Extensions to a Unified Theory of the Cognitive Architecture

by

Nishant Trivedi

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2011 by the
Graduate Supervisory Committee:

Patrick Langley, Chair
Kurt VanLehn
Subbarao Kambhampati

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

Building computational models of human problem solving has been a long-standing goal in Artificial Intelligence research. The theories of cognitive architectures addressed this issue by embedding models of problem solving within them. This thesis presents an extended account of human problem solving and describes its implementation within one such theory of cognitive architecture—ICARUS. The document begins by reviewing the standard theory of problem solving, along with how previous versions of ICARUS have incorporated and expanded on it. Next it discusses some limitations of the existing mechanism and proposes four extensions that eliminate these limitations, elaborate the framework along interesting dimensions, and bring it into closer alignment with human problem-solving abilities. After this, it presents evaluations on four domains that establish the benefits of these extensions. The results demonstrate the system's ability to solve problems in various domains and its generality. In closing, it outlines related work and notes promising directions for additional research.

To Dr. Isaac Asimov whose brilliant and imaginative works of fiction picked my interest as a child and continue to inspire me.

ACKNOWLEDGEMENTS

I am thankful to Dr. Pat Langley for his continued guidance and support during my association with him as a Master's student. I learned a lot of valuable lessons during this time that will be helpful to me in my professional pursuits in the future. I would like to extend my appreciation to the members of my committee, Dr. Kurt VanLehn and Dr. Subbarao Kambhampati for their feedback and comments on my thesis. I am grateful to Mr. Glenn Iba for his help in the overall design of the ICARUS architecture and his implementation of supporting modules as well as for lending his expert advice in matters related to programming in LISP. I would like to thank Mr. Ankur Sharma for the interesting discussions and innovative ideas that we shared. I owe tons of gratitude to my parents who have always believed in me and extended their unquestioning support in all of my endeavors. Finally I would like to thank Mr. Gautam Singh, Mr. Arvind Aylliath, Ms. Trupti Nair, Mr. Pritam Gundecha, Mr. Niranjan Kulkarni, Mr. Reiley Jeyapaul, Mr. Vinay Hanumaiah and all my friends at Arizona State University and elsewhere for sharing the lighter side of life with me and keeping the journey interesting.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS iv

LIST OF TABLES vi

LIST OF FIGURES vii

1 Introduction 1

2 The ICARUS Architecture 5

2.1 Knowledge Representation 6

2.2 Conceptual Inference 9

2.3 Skill Execution 10

2.4 Problem Solving 12

2.5 Limitations of Problem Solving in ICARUS 14

3 Extensions to the Problem-Solving Mechanism 16

3.1 Problem-Based Organization of Goals 16

3.2 Alternate Problem Formulations 17

3.3 Improved Search Control 18

3.3.1 Constraints 18

3.3.2 Heuristics 20

3.4 Chaining Over Negated Goals 20

3.5 Representational Extensions 21

3.6 Summary Remarks 22

4 An Extended Problem-Solving Mechanism 23

4.1 Selecting Bindings 23

4.2 Selecting a Focus Goal 26

4.3 Selecting a Skill 26

4.4 Execution and Updating the Environmental State 27

4.5 Backtracking and Failure Recovery 29

Chapter Page

5 Experimental Evaluations 31

5.1 Problem Solving in Tower of Hanoi 31

5.2 Testing System Scalability 32

5.3 Demonstrating the Benefits of Constraints 34

5.4 Demonstrating the Benefits of Heuristics 37

5.5 Summary Remarks 40

6 Related Research 42

7 Directions for Future Work 44

8 Conclusion 46

BIBLIOGRAPHY 47

APPENDIX 49

A LISP Code 49

LIST OF TABLES

Table		Page
2.1	Sample ICARUS concepts in the Blocks World domain	8
2.2	Sample ICARUS skills in the Blocks World domain	8
2.3	Sample percepts from the Blocks World domain	9
2.4	Sample beliefs from the Blocks World domain	10
3.1	Sample Constraints from Logistics and Blocks World Domains	19
3.2	Sample Problem from the Blocks World Domain	22
3.3	Sample Intention from the Blocks World Domain	22
4.1	Pseudocode for the Extended Problem Solver	24
5.1	Tower of Hanoi Operator	32

LIST OF FIGURES

Figure		Page
2.1	An Overview of ICARUS	6
2.2	Classification of ICARUS Memories	7
2.3	Bottom-up Belief Inference	10
2.4	Top-down Skill Execution	11
5.1	System Runtime in Depots Domain as a function of Problem Complexity	34
5.2	System Runtime in Logistics Domain as a function of Problem Complexity	36
5.3	Number of Nodes Explored in Logistics Domain	37
5.4	Number of Nodes Rejected in Logistics Domain	38
5.5	System Runtime in Gripper Domain as a function of Problem Complexity	40

Chapter 1

Introduction

Building computational systems that model human cognition has been one of the long-standing goals of the AI research community. Psychological studies of human cognition have shown that it involves interactions among a variety of processes, each related to a distinct mental function. One capability that differentiates humans from other animals is the ability to solve novel problems in unfamiliar situations. Early research on this topic introduced representational formalisms, performance processes, and learning mechanisms that continue to play a role in current computational models of human cognition. Moreover, models developed in this tradition have been precise about both the structure and processes that underlie cognition. In this thesis, we describe one such framework for modeling intelligent behavior and propose extensions to its account of problem solving.

The first complete theory of problem solving outlined by Newell, Shaw, and Simon [18], made a number of claims about human cognition. The most basic was that problem solving involves processing of list structures, which is closely related to the *physical symbol system* hypothesis [21]. Another important claim, the *problem space hypothesis*, stated that problem solving involves search through a space of problem states generated by applicable operators. In particular, Newell et al. claimed that much of human problem solving uses a class of search methods known as means-ends analysis [20]. In this type of search, a problem solver first determines the differences between the goal state and the current state. Next it selects one of these differences and an operator that would remove it. Finally, it either applies the operator to bring about the change or creates a subproblem that would transform the current state into one where the selected operator is applicable. As the problem solver gains experience, it acquires domain-specific knowledge that lets it work forward from the current state,

rather than chaining backward, to achieve the goal with little or no search [15].

More recently, Langley and Rogers [14] noted some limitations of the standard theory. In particular, they observed that problem solving:

- occurs in the context of a physical environment.
- abstracts away from physical details and returns only to implement the solution.
- is not a solely mental activity but interleaves reasoning with execution.
- can lead to dead ends that requires restarting on the problem
- can trigger learning that transforms problem solving into informed skill execution.

Langley and Rogers addressed the limitations of the standard theory by embedding this extended acount of problem solving into a theory of cognitive architecture.

A cognitive architecture, by definition [13, 17], provides both a theoretical framework for modeling human cognition and software to support the creation of intelligent agents. In particular, it specifies a unified theory of mental phenomena that remains constant across domains and over time. Thus a cognitive architecture includes structures and processes that underlie the workings of the mind. This includes short-term and long-term memories that store goals, beliefs, and knowledge that an agent utilizes. They also specify the representation of this content and its organization in memory. Finally, cognitive architectures describe the processes that operate on these elements including performance mechanisms that use the content and learning mechanisms that alter it.

There has been substantial research on cognitive architectures resulting into a variety of architectures that differ in their assumptions about the representation, utiliza-

tion, and acquisition of knowledge. However, most cognitive architectures share these basic claims:

- Short-term memories contain dynamic information and are distinct from long-term memories, which store more stable content. Short-term memories are characterized by low capacity, fast access, and rapid storage. In contrast, long-term memories are associative with high capacity, rapid retrieval, and slow storage.
- Both short-term and long-term memories contain elements that are modular in that they can be composed dynamically during performance and learning.
- The memory elements are represented as symbolic list structures and the ones in long-term memory are accessed by matching their patterns against elements in short-term memory.
- Cognitive behavior occurs in discrete cycles that match patterns in long-term memory against short-term elements, then use selected structures to carry out mental or physical actions.
- Learning is incremental and tightly interleaved with performance, with structural learning involving the monotonic addition of new symbolic structures to long-term memory.

Moreover, most research on cognitive architectures focuses on creating intelligent agents that exist over time, since human cognition operates in this setting.

In this thesis we focus on one such cognitive architecture—ICARUS—and report work on extending its capabilities. In the next chapter, we review the operation of ICARUS and limitations of its problem-solving module. We then outline some extensions that address these limitations and describe their implementation details. Chapter 5 presents some hypothesis about the system's behavior and experiments that support

them. Chapter 6 examines related research in this area and in closing, chapter 7 outlines directions for future work.

