. What they both *do* detect, are rules learned that summarize behavior traces that test that a subgoal WME does not exist, which is information that can help the agent engineer to diagnose whether the correctness issue exists in an agent. If they find that only subgoal rules can create the prohibited WME, they can ignore the warning. If they realize that there are rules that can create the prohibited subgoal WME, they can eliminate the correctness issue by redesigning their rules so that the prohibited WME is instead created in the supergoal and any rules that test that the existence of the WME test it relative to the supergoal. Both EBBS and classic chunking also include a

[16]At the risk of making a joke, this rule is truly symbolic.

filter for this issue that, when enabled, will throw out the learned rule and force Soar to learn a justification instead. EBBS also includes an option that will flag the chunk as potentially incorrect but allow it to be learned and added to procedural memory.

7.6.2.7 Behavior Trace Dependent on Knowledge from Non-learning Subgoals

As previously discussed in Section 6.1.5, correctness issues can arise from non-learning subgoals because the mechanisms that prevent learning incorrect rules are not employed in those subgoals. Consequently, we use a modified version of the regression test for C1-2, which test whether an agent can learn from operator selection knowledge, to creates a correctness issue for this agent. The main difference between the agents is that this one subgoals twice, which results in a stack of three goals, which we will refer to as G1, G2, and G3, but *only enables learning in the middle goal G2*. A WME A is created in G1 and a WME B is created in G2. This agent then uses the logic from the C1-2 agent to propose two operators in G3. And like the C1-2 agent, an operator selection rule fires in G3 that *tests A in G1* and creates an operator preference for one of the proposed G3 operators. That operator is selected, and then an operator application rule tests B, which is in G2, and returns a result to G2: WME C. Finally, a rule fires in G2 that tests only C in G2 and returns a result D to G1. The OPL algorithm should then learn a rule that tests feature A, which was tested by the OSK rule in the non-learning subgoal G3, and creates feature D.

- **EBBS Behavior**: EBBS learns the expected rule and passes this regression test.
- **Classic Chunking Behavior**: Classic chunking fails this regression test. Because classic chunking does not incorporate OSK in the rules that it learns, the justification that was learned in G3 did not test A, which led the final chunk to also not test A. Consequently, the chunk is overgeneral.

7.6.2.8 Behavior Trace Dependent on Knowledge Retrieved from LTM Multiple Times

This agent tests whether an OPL learns correct rules in agents that retrieve the same LTM in multiple subgoals, a situation that can cause unintended interactions between LTM recalls. This agent creates a WME A and then issues a semantic memory recall command in the top goal. After the LTM structure is retrieved into the top goal, the agent subgoals. Rules fire in the subgoal that issue another semantic memory recall command that retrieves the same LTM into the subgoal. A rule fires that tests A in the supergoal and one of the WMEs in the subgoal version of the LTM and add a child WME, B, to it. Because the LTM is technically supergoal knowledge, the OPL algorithm should learn a rule that tests A and creates a child WME connected to it, B.

Note that the desired rule is technically also incorrect. This is because the agent tests knowledge from LTM that is retrieved directly into the subgoal, which both EBBS and classical chunking will ignore because it is knowledge that is not accessible to the supergoal.[17] But, this particular overgenerality can be ignored because it is simply the nature of this correctness issue. The agents that face these problems are intentionally using semantic memory to hide certain dependencies in the agent's reasoning from the OPL algorithm. This process, which we call context-specific decontextualization is a key mechanism by which other research projects have used both chunking and EBBS to effect higher-order procedural learning, for example, learning by instruction.

- **EBBS Behavior**: Instance-based semantic memory allows EBBS to avoid this problem entirely because it ensures that each LTM recall is independent and cannot interact with another LTM recall. So, EBBS learns the expected rule and passes this regression test.

  Note that EBBS would still be able to learn a correct rule, even if instance-based semantic memory were not being used. Previous versions of EBBS used rule-repair to create conditions that correct unconnected conditions that arose from the missing dependencies in the behavior trace caused by unexpected interactions between LTM recalls.

- **Classic Chunking Behavior**: Classic chunking fails this regression test and generates a rule where one of the conditions, the one that would match A, is not connected to a goal. Soar reports a critical error and exits. (It reports that the agent may have modified previous results.)

7.6.2.9 Behavior Trace with Reasoning that Requires that a Calculation Meets a Constraint

This agent implements reasoning that is similar to the example used in Section 6.2.1. The agent creates three numerical features on the top goal, N1, N2 and N3, which are equal to 1, 2 and 3 respectively. In the subgoal, a rule fires that adds N1 and N2 together using a RHS function which it then stores in a subgoal WME A. Another rule fires that tests both N3 (using a variable) and whether the value stored in A is less than 5, and, if so, it uses a RHS math function to return the sum of N3 and A as a result. The correct learned rule has conditions that test N1 and N2 using specific values (1 and 2) and tests N3 using a variable. The correct rule learned will have a RHS that is a math function that adds N3, again using a variable, with the specific value of the sum, three.

- **EBBS Behavior**: EBBS learns the expected rule and passes this regression test.

- **Classic Chunking Behavior**: Classic chunking can neither capture the generality of numerical reasoning nor learn rules with RHS math functions, so this issue does not apply. (The

[17]This is why the agent also tests A in the supergoal; testing A guarantees that the rule will have at least one positive condition and be valid.

classic chunk will always contain the specific value of numerical features and the RHS calculations.) As previously discussed in Section 6.2.1, this correctness issue is one that arose when we gave EBBS the ability to learn rules that capture the mathematical reasoning in RHS functions.