Chapter 2

The ICARUS Architecture

As described earlier, cognitive architectures specify a framework to support modeling of intelligent behavior. They make assumptions about representation, inference, execution, problem solving, learning, and other aspects of cognition. ACT-R [1] and Soar [11] are two well-known cognitive architectures. The ICARUS architecture [12] shares important claims with them, including the assumptions that cognition operates in distinct cycles and that it involves symbolic processing. But ICARUS also makes certain distinctive claims, such as:

- Mental structures are grounded in perception and action.
- Concepts and skills encode different aspects of knowledge, and they are stored as distinct but interconnected cognitive structures.
- Each element in a short-term memory has a corresponding generalized structure in some long-term memory and is generated by instantiating the latter.
- The contents of long-term memories are organized in an hierarchical manner that defines complex structures in terms of simpler ones.
- The skill hierarchy is acquired cumulatively, with simpler structures being learned before more complex ones.

Figure 2.1 depicts the overall structure of ICARUS and shows the basic processes that drive it. These are conceptual inference, skill execution, and problem solving, each of which utilize some form of knowledge about the problem domain. In this section, we first describe how this knowledge is encoded and discuss the representational assumptions on which it relies. Next we examine each of the ICARUS processes in turn.

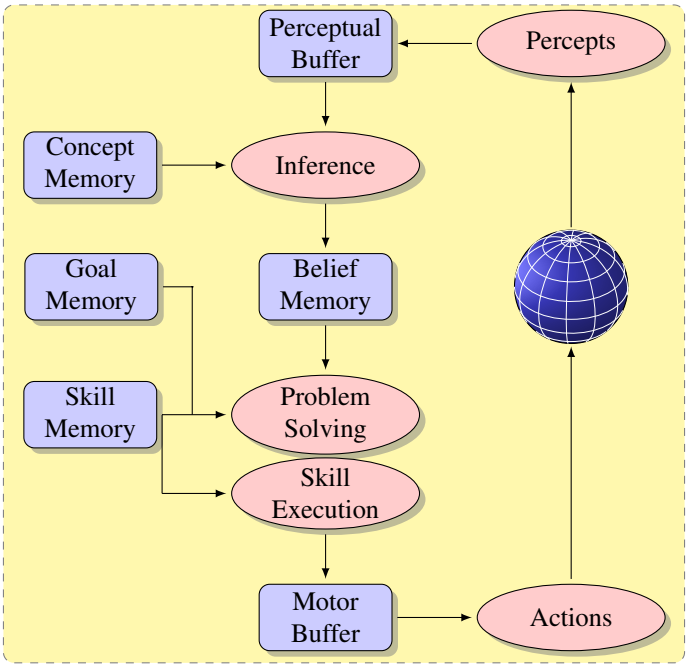

Figure 2.1: An Overview of ICARUS

2.1 Knowledge Representation

To be able to solve problems, an agent must represent knowledge about the problem domain and reason over it. ICARUS represents this kind of knowledge in two distinct forms, concepts and skills. Concepts describe situations that can arise in the problem domain and can be used to reason about the current state of the world. Skills, on the other hand, describe the operations available in the problem domain and can be used to alter the world in ways that achieve goals. Another important distinction made by ICARUS is between long-term and short-term knowledge structures. Concepts and skills are long-term elements that the architecture instantiates with relevent bindings and applies in specific situations. The corresponding short-term knowledge structures are beliefs and intentions that are relevant on a given cognitive cycle. As shown in the figure 2.2, the architecture has a separate memory for each type of structure.

Among the long-term structures, concept definitions are similar to Horn clauses

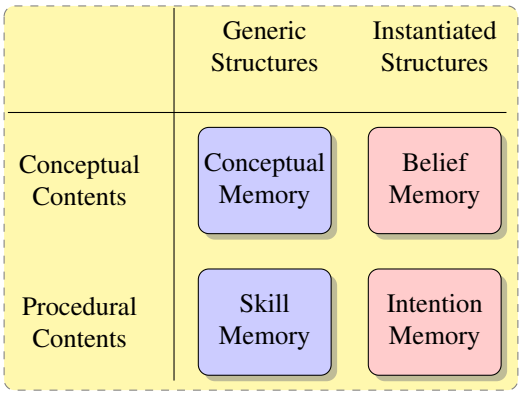

Figure 2.2: Classification of ICARUS Memories

[8] and consist of a head and a body. The head is a logical literal with pattern-match variables and is instantiated when a belief is inferred from a concept. The body consists of conditions that match against percepts or beliefs that refer to other concepts. Concepts are organized in a hierarchy. Ones that refer only to percepts are known as *primitive concepts* and sit at the lowest level of this hierarchy. *Non-primitive concepts*, on the other hand, refer to other conceptual predicates. Table 2.1 shows three sample ICARUS concepts from the *Blocks World* domain. Here the terms preceded by a question mark (?) symbol are pattern-match variables. The first two concepts are primitive since they specify only perceptual matching conditions via the :percepts and :tests fields. However, the last concept refers to other conceptual predicates through the :relations field and hence is an example of a non-primitive concept.

The second type of long-term structure—skills—resemble STRIPS operators [6] and also consist of a head and a body. The head specifies the predicate that a skill achieves and is used to index it. The body of a skill is formed of perceptual conditions, application conditions in form of conceptual predicates, and references to subgoals or actions. As the name suggests, application conditions specify a set of literals that must be true for the skill to be applicable. A *primitive skill* refers to actions that an agent can execute directly to change the world state. In contrast, a non-primitive skill specifies

Table 2.1: Sample ICARUS concepts in the Blocks World domain

```
((holding ?block)
  :percepts ((hand ?hand status ?block) (block ?block)))

((hand-empty)
  :percepts ((hand ?hand status ?status))   :tests ((eq ?status nil)))

((stackable ?block ?to)
  :percepts ((block ?block) (block ?to))
  :relations ((clear ?to) (holding ?block)))
```

subgoals that ICARUS should satisfy in order to achieve the head. These subgoals in turn refer to other skills that should be executed to carry out the parent skill. Thus, skills are organized in a hierarchy that is similar in structure to conceptual memory. Table 2.2 shows two example ICARUS skills, again taken from the *Blocks World* domain. The skill with *(holding ?block)* as its head is primitive, since it specifies actions that can be executed directly in the world, placing it at the lowest level of the skill hierarchy. However, the second is a non-primitive skill, since it specifies two subgoals that refer to other skills in the domain. As with concepts, terms with a question mark (?) symbol denote pattern-match variables.

Table 2.2: Sample ICARUS skills in the Blocks World domain

```
((holding ?block)
  :percepts ((block ?block) (table ?from height ?height))
  :start ((pickupable ?block ?from))
  :actions ((*grasp ?block) (*vertical-move ?block (+ ?height 50))))

((stackable ?block ?to)
  :percepts ((block ?block) (block ?to))
  :start ((hand-empty))
  :subgoals ((clear ?to) (holding ?block)))
```

During its operation, different ICARUS processes instantiate these long-term knowledge structures based on the current state of the world—which is summarized by

the percepts on each cognitive cycle. We now describe the processes that the architecture uses to generate short-term structures.

2.2 Conceptual Inference

The first process invoked by ICARUS is conceptual inference, which is responsible for generating beliefs. The architecture operates in cycles, each of which updates the perceptual buffer that contains descriptions of all visible objects in the environment. These descriptions—called percepts—specify an object's type, a unique identifier, and zero or more attribute-value pairs. As the agent interacts with the environment, its actions may change attributes of some visible objects or it may encounter new objects. Perceptual information reflects these updates in the agent's environment. Table 2.3 shows some example percepts from the Blocks World domain that describe four blocks, a table, and the positions of these objects.

Table 2.3: Sample percepts from the Blocks World domain

```
(block A xpos 0 ypos 2 width 2 height 2)
(block B xpos 0 ypos 4 width 2 height 2)
(block C xpos 0 ypos 6 width 2 height 2)
(block D xpos 0 ypos 8 width 2 height 2)
(hand H1 status empty)
(table T1 xpos 0 ypos 0 width 20 height 2)
```

After ICARUS has updated the perceptual buffer, it invokes an inference module that transforms the perceptual information into a set of beliefs about the current state of the environment. Inference uses concept definitions to infer beliefs and adds them to a short-term memory. Thus beliefs are instances of generic concepts cast as relational literals. Table 2.4 shows some beliefs based on the perceptual information in Table 2.3.

The inference mechanism operates in a bottom-up, data-driven manner. It starts with the updated percepts and matches them against conditions in concept definitions to produce beliefs that are instances of primitive concepts. These beliefs in turn trigger

Table 2.4: Sample beliefs from the Blocks World domain

(clear D)
(four-tower A B C D)
(hand-empty)
(on B A)
(on C B)
(on D C)
(on-table A T1)

inferences of non-primitive concepts and so on. The inference process continues until all the beliefs implied by percepts and concept definitions have been generated. Figure 2.3 shows a schematic representation of this process.