7.6.2.10 Behavior Trace with Reasoning that Conflates a Disjunction of Contexts

To review, this problem occurs when multiple rules test different knowledge structures in the supergoal but create the same WME in the subgoal. To correctly learn a rule in such situations, the cognitive architecture would need to support rules with disjunctive conditions, something Soar does not have the language to do currently.

A detector for context conflation exists but is still a work in progress. While it did function properly, it required some low-level architectural changes[18] that introduced general stability issues. As a result, we decided to disable the detector and re-examine the technical issues in future work.[19]

Because we did not get our conflation detector fully working and neither EBBS nor classic chunking include a mechanism to ameliorate this problem, we did not implement a regression tests for this issue.

7.6.2.11 Behavior Trace that Generates Knowledge Structures Linked to LTM

This issue is unique because it is one that is a direct consequence of instance-based semantic memory. As discussed in Section 6.3.3, if an agent returns a result that has internal links to LTMs, it must create chunks that also creates a working memory structure that is linked to the same LTM. Otherwise, the rule would be incorrect. If the agent performs further reasoning that require that the LTM links exist – for example, the rule may use one of the new semantic memory tests to test that two short-term working memory structures were linked to the same semantic memory – the reasoning would match against the original results but would not match against the WMEs produced by the chunk.

This is probably the simplest of the regression test agents. The agent first creates A in the top goal. It then subgoals and retrieves a knowledge structure B, which has an internal link to a structure in LTM. It tests A in the supergoal and returns B to the supergoal. The correct rule is one that tests A and creates a knowledge structure that is both structurally equivalent to B and linked to the same LTM as B.

- **EBBS Behavior**: EBBS learns the expected rule and passes this regression test. The rule

[18] Keeping all preferences in a slot ordered, for those who are curious.

[19] Part of the reason behind this decision is that we've never actually encountered this correctness issue in any of our agents, so we always considered it low priority.

learned contains special actions that create internals links between the WMEs that the rule creates and the same LTM that was linked to B.

- **Classic Chunking Behavior**: This problem does not exist in classic chunking.[20]

7.6.2.12 Behavior Trace with Reasoning that is Dependent on a Probabilistic Operator

This agent is straightforward. It creates two WMEs in the top goal, A and B. It then subgoals and proposes one operator that tests A and one operator that tests B. It selects between the two operators randomly. An operator application rule then fires that creates a result in the supergoal that points to the WME that its operator proposal tested (either A or B). Because Soar rules cannot represent such probabilistic reasoning in a rule, there is no rule that can be learned from this trace that will always produce the correct inferences.

- **EBBS Behavior**: EBBS fails this regression test. Agent learns an incorrect rule that tests A and creates a WME that points back to A. If the two operators proposed in the subgoal had an equal chance to be selected, this rule will be incorrect half the time. (Because the deliberate reasoning also had a 50% chance to create a WME that points back to B.)
- **Classic Chunking Behavior**: Classic chunking fails this regression test and also learns one of the two incorrect rules.
- **Detection**: EBBS can detect rules with this correctness issue. When EBBS is analyzing OSK to build a ROSK set, it also flags each operator with information indicating whether the operator was chosen probabilistically. If it encounters any instantiations that tests such probabilistically chosen operators when it is backtracing through the behavior trace, it flags the learned rule that it's building accordingly. If the correctness filter for this issue is enabled, EBBS does not add the rule to memory and instead builds a justification for it instead. Classic chunking does not have a mechanism to either detect this correctness issue or block these rules from forming.

7.6.2.13 Behavior Trace Dependent on Knowledge from LTM that is Either Changing or Retrieved Using an Opaque Mechanism

This agent proposes an operator in a subgoal that will issue a semantic memory recall. The proposal for the subgoal operator tests a feature A in the supergoal. The operator is selected and issues a semantic memory recall command. Soar responds to the query and creates a data structure B in the subgoal that corresponds to a long-term memory structure B*. An operator application rule

[20] Note that if instance-based semantic memory had existed back then, classic chunking would also have this issue.

then fires that returns B to the supergoal, which causes a rule to be learned. A set of rules then fire in the supergoal that creates a structure C and issues a semantic memory store command that will effectively replace B* with a new LTM based on the structure C. Because the LTM knowledge that the trace is dependent on, B*, no longer exists, there is no correct rule that can be learned from this trace.

- **EBBS Behavior**: EBBS fails this regression test. It learns a rule that tests feature A and creates a knowledge structure B based on and linked to B*. Because B* was removed from LTM, this rule is incorrect.
- **Classic Chunking Behavior**: Classic chunking fails this regression test and also learns the same incorrect rule.
- **Detection**: EBBS can detect rules with this correctness issue. If any rules in a behavior trace test knowledge structures retrieved from LTM, EBBS flags the chunk accordingly. If the correctness filter for either of these issues is enabled, EBBS does not add the rule to memory and instead builds a justification for it instead. Note that this detector differs from the one described in Section 7.6.2.3, which detects rules learned from behavior traces that test LTM knowledge structures that are generated by a LTM retrieval that is also in the behavior trace. This detector flags *any rule learned from a trace that tests at least one WME that is linked to a LTM, even if it was retrieved in a different subgoal*. If the correctness filter for this issue is enabled, EBBS does not add the rule to memory and instead builds a justification for it instead. Classic chunking does not have a mechanism to either detect this correctness issue or block these rules from forming.

Note that this agent can also be used to evaluate whether an OPL algorithm can learn from reasoning that is dependent on knowledge from LTM retrieved using an opaque mechanism. The two issues are functionally equivalent because they both arise when two identical LTM recalls return different knowledge structures. They differ in only the underlying reason.

7.6.3 More Complex Regression Tests

In the spirit of transparency, it is worth noting that the regression tests described in the previous sections were created purely to facilitate this discussion.[21] They were not actually used in the development of EBBS. Instead, we used a suite of 100 more complex agents as correctness and generality unit tests.

[21]It was obvious to us beforehand how each agent would do on each test.

To use these agents as regression unit tests, we developed a "key" for each agent: a set of "gold standard" rules that we expected the agent to learn. To generate keys for the 77 simple agents, we ran the agents and manually verified that each rule that the agent learned is correct. (These agents were added over several years. Each time we faced and overcame some issue with our algorithm, we would add an agent to test it.) For the more complex agents, manually verifying the rules learned was not practical. And, as previously discussed, we have no automated method to determine or prove that a set of learned rules are correct. So, we essentially used task performance to evaluate whether the agent was learning the rules we wanted. If the agent could solve the problem reliably, we would add the agent to our test agents and use the rules that it learned at the time as its key. As we improved and refined the algorithm, we would update the rules in each agent's key.

The agents in our unit tests include:

- 12 of the evaluation agents we described in the generality, correctness, task performance and computational overhead sections of this chapter.[22]
- Two agents that model PRIMS learning (Stearns et al., 2017), which were derived from the agents used in Bryan's Stearns thesis.
- One agent based on James Kirk's game-learning agent (Kirk and Laird, 2014). This agent learns the rules for 90 different games and is the largest and most complex of our unit test agents.
- Eight agents based on reasoning performed by ROSIE (Mininger and Laird, 2016), a robot that learns by instruction.[23]
- 77 agents that test specific aspects of EBBS. These simple agents are at about the same level of complexity as the correctness regression tests that we described in this section.

These agents and their keys can be found in the current distribution of Soar. To run all 100 of them and see how their rules compare to their keys, the reader can run the `UnitTests` executable. (The `UnitTests` program uses the RETE duplicate detection capabilities to verify that the rules learned matched the rules in the agent's key.)

Note that we did not use the PRIMS, game-learning, or ROSIE agents in our detailed evaluation because none of them were able to complete their task either with classical chunking enabled or when learning is disabled.

[22]The NLP and Kenken agents, which were developed after EBBS was completed, are the two that are not included.

[23]None of these are complete ROSIE agents. They are simplified excerpts of ROSIE's reasoning that were given to me every time my algorithm wasn't able to learn something they wished it could learn.

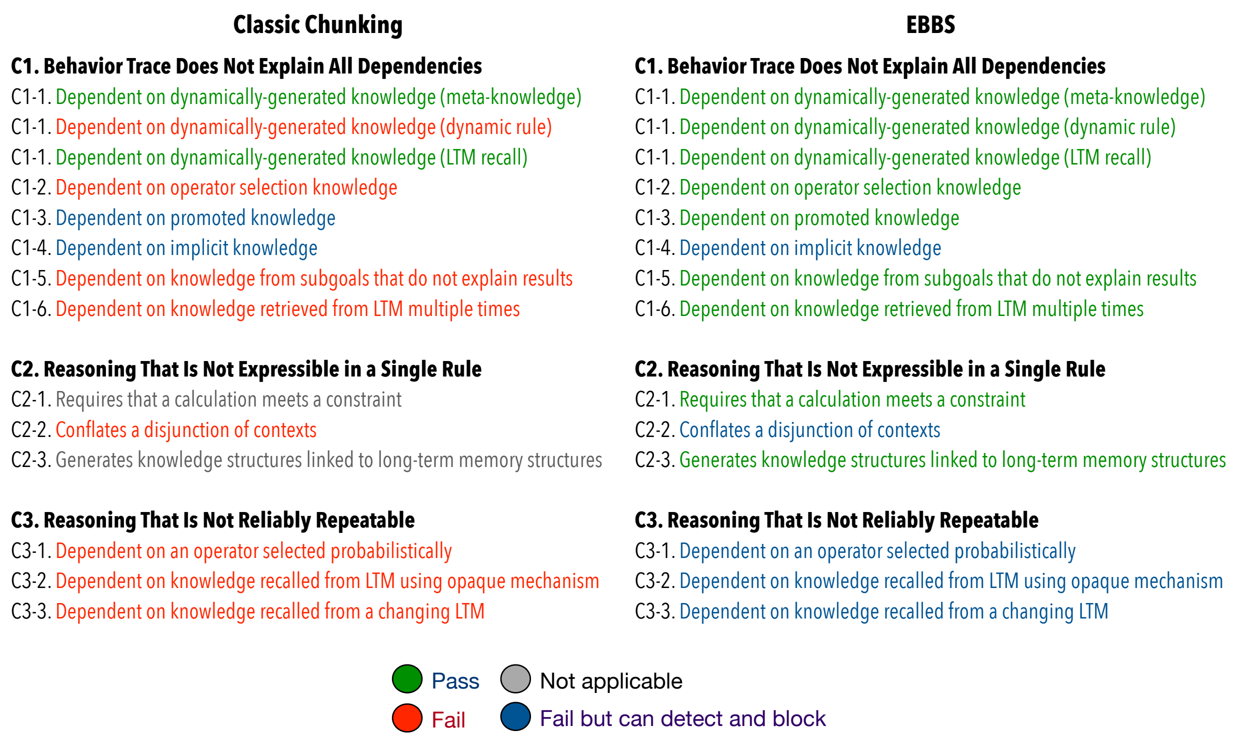
7.6.4 Additional Comments on Correctness