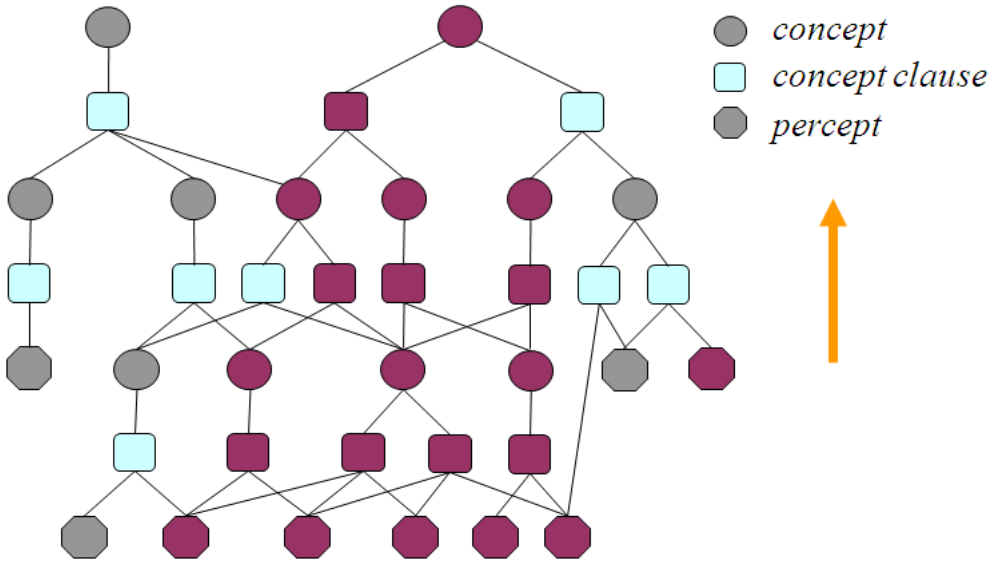

Figure 2.3: Bottom-up Belief Inference

2.3 Skill Execution

Once ICARUS has inferred beliefs about the current state of the world, a skill execution module utilizes this information to carry out actions in the environment. While making decisions, this module refers to its goals, which are instances of known concepts, and skills, which describe how to accomplish these goals. Although ICARUS can have

multiple top-level goals, it can attend to only one goal at a time. Hence the first step in skill execution is to select an unsatisfied top-level goal with the highest priority after which it retrives a skill with a head that matches this goal.

Unlike the inference mechanism, skill execution proceeds in a top-down manner. It finds a chain of skill instances such that the topmost skill indexes the agent's goal. Moreover this chain descends through the skill hierarchy using subskill links, unifying arguments in a consistent manner with those of parents, and ends in a primitive skill. We call such a chain through the skill hierarchy that has a primitive skill as the leaf nodes as a *skill path*. Figure 2.4 presents this process in schematic terms.

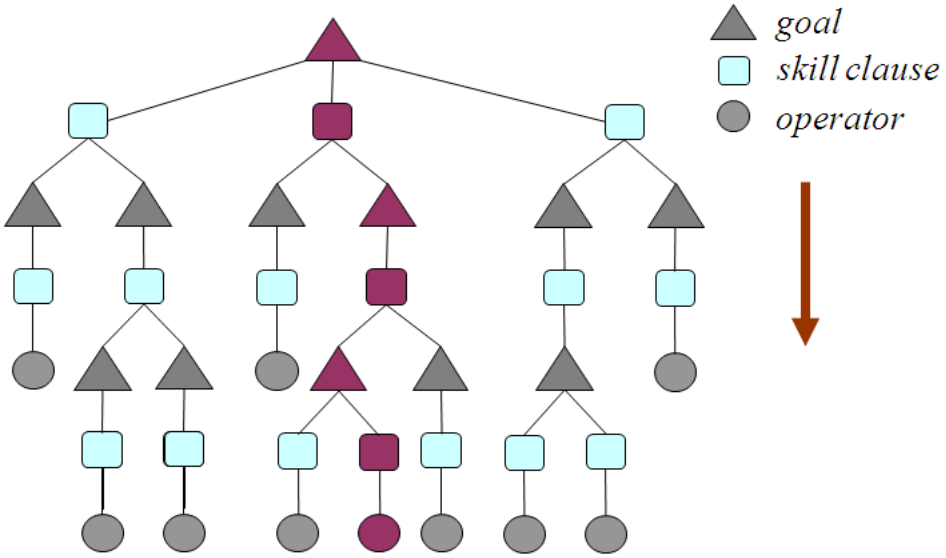

Figure 2.4: Top-down Skill Execution

The execution module only considers skill paths that are applicable. A skill path is applicable if, along the path, no concept instance that corresponds to a goal is satisfied, the conditions of the primitive skill at the leaf are satisfied, and, for each skill in the path not executed in the previous cycle, the conditions are satisfied. The last condition lets ICARUS distinguish between the initiation and continuation of skills that take many cycles to achieve their effects. To this end, ICARUS retains the skill path selected during the previous cycle along with its variable bindings.

ICARUS also includes two preferences that influence the execution on every cycle and serve to provide a balance between reactivity and persistence. The first one applies when the architecture must choose between two or more subskills. It drives ICARUS to select for execution the first subskill for which the corresponding concept instance is not satisfied. This supports reactive control, since it essentially reconsiders subskills that were completed previously but had their effects undone by unexpected events and reexecutes them to rectify the situation. The second one guides ICARUS' choice among two or more applicable skill paths. It prefers paths that share more elements with the start of the skill path executed on the previous cycle. This biases the agent towards continuing with higher-level skills already being executed until the goals associated with them are achieved or the skills becomes inapplicable.

2.4 Problem Solving

As described earlier, the execution module selects and executes skills that achieve the current goal and continue doing so until all top-level goals are satisfied or until it hits an impasse [22]. An impasse occurs when the module fails to find an applicable skill path through the skill hierarchy. This causes ICARUS to invoke its problem-solving module, which utlizes a variant of means-ends analysis [20] to chain backward from the goal. The problem-solving process relies on a structure called the goal stack, which stores subgoals and skills that achieve them. The top-level goal is the lowest element in this stack. Although problem-solving in ICARUS is similar in some ways to means-ends analysis, it differs in two important ways. First, it has a bias towards eager execution so that it carries out in the environment any skill as soon as it becomes applicable. This results in a tight interleaving of problem solving and execution. Second, the problem solver can chain off not only the application conditions of skills but also definitions of concepts whose head matches the current goal.

When invoked, the problem solver pushes the current unsatisfied goal *G* onto

the goal stack. On each cycle it checks if *G* has been achieved, in which case it pops *G* and focuses on *G*'s parent. In case *G* is a top-level goal, the module halts. If *G* is not satisfied, ICARUS retrieves all primitive skills that have *G* as one of their effects. To choose among these candidate skills, the system examines the conditions for each and selects the one with the fewest unsatisfied primitive components in the current enviromental state. In case of ties, it randomly selects one from this set. If all of the selected skill's conditions are met, the architecture executes it until the associated goal is achieved, which is then popped from the goal stack. If the conditions of the selected skill is not satisfied, the system pops it onto the stack, making it the current goal, and the problem solver directs its attention to it. Upon achieving the condition, ICARUS pops the goal stack and since executes the skill. [1]

If the problem solver fails to find any skill clause that achieves *G*, it tries to find a concept definition whose head matches *G* and uses it to generate subgoals. The system then identifies which subgoals are not satisfied, selects one at random and pushes it onto the goal stack. The problem solver then attempts to achieve this goal which causes additional chaining off skill conditions and/or concept definitions. On achieving the subgoal, the architecture pops the goal stack and turns it attention to any other unsatisfied subgoals. Achievement of all subgoals means that *G* is satisfied, so the system pops the stack and focuses on the parent of *G*.

It is evident that the problem-solving module makes decisions about selection of skills and the order in which it should achieve subgoals. There is a possibility that the system will make an incorrect choice so that for a given subgoal it runs out of options. Alternatively ICARUS can reach the maximum depth of the goal stack. When either of these situations occur, ICARUS pops the failed goal, stores it with its parent goal to avoid possible repetition in future, and backtracks to explore other options. Thus,

[1] Danielescu et al. [4] report a modified version of problem solver that can also operate on non-primitive skills with multiple conditions, any number of which can be unsatisfied.

overall the problem-solving process resembles backward-chaining, depth-first search through the problem space. However, it has a low memory load since it retains only the current stack and interleaves problem solving with execution whenever possible.