The following are a collection of observations that show how EBBS's correctness mechanism makes progress towards automatic online procedural learning:

- Several of our evaluation agents that limit their learning to only certain subgoals with classical chunking *can now learn correct rules from all subgoals.* (With classic chunking, these agents learn incorrect rules in the previously disabled subgoals.)
  - BlocksWorld Hierarchical Look-ahead
  - BlocksWorld Look-ahead State Evaluation
  - BlocksWorld Look-ahead
- If the OPL algorithm does not include operator selection knowledge in the rules that it learns, the following agents learn incorrect rules that cause the following problems:
  - BlocksWorld Look-ahead, Water-Jug Look-ahead and Mouse and Cats Planning agents will *sometimes* fail to complete their task.
  - BlocksWorld Operator Subgoaling and BlocksWorld Operator Subgoaling Reinforcement-Learning agents are always unable to complete their task.
  - BlocksWorld Hierarchical Look-ahead will sometimes fail to find a solution, but only in classic chunking mode.
  - Mouse and Cats Look-ahead Reinforcement-Learning agent, an agent we have not discussed, will learn rules that do not converge and improve performance.

7.6.5 Conclusions on our Evaluation of Correctness

In this section, we attempted to answer the question: does the EBBS approach learns correct rules in situations that the current state of the art online procedural learning system cannot? If we review the specific situations listed in Figure 7.52, we see that the answer is yes. We have shown evidence that all of the correctness issues that occur when summarizing behavior traces that do not explain all dependencies have been remedied, except one, C1-4, behavior traces that are dependent on implicit knowledge, which is the one correctness issue that has no clear solution. We have also shown evidence that two out of three of the correctness issues that occur when summarizing behavior traces with reasoning that is not expressible in a single rule have been remedied. A detector for the issue that was not addressed, C2-1, behavior traces with reasoning that conflates a disjunction of contexts, exists but has technical issues. Finally, all of the correctness issues related to unreliable knowledge are detectable, which allows EBBS to learn a justification instead of a rule when it detect that a trace is dependent on unreliable knowledge. This is a viable way to reduce the effect of these three correctness issues because justifications are essentially immune to the problems posed by unreliable knowledge. In summary, while some correctness issues will always exist, most notably behavior traces dependent on implicit knowledge, our evaluation has shown that EBBS eliminates many of the most common sources of correctness issues that existed in classical chunking and detects the ones that it cannot currently ameliorate.



Classic Chunking	EBBS
C1. Behavior Trace Does Not Explain All Dependencies	**C1. Behavior Trace Does Not Explain All Dependencies**
🟢 C1-1. Dependent on dynamically-generated knowledge (meta-knowledge)	🟢 C1-1. Dependent on dynamically-generated knowledge (meta-knowledge)
🔴 C1-1. Dependent on dynamically-generated knowledge (dynamic rule)	🟢 C1-1. Dependent on dynamically-generated knowledge (dynamic rule)
🟢 C1-1. Dependent on dynamically-generated knowledge (LTM recall)	🟢 C1-1. Dependent on dynamically-generated knowledge (LTM recall)
🔴 C1-2. Dependent on operator selection knowledge	🟢 C1-2. Dependent on operator selection knowledge
🔵 C1-3. Dependent on promoted knowledge	🟢 C1-3. Dependent on promoted knowledge
🔵 C1-4. Dependent on implicit knowledge	🔵 C1-4. Dependent on implicit knowledge
🔴 C1-5. Dependent on knowledge from subgoals that do not explain results	🟢 C1-5. Dependent on knowledge from subgoals that do not explain results
🔴 C1-6. Dependent on knowledge retrieved from LTM multiple times	🟢 C1-6. Dependent on knowledge retrieved from LTM multiple times
C2. Reasoning That Is Not Expressible in a Single Rule	**C2. Reasoning That Is Not Expressible in a Single Rule**
⚪ C2-1. Requires that a calculation meets a constraint	🟢 C2-1. Requires that a calculation meets a constraint
🔴 C2-2. Conflates a disjunction of contexts	🔵 C2-2. Conflates a disjunction of contexts
⚪ C2-3. Generates knowledge structures linked to long-term memory structures	🟢 C2-3. Generates knowledge structures linked to long-term memory structures
C3. Reasoning That Is Not Reliably Repeatable	**C3. Reasoning That Is Not Reliably Repeatable**
🔴 C3-1. Dependent on an operator selected probabilistically	🔵 C3-1. Dependent on an operator selected probabilistically
🔴 C3-2. Dependent on knowledge recalled from LTM using opaque mechanism	🔵 C3-2. Dependent on knowledge recalled from LTM using opaque mechanism
🔴 C3-3. Dependent on knowledge recalled from a changing LTM	🔵 C3-3. Dependent on knowledge recalled from a changing LTM

🟢 Pass ⚪ Not applicable
🔴 Fail 🔵 Fail but can detect and block

Figure 7.52: The results of regression tests with classic chunking and EBBS.

7.7 Evaluation Conclusions

In this section, we summarize our evaluation of whether EBBS improves Soar's online procedural learning against our desiderata by answering the following four questions:

- **E1. Task Performance**: Does the EBBS approach improve task performance as much or more than the current state of the art online procedural learning system?

  Our answer is yes. Our experiments have shown strong evidence that EBBS improves task performances as much as classical chunking, and in some cases more. In domains or tasks where learning rate is important, the improvements can be dramatic.

- **E2. Generality**: Does the EBBS approach learn more optimally general rules than the current state of the art online procedural learning system?

  Our answer is yes. Our experiments have shown strong evidence that EBBS captures more of the generality of the underlying rules. In many agents, EBBS summarizes an agent's reasoning in fewer rules than classical chunking, and those rules prevent more impasses and find more opportunities to fire than the rules of classical chunking. We have also shown evidence that the rules that EBBS learns are indeed reasoning generally over knowledge types that chunks cannot reason over, which is the key reason that classic chunking learns overly specialized rules.

- **E3. Computational Costs**: Does the computational overhead added by the EBBS approach allow an agent to maintain reactivity?

  Our answer is yes. While expensive chunks do still exist and are likely to be more common under EBBS, we found that there were no cases in our evaluation agents where they obviate the total speed-up provided by online procedural learning. More importantly, our experiments have shown strong evidence that the additional costs of EBBS do not prevent an agent from maintaining reactivity. Both the average and maximum decision cycle times of all of our evaluation agents are orders of magnitude below problematic values.

- **E4. Correctness**: Does the EBBS approach learn correct rules in situations that the current state of the art online procedural learning system cannot?

  Our answer is yes. Our experiments have shown that EBBS eliminates many of the most common sources of correctness issues that existed in classical chunking and detects the ones that it cannot.

CHAPTER 8

Conclusion

In this thesis, we have made the following contributions:

1. We explicated the underlying deficiencies in the original implementations of chunking that arose from many of its core assumptions. We explained the key issue behind why chunking does not always learn optimally general rules and provided a detailed description of 12 different correctness issues.

2. We provided four architecture-agnostic necessary conditions for correct behavior summarization when performing online procedural learning and showed how they can be used to organize the varied correctness issues of an online procedural learning algorithm into a taxonomy whose categories suggest mitigation strategies.

3. We defined four desiderata for an ideal online procedural learning algorithm that are architecture-agnostic and can be used to evaluate an online procedural learning system.

4. We presented five architecture-agnostic strategies that can be used to improve dynamic procedural knowledge learning. We demonstrated this approach by applying three strategies to chunking's generality issues and four strategies to its correctness issues.

5. We developed mechanisms that implement our five strategies and used them as the basis for a completely new implementation of dependency-based procedural learning called explanation-based behavior summarization (EBBS). Our implementation can continually learn general and correct procedural knowledge while maintaining the reactivity that real-world agents require. This implementation is robust to agent design and can capture the generality of problem-solving that the current state of the art online procedural learning system, chunking, cannot.

6. We detailed how this system captures previously ignored information and uses a novel unification algorithm called Distributed Identity Graph Unification (DIGU) that is informed by

insights we've had on the tight relationship between generality and correctness in online procedural learning systems within cognitive architectures. We showed how DIGU can capture more of the generality in the reasoning being summarized, interfaces with the various mechanisms that improve correctness by using the formalism of object "identity," and is uniquely designed to handle the computational demands that arise from online procedural learning in a cognitive architecture.

7. We described the algorithmic mechanisms that remedy or detect each of the 12 correctness issues described in our chunking analysis.

8. Finally, we evaluated our implementation and showed that our system learns more optimally general rules that capture reasoning that previously could not be captured, effect correct behavior and improve agent performance without sacrificing agent reactivity. We did this by gathering data from several experiments that used 14 agents across 8 different tasks.

One of the primary motivations for this work was to give Soar the capabilities needed to pursue research on higher level procedural learning approaches. So, to conclude this thesis, we examine whether the EBBS approach improves online procedural learning enough so that it can be used in research projects in ways that could not be achieved with the current state of the art online procedural learning system, chunking. In Section 3.7, we listed the various problems that ROSIE, Soar's interactive task learning agent, faced while trying to use chunking as a basis for its level-2 learning:

1. ITL agent cannot tolerate undergenerality. Other agents can compensate for undergeneral rules with extra training, i.e. they learn multiple rules that are equivalent to an optimally general rule. Agents that learn by instruction cannot repeatedly ask a human the same question. So, these agents need to learn optimally general rules from a single instruction (Laird et al., 2017), a capability chunking cannot currently provide.

2. ITL agents utilize many of the types of reasoning that leads chunking to learn incorrect rules. In terms of our correctness taxonomy, our current ITL agents utilize nearly every type of problematic reasoning we have described.

3. ITL agents must learn rules that encode the knowledge acquired from the instruction without being directly dependent on a declarative representation of the instructor's utterance. (A rule that was dependent on the words of the instructor would have little utility since those instructions are only available during the interaction with the instructor.)

From the evaluation in Chapter 7, we can now see that we have completely addressed (1) and made significant on progress on (2).