2.5 Limitations of Problem Solving in ICARUS

In our experience with using ICARUS to create intelligent agents, we noted that, although the architecture incorporates the main claims of the standard theory of problem solving and its extensions, there still remain some limitations. These manifest as difficulties in programming, excessive amounts of search, and complete inability to handle certain problem types. Analysis suggests that both the representational commitments of the system and the design of the component processes are responsible for these drawbacks.

One issue is that, problem solving in ICARUS is driven by individual goal elements. This means that, when a problem solution requires the architecture to satisfy a set of goals, the programmer must define a concept that encapsulates this set under a single predicate which can then be used as the top-level goal. In addition to requiring extra effort for the programmer, the single-goal representation prevents the system from reasoning about goal interactions. Another limitation arises from the way in which ICARUS chooses subgoals during problem solving. Recall that it matches each condition of a skill or concept definition against the beliefs and picks one out of those that are unsatisfied. The conditions are a conjunctive set and depending on the structure of the underlying problem space there can be more than one ways to partially satisfy them. Unfortunately, by matching the conditions individually, the problem solver cannot use this important information.

Another limitation relates to the uninformed choice of subgoals and skills. Since the architecture does not support specification of constraints or control rules [3] that can select, reject, or prioritize subgoals. Whether such constraints are embedded

in problem descriptions or learned from experience, they can play an important role in reducing the amount of search involved in problem solving. We have noted that the ICARUS problem solver carries out much less directed search than humans on the same task. Its reliance on domain-specific knowledge in form of concepts and skills helps when it is available, but for unfamiliar domains it fails to employ domain-independent heuristics that, for humans, informs the choice of skills.

Finally, the current version of the problem solver cannot deal with negated skill conditions with the same generality as it can handle positive conditions. In fact the system can achieve an unsatisfied negated condition only if there is a skill that produces that negation as an effect. In absence of such a skill it cannot chain off the negation of a defined concept whose head matches the condition. In the next section, we propose extensions to ICARUS ' problem-solving mechanism that address these limitations.

Chapter 3

Extensions to the Problem-Solving Mechanism

In the previous chapter we examined the working of ICARUS and described the limitations of its problem-solving module. We now present extensions to the architecture that address these limitations. These introduce sets of goals in form of problems, alternative problem formulations, heuristics and control rules to guide search, and the capability to chain over negated goals. In the rest of the chapter we describe each extension in detail. We conclude the chapter by presenting the new cognitive structures introduced as a result of these changes.

3.1 Problem-Based Organization of Goals

Recall from chapter 2 that ICARUS' goals describe some aspect of a situation that is desired by an agent, and that they are similar to beliefs except that they can have pattern-match variables that must be bound and can appear in a negated form. Also recall that the problem solver treats goals as individual elements. We claim that problem solving in humans is organized around goal sets that describe desirable classes of states for an agent. Traditional AI planners have addressed this issue by specifying a set of conjunctive goals. We have extended the architecture's knowledge representation to include *problems*, which consist of a set of goals that can share variables. For example, $((on\ ?x\ ?y)(on\ ?y\ ?z))$ is a set of goals from the Blocks World domain with two members.

The introduction of problems implies that the architecture, rather than checking for the achievement a single goal, must ensure that it has satisfied all the goals that form a problem. Another important implication of this representation is that chaining backwards off inapplicable skills or concepts must produce subproblems instead of individual subgoals. Thus, as the problem solver progresses towards a solution, it

should form a chain of subproblems linked via skill instances. This chain has the top-level problem at its head and terminates in a subproblem which has all of its members satisfied in the current situation.

3.2 Alternate Problem Formulations

Previously, we saw that ICARUS' problem-solving mechanism randomly matched subgoals against beliefs and pursued the first one that was unsatisfied. However, we have seen that this approach ignores the information about the organization of the underlying problem space. In the new framework, problem solving attempts to transform the current state into one that satisfies the problem and relies on the differences between the two to do so. This makes it clear that there can be more than one way to characterize these differences, each of which corresponds to a distinct problem formulation that partially satisfy the given goal set. This requires the system to choose among different possible formulations of the current problem. An example from the Blocks World domain should clarify this idea. Assume that a problem requires achievement of three conjunctive goals—$\{(on\ ?x\ ?y)\ (on\ ?y\ ?z)\ (on\ ?z\ ?w)\}$—and that in the current belief state $(on\ A\ B)$ and $(on\ B\ C)$ hold true. One possible set of bindings $\{?x \rightarrow A,\ ?y \rightarrow B,\ ?z \rightarrow C\}$ leads to the single unsatisfied goal $(on\ C\ ?w)$. But the other possibility $\{?y \rightarrow A,\ ?z \rightarrow B,\ ?w \rightarrow C\}$ produces a completely different goal $(on\ ?x\ A)$.

We posit that this ability to render alternative problem formulations is an important aspect of human problem solving and justifies the need of an extra level of decision making. We have included this step in the new version of ICARUS in the form of selection of different *bindings* for the pattern-match variables in the problem. Thus, whenever the architecture encounters a new problem or creates a subproblem, it first generates possible bindings by comparing the problem with the belief state and selects one from the available choices. This selection drives the behavior of the agent since it determines which differences it must reduce to arrive at a solution.

3.3 Improved Search Control

In the extended framework, there are three main sources of search during problem solving: the choice of bindings to determine the difference between the current state and the goal state, the order of goal achievement, and the choice among different skills or multiple instances of the same skill that achieve a given goal. As mentioned earlier, in the older version these choices either did not exist or were uninformed, resulting in excessive search. To overcome this, the new problem solver includes two different search control mechanisms that reduce the probability of making the wrong decisions and enable more informed search in the problem space. We now examine each of these mechanisms in turn.

3.3.1 Constraints

The first mechanism introduces the ability to use constraints to guide the architecture's choice of bindings as well as the selection of a single goal from the set of goals. The latter choice arises because ICARUS retains the ability to focus on one goal. We include two types of constraints that operate either on bindings or on goals, using a distinct memory structure to represent each of them. This structure consists of head (an identifier for the constraint), type, and conditions. While the type specifier is used to determine the applicability of a constraint, the conditions specify literals that should be satisfied in order for it to be active. These refer to the agent's goals or to the current belief state with the latter being optional.

Additionally, constraints that apply to goals include directives that specify some action that the architecture should take when they are active. Table 3.1 lists three sample constraints from different problem domains. As we describe later, ICARUS employs each type of constraint in a different manner. For constraints on bindings, it substitutes each possible set into the conditions and rejects the set if they are not satisfied. An

example should clarify this idea. Assume that ICARUS is trying to solve a problem that specifies a single goal $(truck-at\ ?truck\ L03)$ with the restriction that trucks cannot travel across cities. Furthermore, assume the current belief state consists of the beliefs: $\{(truck\ T01)\ (truck\ T02)\ (location\ L01)\ (location\ L02)\ (location\ L03)\ (truck-at\ T01\ L01)\ (truck-at\ T02\ L02)\ (different-city\ L02\ L03)\}$. Then the $(city-reachable)$ constraint in Table 3.1 would direct the system to reject the binding choice $T02$ for the variable $?truck$ since it is not in city of the target location $L03$.

Table 3.1: Sample Constraints from Logistics and Blocks World Domains

```
((city-reachable)
  :type bindings-constriant
  :goal-conditions ((truck-at ?truck ?location))
  :belief-conditions ((truck-at ?truck ?curr-location)
                      (different-city ?location ?curr-location)))

((order-goals)
  :type goal-constriant
  :goal-conditions ((on ?x ?y)(on ?y ?z))
  :delete ((on ?x ?y)))

((clear-block)
  :type goal-constriant
  :goal-conditions ((on ?x ?y))
  :belief-conditions ((on ?w ?x))
  :add ((not (on ?w ?x))))
```

In contrast, the architecture uses constraints on goals to impose an ordering on candidate goals by rejecting some candidates in favor of others or add new goals with higher priority. To give an example, if the problem solver is trying to satisfy a problem with goals $\{(on\ ?x\ ?y)\ (on\ ?y\ ?z)\}$ then the $(order-goals)$ constraint would direct it to achieve the goal $(on\ ?y\ ?z)$ before $(on\ ?x\ ?y)$ which makes sense intuitively. On the other hand if the current belief state described a situation in which some other block is on $?x$, the $(clear-block)$ constraint would add a new high priority goal $(not\ (on\ ?w\ ?x))$, steering ICARUS to satisfy the negation before it considers any of the

problem goals.