The third item refers to a key capability that level 2 learning often needs from a level-1 procedural learning system: the ability to ignore aspects of an agent's reasoning when learning rules. This process, which we call context-specific decontextualization, allows an agent to force the procedural learning algorithm to ignore targeted aspects of reasoning in a subgoal. Previous Soar systems that learned by instructions would force learned knowledge to be independent of declarative instructions by putting reasoning that is dependent on instructions in rules that create operator selection knowledge, which the agent designer knew chunking would ignore (Huffman and Laird, 1995), thus producing rules that do not rely on the original instructions. Current systems use semantic memory retrievals to achieve the same thing, which is why the correctness issues related to semantic memory were so debilitating for those projects. Our instance-based semantic memory system eliminated those issues, which means that we've cleared the last roadblock that ITL agents had using Soar's automatic level-1 procedural learning as a basis for its level-2 learning by instruction.

Ultimately, the best evidence is that multiple research systems have already used EBBS in their research, often as a basis for level 2 learning. These projects, which investigate natural language comprehension (Lindes, 2022), game learning (Kirk et al., 2016), interactive task learning (Lindes et al., 2017; Kirk and Laird, 2016; Mininger and Laird, 2016), and a PRIMS model of human procedural learning (Stearns et al., 2017), have all used various implementations of our explanation-based behavior summarization system. For the game-learning and interactive task learning work in particular, the capabilities of EBBS were a fundamental and very necessary component for all of their learning results.

APPENDIX A

Soar Terminology Primer

- **Symbols**: Signifiers that are the atomic units of Soar's representation:
  - **Constants**: There are three types of constants: strings, floating point numbers and integers. Examples are *blue*, *2.3* and *33*.
  - **Soar Identifiers(SI)**: These are special symbols created at run-time to group a set of features together to correspond to an abstraction or object. Soar identifiers are transient and not guaranteed to be the same across runs. They take the form of a letter followed by a number, for example *C1*. We will use the abbreviations **SI** to avoid confusion with the identifier element of other Soar data structures.
  - **Variables**: These symbols are only used in rule conditions and can match any symbol type. They are indicated by angle brackets, for example <*variable*>.
- **Working Memory**: A set of symbol triples that represents an agent's current knowledge of the world and its current state in problem solving. Soar's WM is a directed graph and is also referred to as Soar's *short-term memory*.
- **Element**: Because Soar's working memory is a directed graph, many of its data structures are triples. Regardless of what type of triple it is, *they all have an identifier, attribute and value elements*.
- **Working Memory Element (WME)**: A tuple of three *symbols*. A WME is equivalent to a fact in a predicate logic formalism.
  - The identifier element must be always be an SI.
  - Attributes can be any constant type.
  - The value element can be either a Soar identifier or a constant.
- **Condition**: A tuple of three *tests*. Each test is a symbol along with an optional list of constraint tests.

  - The identifier element must be always be a variable.
  - The attributes and value elements can be any constant type *or a variable*.
  - Note that a condition can be marked as a *negative condition* or a *conjunction of negative conditions* if it tests for the absence of a feature or set of features.

- **Top goal**: A Soar identifier that is the root node of working memory and represents the agent's top-level problem space.

- **Subgoal**: A Soar identifier that is the root node of a subservient problem space's local working memory. This is created when an agent impasses.

- **Superstate Edge**: When an agent impasses, Soar connects a subgoal's local memory to its supergoal's local memory with a superstate edge. This aspect of the representation is important to our discussion because it allows the architecture to *trivially determine whether a piece of knowledge is accessible to a particular goal*.

- **Results**: Working memory elements created in a goal by a subservient subgoal. When a rule fires in a subgoal and creates a new WME linked to a supergoal, we say that the rule is returning a result to the supergoal.

- **Supergoal knowledge**: In this discussion, supergoal knowledge are the working memory elements that are connected to at least one of the supergoals. So, saying that a condition tested supergoal knowledge is shorthand for saying that either the condition's identifier is the supergoal symbol or there exists a path of WMEs that links the supergoal symbol to the symbol tested by the identifier element.

- **Local knowledge**: Similarly, local knowledge is the set of working memory elements for which no path of WMEs exists that links each one's identifier to a supergoal symbol.

- **Variablization**: The process of generalizing a rule by replacing tests for specific values with variables.

- **Heuristic-Based Variablization**: A term to describe chunking's variablization mechanism. Heuristic-Based variablization replaces every Soar identifier in a rule with a variable, always assigning each identifier the same variable.

- **Semantic Memory**: This is a representation of an agent's long-term, semantic memory. This knowledge is not directly accessible to an agent. The agent must deliberately request a subset of semantic knowledge deliberately, either through a specific retrieval or a query based on a set of features.

BIBLIOGRAPHY

Anderson, J. R. Skill Acquisition: Compilation of Weak-Method Problem Solutions. *Psychological Review*, 1987. ISSN 0033295X. doi:10.1037/0033-295X.94.2.192. 2

Cassimatis, N. L. *Polyscheme: A Cognitive Architecture for Integrating Multiple Representation and Inference Schemes*. Ph.D. thesis, Massachusetts Institute of Technology, 2002. 3

Dejong, G. Explanation-Based Learning. In Dejong, G., editor, *Computing Handbook, Third Edition: Computer Science and Software Engineering*, pages 37.1–37.19. 2014. ISBN 9781439898536. 37

Dejong, G. and Mooney, R. Explanation-Based Learning : An Alternative View. *Machine Learning*, 1(1983):145–176, 1986. ISSN 15730565. doi:10.1023/A:1022898111663. 12, 59