3.3.2 Heuristics

The second mechanism equips the problem solver with two domain-independent heuristics that inform the selection of skills. The first prefers skill instances that would achieve more unsatisfied goals in the current problem over the ones that achieve fewer goals. The intuition behind this is to reduce the number of steps required to solve a problem. It also provides a means of masking the primitive skills with fewer effects with higher-level skills that produce more results. To illustrate, say the current goals are $\{(on\ A\ B)\ (on\ B\ C)\}$ and the current beliefs are $\{(on-table\ A)\ (on-table\ B)\ (on-table\ C))\}$, then this heuristic will prefer the $(three-tower\ ?x\ ?y\ ?z)$ skill with effects $\{(on\ ?x\ ?y)\ (on\ ?y\ ?z)\}$ over the $(stack\ ?x\ ?y)$ with the effect $(on\ ?x\ ?y)$.

The other heuristics examines the conditions of the skill instances and select ones with fewer unsatisfied conditions over the ones with more. Intuitively, this tries to reduce the number of steps required to achieve the current goal. An extreme case occurs when all the conditions of a skill instance are satisfied, in which case it becomes immediately applicable. Thus, if the architecture is trying to achieve the goal $(on\ A\ ?x)$ with beliefs specifying a situation in which $\{(on\ B\ C)\ (holding\ A)\}$ hold, this heuristic will prefer the skill instance $(stack\ A\ B)$ over $(stack\ A\ C)$ since in the former instantiation both the conditions $\{(holding\ ?x)\ (not\ (on\ ?any\ ?y)\}$ of the skill $(stack\ ?x\ ?y)$ are satisfied. As we report shortly, the effectiveness of two heuristics varies with the number of primitive and non-primitive skills available to the system.

3.4 Chaining Over Negated Goals

Our final extension addresses ICARUS' inability to chain off a negated condition, which, like chaining off its positive counterparts, involves finding a concept whose head matches the non-negated literal in the condition. But unlike the latter, negation involves univer-

sal quantification and hence the system needs to negate the relations of the matching concept. Recall that conceptual relations are connected via conjunction and negating them produces a disjunctive list in accordance to De Morgan's law. Consequently, it is sufficent to satisfy only one of the negated relations in the resulting list to achieve the parent condition. But this requires special treatment during problem solving.

The new problem-solving module incorporates disjunctive problems, which are created when the architecture chains off the negation of a defined concept. Such sub-problems are composed of disjunctive goals and are labeled accordingly so that they are considered as achieved when any member of the set becomes satisfied. Otherwise, the rest of the system is oblivious to different types of problems and interleaves disjunctive and conjunctive tasks as necessary.

3.5 Representational Extensions

Having described the modifications to ICARUS' problem-solving mechanism, we now examine the cognitive structures introduced by these changes. The first such structure is the one used to represent problems in memory. It consists goals, bindings, focus, and intention fields. It also contains a field to store a list of failure contexts relevant to the problem and to store the parent intention of the problem. Finally it contains a flag that specifies if the problem is disjunctive. Table 3.2 shows a top-level conjunctive problem for a sample task in the Blocks World domain.

We also modified the structure of skills so that the head no longer requires to refer to an instance of a known concept and restricted the use of action field to specify only one action. Moreover, we introduced an explict structure to represent intentions in memory. This inherits the head, conditions, effects, subskills, and action fields from the corresponding skill structure. In addition, it contains: a bindings field that specifies values for some or all of the pattern-match variables in the skill; a problem-parent field that links it back to the problem for which it was generated; and a flag that identifies

Table 3.2: Sample Problem from the Blocks World Domain

```
(problem
  :id 1
  :goals ((on ?x ?y) (on ?y ?z))
  :bindings ((?x . A) (?y . B) (?z . C))
  :focus (on B C)
  :intention (stack B C)
  :intention-parent NIL
  :disjunctive-problem? NIL
  :failure-context NIL)
```

if the intention is currently executing. Table 3.3 shows the instance of a skill from the Blocks World domain.

Table 3.3: Sample Intention from the Blocks World Domain

```
(intention
  :head (stack ?block ?to)
  :bindings ((?block . B)(?to . C))
  :conditions: ((holding B) (clear C))
  :effects: ((on B C) (hand-empty) (not (holding B)) (not (clear C)))
  :subskills: NIL
  :action: (*put-down 'B xpos (+ ypos height))
  :executing? T
  :problem-parent: (problem :id 1 ...))
```

3.6 Summary Remarks

The extensions we have described let ICARUS solve classes of problems that it could not handle earlier. In addition, the new problem solver carries out a selective exploration of the problem space with a significant reduction in the amount of search involved. At the same time, these mechanisms occasionally mislead it down to paths that later fail. Both characteristics are also observed in human problem solving and we claim that our extensions bring the architecture closer to human levels.

Chapter 4

An Extended Problem-Solving Mechanism

Having described the extensions to ICARUS' problem-solving mechanism and the cognitive structures introduced by them, we are now ready to examine their implementation in detail. Before we begin, it is important to note that problem solving in the architecture is tightly interleaved with skill execution and consequently we also needed to make significant changes to its execution module. However, these do not fall within the scope of the work reported in this thesis and we will mention them only in passing.

As described earlier, finding a solution to a problem involves search for a solution state in the problem space. Table 4.1 presents the pseudocode for this process that, like its predecessor, uses a variant of means-ends analysis. As stated in the table, given a problem, the system carries out one of the following four steps on every cycle: select a problem formulation/bindings; choose a focus goal; select a skill; execute a skill and update the problem; or backtrack and recover from failures. In the rest of the chapter we describe each of these steps in detail.

4.1 Selecting Bindings

Recall that ICARUS uses problems to represent a set of goals it wants to achieve. While trying to accomplish a problem, the architecture first compares the problem with the current belief state to determine if the latter satisfies the goal set in a consistent manner. If so, it directs attention to other problems or, having solved all problems, simply halts. However, when a problem is not satisfied, the comparison produces one or more differences in the form of bindings for the pattern-match variables in the goals, each of which corresponds to a distinct way of formulating the problem.

The mechanism responsible for this comparison operates in four steps. First it isolates from the set of goals those that the problem solver cannot accomplish since

Table 4.1: Pseudocode for the Extended Problem Solver

```
Solve(S)
 If there is no active problem P,
    Select P from stack S.
 Else on each cycle,
    If P does not have bindings,
       Find bindings B such that.
       B has not been tried before,
       B satisfies any applicable binding constraint, and
       Substituting B in P does not yield a problem that is similar
          or harder than any parent of P.
       If no such bindings B can be found,
          If P is the top problem, fail and halt.
          Else set active problem to P's parent problem P',
             Fail on intention I of P' and backtrack.
       Else if B satisfies P
          If P is the top problem, report success and halt.
          Else set active problem to P's parent problem P',
             Execute the intention I of P'.
       Else set B as bindings of P.
    If P does not have a focus goal
       Select goal G such that,
       G ∈ P, G is unsatisfied,
       G hasn't been tried before, and
       G satisfies any applicable goal constraint.
       If no such goal G can be found,
          Fail on B and backtrack.
       Else set G as the focus of P.
    If P does not have an intention
       Find skill S such that,
       S achieves G,
       S has not been tried before, and
       S satisfies the skill selection heuristic.
       If no such skill S can be found,
          Find concept C whose head matches G
          If no such concept C can be found,
             Fail on G and backtrack.
          Else create intention I from C and
             Set I as intention of P.
       Else create intention I from S and
          Set I as intention of P.
```

none of the skills directly achieve them and the concepts to which they refer are either primitive or consist of conditions that are unachievable themselves. We refer to such goals as *irreducible differences*. Intuitively, any set of bindings that does not completely satisfy the irreducible differences will fail eventually and we use the differences to eliminate such bindings choices, with an aim to reduce the search in problem solving.

To this end, the system generates a set of intermediate bindings $S = \{I_1, I_2, \ldots, I_n\}$, by computing all possible ways to completely and consistently satisfy every irreducible difference. This process is recursive in nature and ICARUS starts by picking the first difference and determining all beliefs that match against it. Every match provides distinct bindings for the chosen difference and for each such bindings, the architecture recursively computes all possible bindings for the remaining literals. This continues until it has considered all irreducible differences in this fashion.

After this, the mechanism turns to the remaining goals of the problem. It carries out a greedy search through the space of all partial matches for these goals and generates a partial match I_i' corresponding to each intermediate bindings I_i, by substituting I_i into the problem and comparing the elements against the current belief state. More specifically, for each I_i, ICARUS starts with an match set consisting of only the irreducible differences and randomly selects one of the remaining goals that it has not considered before. It then tries to find a belief that matches the goal in a manner that is consistent with I_i. If it is successful in finding such a belief, the system extends I_i by incorporating any new bindings and adds the goal to the match set. Processing all goals in this manner, it produces a partial-match binding I_i'. The system then repeats the whole process for I_j, where $j \neq i$,and continues to do so until it has considered all elements of the set S.

The result is a set $S' = \{I_1', I_2', \ldots, I_n'\}$ of partial-match bindings corresponding to S. The mechanism rejects any I_i' that violates a binding constraint and selects one bindings which, when substituted in the problem, satisfies the maximum number of

goal elements. In case of a tie, it makes a random choice and stores these bindings in the corresponding field of the problem. This choice determines the goals that the problem solver must satisfy and drives its later behavior.

4.2 Selecting a Focus Goal

The next step in the problem-solving process is to reduce the differences between the problem and the current belief state as characterized by the selected bindings. To this end, ICARUS selects one of the unsatisfied goals as its focus and stores it in the problem's *:focus* field. The current implementation randomly picks one unsatisfied goal from the problem goals but, as discussed in the previous chapter, it is possible to specify control rules or constraints that inform this selection. Recall that there are two types of constraints that apply to goals. The first kind specifies an ordering on the goals (based on expected difficulty, the interactions between goals, etc.) and when applicable removes lower-order goals from the choices available to the system.

The second type instructs the architecture to pursue new, high-priority goals that describe a state that falls on every solution path for the given problem. When the constraint ceases to be active once and the system turns its attention to the problem goals. As the problem solver makes progress towards a solution, it eventually selects and achieves each of the problem goals and may also return to any previously satisfied goal in case it was undone due to any actions. As in case of bindings, in event of a tie, ICARUS makes a choice at random.

4.3 Selecting a Skill

After it has selected a focus goal *G*, ICARUS tries to find and instantiate a skill that would achieve it. To this end, it retrieves all skill clauses that have effects which unify with *G*. There can be more than one such skill clause; in fact, the same skill can match more than once if it includes multiple effects having the same predicate as *G*. The

architecture selects one among the choices by employing the skill-selection heuristics described in the previous chapter, again choosing randomly in case of a tie. As noted, the extended problem solver includes two such heuristics which it applies differently based on its configuration. The system instantiates the selected skill with the associated bindings generated by matching *G* to create an intention which represents the system's provisory commitment to apply a partially bound skill and stores this in the problem's *:intention* field.

Upon selecting and storing an intention, ICARUS generates a subproblem from its conditions. When all the goals of such a subproblem are satisfied in the present state, the parent intention becomes applicable, and as we describe later, the architecture initiates its execution in the environment. In contrast, if some goals remain unsatisfied, the system attempts to solve the subproblem using the mechanisms we have already discussed.

Sometimes the problem solver fails to find a skill with effects that match the focus goal. In such a situation, it tries to find a non-primitive concept whose head matches the focus and upon succeeding creates a pseudo-intention from the conditions of the conceptual rule in order to apply it. It treates the pseudo-intention in the same way as an intention based on a skill and proceeds to chain off the former by creating a subproblem from its conditions. However, there is one difference: the satisfaction of all goals for a subproblem created from a pseudo-intention does not lead to execution but instead lets the system infer the head of the rule that generated the intention and thus in turn satisfy the focus.

4.4 Execution and Updating the Environmental State

Recall that problem solving aims at converting the agent's current state into one in which its problem goals are satisfied and, to this end, it selects bindings, focus goals, and intentions. Eventually it selects an intention all of whose conditions are satisfied in

the current state and thus is applicable. ICARUS then invokes its skill-execution module on such an intention. If it is an instance of a primitive skill, the architecture executes the action associated with the intention to change the environment. If not then the system goes down the skill-hierarchy until it finds a primitive skill that it can execute. After finishing this skill, it moves on to other subskills, ascending or descending the skill hierarchy as necessary until all the effects of the parent skill are satisfied. Execution then halts and passes control back to problem solving.

The problem solver selects a skill because it has effects matching the current goal and execution of its corresponding intention to completion achieves that goal. Moreover, it is also possible for a skill to achieve more than one goals. Thus skill execution may lead to new bindings for variables in the satisfied goals. This, along with the fact that goals in a problem can share variables that need to be bound consistently, requires that ICARUS update the bindings of its current problem after each successful execution. Hence when the control is passed back to the problem solver, it resumes from the select bindings step and updates bindings to account for the effects of execution. It then moves on to other goals, selecting and executing skills to achieve them as well as updating bindings.

The interleaving of selection of bindings, goals, and skills with execution can lead to a state where all goals of the current problem P are satisfied. When this happens, ICARUS marks P as solved and directs its attention to P's parent P'. If P was created from conditions of a skill, then its satisfaction leads to the execution of the intention stored in P'. On the other hand, if the source was an inference rule, then accomplishment of P leads to the inference of the rule's head. Both of these satsify a goal in P'. Finally, if P is a top-level problem, the architecture just halts.

4.5 Backtracking and Failure Recovery

As might be expected, problem solving does not always find a solution without facing difficulties. There are three possible ways in which its choices can lead to failures. First, it can realize that the bindings it selected for a subproblem make it equivalent to or more difficult than some problem higher up in the problem chain, in that all the goal elements of the ancestor unify with those of the subproblem. When it detects such a repetition, it abandons the selected bindings and considers alternatives. Second, ICARUS can realize that it is passing through the same environmental states repeatedly. To breakout of such a loop, it abandons the first intention all of whose effects match a repeated state. To support such detection, it stores the environmental states for comparison.

Finally, the architecture abandons any bindings, goals or, intentions that it has not already rejected. In order to do this, it represents and stores choices that have failed in the past by associating a set of *failure contexts* with each problem. If a context consists only of bindings B, it means that the choice B did not work out. If it contains both bindings B and a goal G, it denotes that the problem solver failed to accomplish G with B as the bindings, whereas a context that includes bindings, a goal, and an intention I implies that I failed to execute in the setting. Everytime ICARUS rejects a choice for any of the reasons above, it creates and stores an appropriate failure context with the current problem. Later, it utilizes these contexts to rule out candidates while making a decision.

In the event that it fails due to any of the above causes, the system takes a step back to consider other choices. Thus, when it fails on an intention I, it considers other skills for the current focus goal G. If no such skills exist or if it has tried all options, it rejects G and looks for other goals to set as the focus. Finally, if no other choices exist for the focus goal, it abandons the current bindings B and considers different bindings. If no alternatives remain, it gives up on problem P and the intention associated with P's

parent.

Since ICARUS executes an intention as soon as it becomes applicable, it might change the environment in ways that cause difficulty in satisfying other goals, in which case it cannot backtrack mentally after taking the action. In such circumstances, the architecture attempts to solve the problem from the current environmental situation, eventually undoing the undesirable action if needed. This phenomenon is an unavoidable sideeffect of the eager execution stategy that the system employs to avoid keeping entire solution plansin memory. Eventhough it has clear disadvantages, this strategy is observed when humans attempt to solve complex problems.

Chapter 5

Experimental Evaluations

In the previous chapters we described extensions to ICARUS' problem-solving mechanism and made claims about the benifits offered by them. We now present evalutions of the system that support these claims. To study the behavior of the extended problem-solving mechanism, we carried out systematic experiments on it, employing various problem domains that tested the architecture's capabilities. In this chapter we describe these experiments, for each of which we examine the hypothesis, dependent and independent variables, knowledge provided, and the behavior of the system. We conclude by summarizing the observations.

5.1 Problem Solving in Tower of Hanoi

Before conducting systematic experiments, we evaluated the system on the classic *Tower of Hanoi* puzzle to establish that the modified problem solver worked as intended. The puzzle specifies a set of disks that one must move from a source peg to a target peg, one disk at a time. The challenge in this problem arises from the conditions that govern movement of the disks. In particular, it is only possible to move an unobstructed disk to the target peg which in turn should not have any smaller disks on it. Table 5.1 shows the single skill we supplied to the architecture that allowed it to move disks and encapsulated these conditions in form of negations.

For this demonstration, the top-level problem specified goals that were all grounded, so there was no choice in bindings but the system still had to select one of these as its focus, which produced some search. During problem solving, ICARUS repeatedly encounters situations in which one of the negated conditions is unsatisfied. Because no skill achieves these conditions directly, the system chains off this condition and creates a disjunctive subproblem by negating the relations of a concept with a head that

Table 5.1: Tower of Hanoi Operator

```
((move ?disk ?from ?to)
  :conditions ((peg ?from) (peg ?to) (disk ?disk)
               (different-peg ?to ?from) (on-peg ?disk ?from)
               (not (blocking-from ?smaller-from ?disk))
               (not (blocking-to ?smaller-to ?disk ?to)))
  :action (*move ?disk ?from ?to))
  :effects ((on-peg ?disk ?to)
            (not (on-peg ?disk ?from)))
```

matches the literal in the unsatisfied condition. The architecture must satisfy only one of the goal elements to solve a disjunctive problem. There is some search involved in achieving this goal and on doing so, it returns to a higher-level problem thus switching between conjunctive and disjunctive goals before it has satisfied the top-level problem.