Doorenbos, R. B. Production Matching for Large Learning Systems. *Information Sciences*, 9(4):208, 1995. ISSN 15249050. doi:10.1.1.83.4551. 13, 152

Gobet, F. and Lane, P. C. R. Constructing a Standard Model : Lessons from CHREST. In *AAAI 2017 Fall Symposium*, pages 338–343. 2017. 4

Gratch, J. M. and Gerald, F. Utility Generalization and Composability Problems in EBL. Technical report, Illinois Univ., Urbana. Dept. of Computer Science, 1991. 27

Gunetti, P. and Thompson, H. A soar-based planning agent for gas-turbine engine control and health management. In *IFAC Proceedings Volumes (IFAC-PapersOnline)*, volume 17, pages 2200–2205. 2008. ISBN 9783902661005. doi:10.3182/20080706-5-KR-1001.1012. 2

Huffman, S. B. and Laird, J. E. Flexibly Instructable Agents. *Journal Of Artificial Intelligence Research*, 1995. doi:10.1613/jair.150. 38, 187

Keller, R. M. Defining operationality for explanation-based learning. *Artificial Intelligence*, 35(2):227–241, 1988. ISSN 00043702. doi:10.1016/0004-3702(88)90013-6. 27

Kim, J. and Rosenbloom, P. S. Bounding the cost of learned rules. *Artificial Intelligence*, 120(1):43–80, 2000. ISSN 00043702. doi:10.1016/S0004-3702(00)00025-4. 15

Kirk, J., Mininger, A. and Laird, J. Learning task goals interactively with visual demonstrations. In *Biologically Inspired Cognitive Architectures*, volume 18, pages 1–8. 2016. doi:10.1016/j.bica.2016.08.001. 37, 187

Kirk, J. R. and Laird, J. E. Interactive Task Learning for Simple Games. *Advances in Cognitive Systems*, 3:13–30, 2014. 5, 181

Kirk, J. R. and Laird, J. E. Learning General and Efficient Representations of Novel Games Through Interactive Instruction. *Advances in Cognitive Systems*, 4:1–14, 2016. 5, 37, 187

Kotseruba, I. and Tsotsos, J. K. A Review of 40 Years of Cognitive Architecture Research: Core Cognitive Abilities and Practical Applications. *CoRR*, 2016. 5, 17

Laird, J. *The Soar Cognitive Architecture*. 2012. 17, 18, 24

Laird, J. and Mohan, S. Learning Fast and Slow: Levels of Learning in General Autonomous Intelligent Agents. In *AAAI 2018, Senior Track, New Orleans, LA*, Ebbinghaus 1885. 2018. 1

Laird, J. E. *Universal Subgoaling*. Ph.D. thesis, Carnegie Mellon University, 1983. 18

Laird, J. E., Derbinsky, N. and Tinkerhess, M. Online Determination of Value-Function Structure and Action-value Estimates for Reinforcement Learning in a Cognitive Architecture. In *Advances in Cognitive Systems*, volume 2, pages 221–238. 2012a. 25

Laird, J. E., Gluck, K., Anderson, J., Forbus, K. D., Jenkins, O. C., Lebiere, C., Salvucci, D., Scheutz, M., Thomaz, A., Trafton, G., Wray, R. E., Mohan, S. and Kirk, J. R. Interactive Task Learning. *IEEE Intelligent Systems*, 2017. ISSN 15411672. doi:10.1109/MIS.2017.3121552. 2, 38, 186

Laird, J. E., Newell, A. and Rosenbloom, P. S. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987. ISSN 00043702. doi:10.1016/0004-3702(87)90050-6. 4

Laird, J. E. and Rosenbloom, P. Evolution of the SOAR Architecture. In Laird, J. E. and Rosenbloom, P., editors, *Mind Matters: A Tribute to Allen Newell*, pages 1–50. 1996. 4

Laird, J. E. and Rosenbloom, P. S. Integrating execution, planning, and learning in Soar for external environments. *AAAI 1990*, pages 1022–1029, 1990. 1

Laird, J. E., Rosenbloom, P. S. and Newell, A. Chunking in Soar: The Anatomy of a General Learning Mechanism. *Machine Learning*, 1(1):11–46, 1986. ISSN 15730565. doi:10.1023/A:1022639103969. 2

Laird, J. J., Kinkade, K., Mohan, S. and Xu, J. Cognitive Robotics Using the Soar Cognitive Architecture. In *AAAI 2012 8th International Workshop on Cognitive Robotics*, pages 46–54. 2012b. ISBN 9781577355717. 15

Lake, B. M., Ullman, T. D., Tenenbaum, J. B. and Gershman, S. J. Building Machines That Learn and Think Like People. *Behavioral and Brain Sciences*, abs/1604.0:1–101, 2016. ISSN 14691825. doi:10.1017/S0140525X16001837. 5

Lindes, P. *Constructing Meaning, Piece by Piece: A Computational Cognitive Model of Human Sentence Comprehension*. Ph.D. thesis, University of Michigan, 2022. doi:10.7302/4697. 187

Lindes, P., Mininger, A., Kirk, J. R. and Laird, J. E. Grounding Language for Interactive Task Learning. *ACL WS Robot-NLP*, 2017. 29, 37, 187