The old problem-solving module could not handle this task unless provided with carefully crafted skills that achieved negations of defined concepts. Thus, successful tests on this domain established that the modified problem solver could solve problems in the Tower of Hanoi domain with domain representation that was not kludged and the ability to chain off negations helped in this. These results encouraged us to carry out systematic experiments, to which we now turn.

5.2 Testing System Scalability

Our first experiment was aimed at demonstrating that the new version of ICARUS solved problems with reduced effort over its ancestor. In particular, we claimed that the extended problem-solving mechanism carried out less search than the older version and consequently scaled better. To test this hypothesis, we ran both the systems on problems from the *Depots* domain, that varied in their complexity. The independent variables in this experiment were the version of the system used and the complexity of the problem as measured by the number of goals required to be satisfied. The dependent variable

was the CPU time taken to solve a problem.

Problems in the Depots domain specify a number of locations with stacks of crates that an agent must move to a different location. The available operators include using hoists to lift and drop crates and trucks to transport them. Satisfying a goal involves loading a crate in a truck, driving it to a target location, and stacking it as desired. Based on the initial configuration of crates, some tasks may also require moving other crates in order to make this possible, so even for a problem with a particular complexity, the length of the solution can vary with changes in the initial state. To ensure a fair distribution of the results, for each level of problem complexity we measured the run time of both systems on five different initial configurations and then averaged them to get a final value.

Before examining the results, we briefly outline the working of the new system. For this study, we gave the new version of ICARUS 12 conceptual rules that let it recognize arrangements of crates, hoist location and status, and truck location and contents. We also provided four primitive skills for hoisting and dropping a crate, loading or unloading a crate into a truck, and moving the truck. The system was configured to use the minimum-unsatisfied-conditions heuristic to guide skill selection. Again, all goals in the top-level problem were grounded, so the extended problem solver had no choice of bindings. However, the system had to choose from many possible focus goals, from which it picked at random. This choice in turn led to selection of a skill for dropping the relevant crate and produced a subproblem based on its conditions.

Typically only one of these conditions was not satisfied—that some hoist be holding the crate at the desired location. There were two skills that would achieve this end, one that hoisted the crate and one that unloaded it from a truck. The heuristic led the architecture to select the hoist skill, but later it realized that this required the crate to be at the desired location. Detecting this loop, it backtracked and selected the unload option. Continued reasoning along these lines eventually led to executable skills and

successful placement of the crate. ICARUS then repeated this entire process for each crate, halting after it had transferred them all to their desired locations.

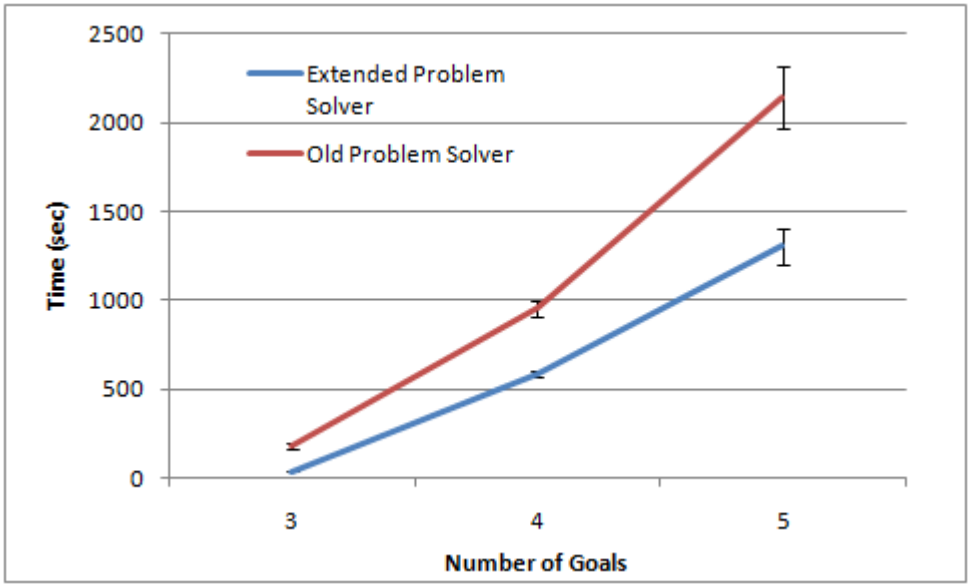

Figure 5.1: System Runtime in Depots Domain as a function of Problem Complexity

Figure 5.1 shows the plots of the average time taken by both the system for each level of problem complexity, with the error bars showing 95% confidence intervals. Both the systems were able to solve all the problems and as can be seen, the extended problem solver took much less time than the older version presumably because it carried out less search. This supports our initial hypothesis that the new system scales better with respect to complexity, at least as measured by the number of goals.

5.3 Demonstrating the Benefits of Constraints

A second experiment focused on establishing benefits of using constraints to guide search. We hypothesized that the availability of domain-specific constraints would reduce the amount of search involved in finding a solution to a problem. To operationalize this hypothesis into a testable claim, we carried out a lesion study on the modified problem solver by evaluating it on problems in the *Logistics* domain. Thus, the independent variables for this experiment were the absence or presence of constraints and the com-

plexity of problems as measured by the number of member goals in the problem. The dependent variables were the CPU time taken to find a solution, the number of nodes explored by the architecture, and the number of nodes out of these that failed.

The Logistics domain specifies a number of packages placed at locations that are spread across different cities. Operations in the domain allow loading and unloading of packages and provide trucks and airplanes to move them. Moreover, there are restrictions on the movement of these vehicles, with trucks being able to move only between locations within the same city and planes being able to move only between airports. Problems typically involve moving one or more packages across locations using a combination of ground and air transport. Thus solving a problem requires loading a package into a truck, driving it to an airport, unloading the package and loading it onto an airplane, flying the plane to a different city and then using another truck to deliver it to the target location. Unlike the Depots domain, problems in Logistics with the same complexity are isomorphic and hence have the same solution length. Hence we did not employ the averaging technique for measuring system run times.

In this experiment, we supplied ICARUS with 11 conceptual rules that let it reason about the location of packages, trucks, and airplanes, the contents of trucks and airplanes, and the type (airport or regular) and city of a location. We also furnished it with four primitive skills that let it load or unload a package, drive trucks, and fly planes. As before, the system was configured to use the minimum-unsatisfied-conditions heuristic to guide skill selection. Once again, since all goals in the top-level problem were grounded, there was no choice of bindings, but there was a choice in selection of focus goals. This in turn led to selection of a skill that would unload a package from a truck at the target location, leading the problem solver to create a subproblem from the skill's conditions.

One of these conditions specified that the truck should contain the package to be unloaded, which led to selection of a skill to load the truck. Further backchaining

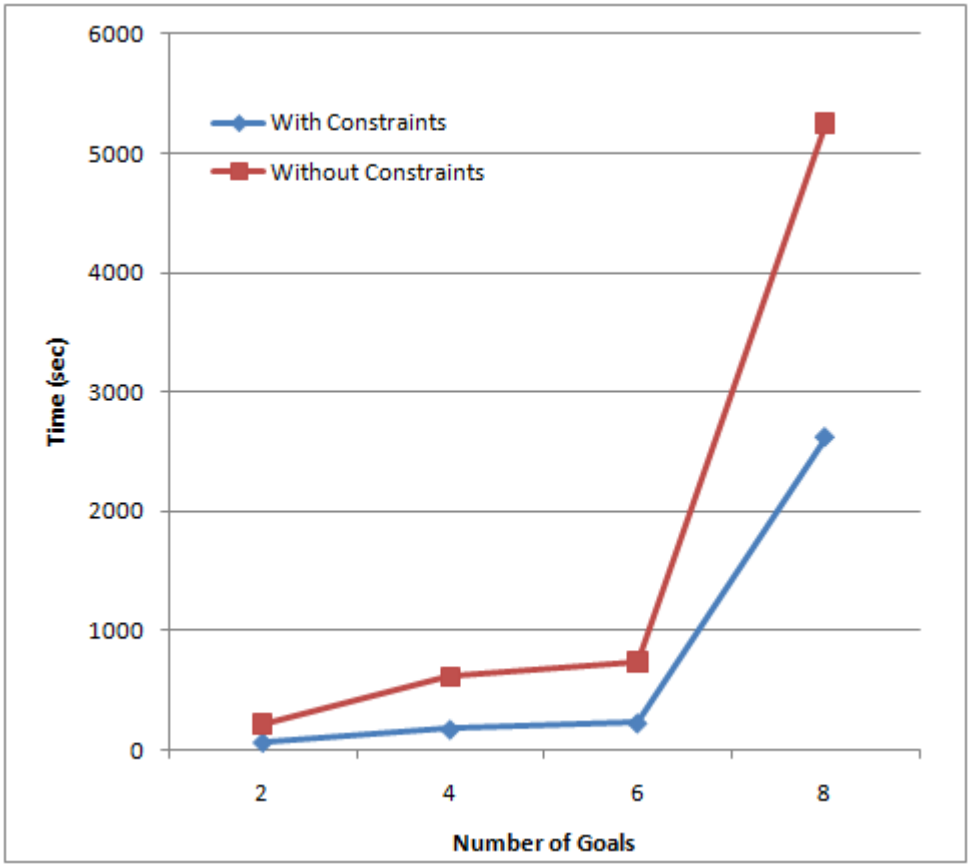

Figure 5.2: System Runtime in Logistics Domain as a function of Problem Complexity