Miller, C. S. and Laird, J. E. Accounting for Graded Performance Within a Discrete Search Framework. *Cognitive Science*, 20(4):499–537, 1996. ISSN 03640213. doi:10.1016/S0364-0213(99)80013-5. 24

Miller, G. A. The Magical Number Seven. *Psychological Review*, 63:81–97, 1956. ISSN 0033-295X. doi:10.1037/h0043158. 28

Mininger, A. and Laird, J. Interactively Learning Strategies for Handling References to Unseen or Unknown Objects. *Advances in Cognitive Systems*, 5, 2016. 5, 37, 181, 187

Mininger, A. and Laird, J. E. Interactively Learning a Blend of Goal-Based and Procedural Tasks. In *AAAI 2018*. 2018. 5

Mitchell, T. M. Learning and Problem Solving. In *IJCAI*, volume 8, pages 1139–1151. 1983. 27

Mohan, S. and Laird, J. E. Learning Goal-Oriented Hierarchical Tasks from Situated Interactive Instruction. In *AAAI 2014*, pages 387–394. 2014. ISBN 9781577356776. 1, 3

Mooney, R. *A General EBL Mechanism and its Application to Narrative Understanding*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1988. 27

Mooney, R. The effect of rule use on the utility of explanation-based learning. In *AAAI 1989*, pages 725–730. 1989. 13, 54, 153

Mooney, R. J., Bennett, S. W. and Urbana, A. A Domain Independent Explanation-Based Generalizer. In *AAAI 1986*, pages 551–555. 1986. 15, 28, 54

Newell, A. Reasoning, Problem Solving and Decision Processes : The Problem space as a Fundamental Category. In Nickerson, R., editor, *Attention and Performance (Volume VIII)*. Hillsdale, NJ, 1980. ISBN 0-262-68071-8. 17

Newell, A. Unified theories of cognition and the role of Soar. In Newell, A., editor, *Soar A cognitive architecture in perspective A tribute to Allen Newell Studies in cognitive systems Vol 10*, pages 25–79 ST – Unified theories of cognition and the. 1992. 4

Newell, A. *Unified Theories of Cognition*. 1994. 15

Newell, A. and Rosenbloom, P. S. Mechanisms of skill acquisition and the law of practice. In Newell, A. and Rosenbloom, P. S., editors, *Cognitive skills and their Acquisition*, volume 6, pages 1–55. 1981. ISBN 0898590930. doi:10.1.1.910.5264. 28

Rosenbloom, P. S., Laird, J. and Newell, A. *The Soar Papers: Research on Integrated Intelligence*. August 2015. 1993. 23

Rosenbloom, P. S. and Laird, J. E. Mapping EBG onto Soar. In *AAAI 1986*, pages 561–567. 1986. 32

Russell, S. and Norvig, P. *Artificial Intelligence A Modern Approach Third Edition*. 2010. ISBN 9780136042594. doi:10.1017/S0269888900007724. 122

Stearns, B., Laird, J. and Assanie, M. Applying Primitive Elements Theory for Procedural Transfer in Soar. In *Proceedings of the 15th International Conference on Cognitive Modeling*. 2017. 88, 181, 187

Steier, D. M., Laird, J. E., Newell, A., Rosenbloom, P. S., Flynn, R. A., Golding, A., Polk, T. A., Shivers, O. G., Unruh, A. and Yost, G. R. Varieties of Learning in SOAR: 1987. In *AIP-6 Technical Report*. 1987. ISBN 978-0-934613-41-5. doi:http://dx.doi.org/10.1016/B978-0-934613-41-5.50034-9. 23

Taatgen, N. A. The nature and transfer of cognitive skills. *Psychological review*, 120(3):439–471, 2013. ISSN 19391471. doi:10.1037/a0033138. 4

Taatgen, N. A. and Lee, F. J. Production compilation: a simple mechanism to model complex skill acquisition. *Human factors*, 45(1):61–76, 2003. ISSN 0018-7208. doi:10.1518/hfes.45.1.61.27224. 3

Tambe, M. and Newell, A. Some Chunks Are Expensive. In Laird, J., editor, *Machine Learning Proceedings 1988*, pages 451–458. Morgan Kaufmann, San Francisco (CA), 1988. ISBN 978-0-934613-64-4. doi:https://doi.org/10.1016/B978-0-934613-64-4.50051-7. 152, 155

Waldinger, R. Achieving Several Goals Simultaneously. *Machine Intelligence 8*, 8(Technical Note 107):94–136, 1977. 28, 54

Wray, R. E. and Laird, J. E. An Architectural Approach to Ensuring Consistency in Hierarchical Execution. *Journal of Artificial Intelligence Research*, 19:355–398, 2003. ISSN 10769757. doi:10.1613/jair.1142. 9, 30, 47, 105

Wray, R. E., Laird, J. E. and Jones, R. M. Compilation of non-contemporaneous constraints. In *AAAI 1996*. 1996. 46, 105

Zerr, F. and Ganascia, J. G. Integrating an Explanation-Based Learning Mechanism into a General Problem-Solver. In *EWSL'91 Proceedings of the 5th European Conference on European Working Session on Learning*, pages 62–80. 1991. 32, 40

Zhong, S., Ma, H., Zhou, L., Wang, X., Ma, S. and Jia, N. Guidance compliance behavior on VMS based on SOAR cognitive architecture. *Mathematical Problems in Engineering*, 2012, 2012. ISSN 1024123X. doi:10.1155/2012/530561. 1, 26