induced selection of skill that would fly the package to a location where the truck can be loaded. Continuing in this manner, the architecture finally selects a truck at the source location and transports the package to the source city's airport. The whole process was then repeated for other packages. Whenever faced with the choice of a truck for a specific location, ICARUS would look for constraints to guide this choice. The constraints, if present, would cause it to reject trucks that could not travel to the location under consideration, thus preventing the problem solver from making choices that would eventually fail.

As the plots in Figure 5.2 show, the system with constraints ran substantially faster, presumably because it carried out less search, than when constrainte were unavailable. Moreover, the extended problem solver also created fewer subproblems and

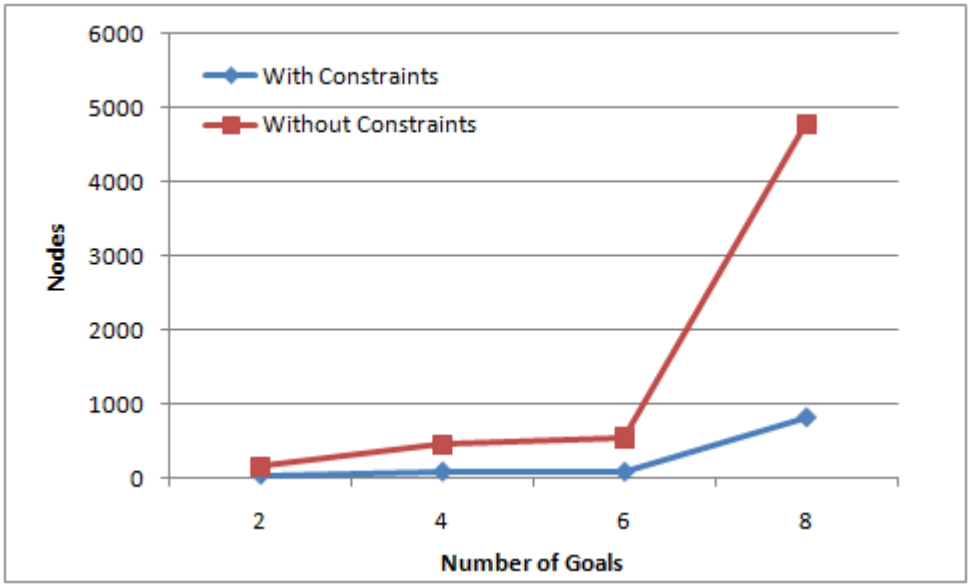

Figure 5.3: Number of Nodes Explored in Logistics Domain

had to abandon them less often when it employed constraints. This is evident from the plots in Figures 5.3 and 5.4. These results support our claim that using constraints reduces the amount of search involved in problem solving.

5.4 Demonstrating the Benefits of Heuristics

Our final study involved assessing the advantages of utilizing domain-independent heuristics to guide search. The initial assumption was that, when employed individually, either heuristic—minimum-unsatisfied-conditions and maximum-effects-matched—would provide comparable benefit in terms of reducing search over the case when none was used. We carried out prelimiary experiments in the form of lesion studies in the Depots and Logistics domains. We will not provide the details of these experiments here, but the results from these studies did not support our assumption.

In fact, we found that while the conditions heuristic did help ICARUS perform better than it did when it had no guidance in form of heuristics. However, there was no evidence of any reduction in search and backtracking when using just the effects

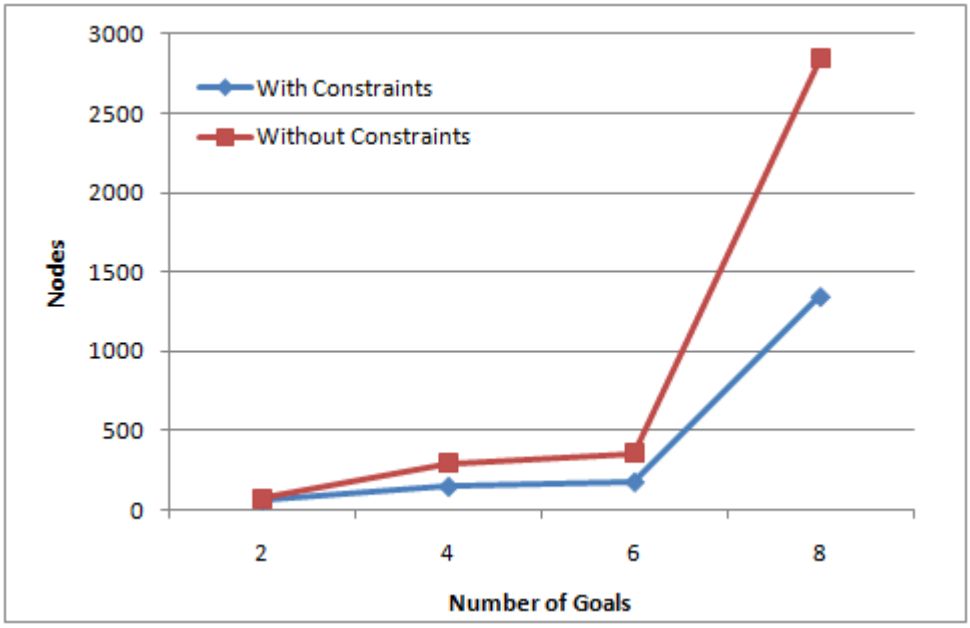

Figure 5.4: Number of Nodes Rejected in Logistics Domain

heuristic and it provided almost no advantage over runs in which the problem solver used neither heuristic. This suggested that the heuristic's effectiveness may be linked to the type of skills available to the agent. In particular, we had supplied ICARUS with only primitive skills in both the Depots and Logistics domains. Since primitive skills have only a few effects, they scored the same on the maximum-effects criterion and it did not provide much of a help in skill selection.

This observation led to the hypothesis that the maximum-effects-matched may provide more guidance in domains where the architecture has access to non-primitive skills and the reduction in search in such case may be comparable to that achieved by the conditions heuristic. To test this claim, we provided the system with higher-level skills for the *Gripper* domain and ran it on problems of varying complexity in two distinct settings using different heuristics. Thus the independent variables in this case were the complexity of the problems in terms of the number of goals and the type of heuristic used. The dependent measure was again the CPU time taken to solve problems.

The Gripper domain involves several rooms, a number of balls, and a robot with multiple grippers that can hold one object per gripper. Problems specify target rooms for several balls that begin in a source room. Satisfaction of these goals requires an agent to move particular objects to specified rooms, making as few trips as possible by carrying multiple balls at a time. As in the Logistics domain, problems in Gripper with the same complexity are isomorphic, so we did not run the system multiple times for a particular complexity level. For this domain, we provided Icarus with ten concepts for recognizing relevant situations, such as whether a gripper is holding an object and whether the agent is holding as many objects as possible. We also gave the system four skills, one of which was both hierarchical and recursive.

As before, the problem solver does not have to select bindings at the top level, as all goals are grounded. It proceeds to select one goal as its focus and chooses the only relevant skill for transporting a set of held objects. Because the skills conditions are unsatisfied, ICARUS creates a subproblem to achieve them. After selecting bindings, the subproblem has only one unmatched condition—that the agent should be holding as many objects as possible—so this becomes the new focus goal. The system does not include a skill that produces this as an effect, but it does have a definition for the relevant concept. The problem solver repeatedly chains backward off this rule, leading it to execute the primitive skill for picking up objects until it is holding the maximum number possible. This in turn satisfies the conditions of the original skill instance, causing ICARUS to move the robot to the target room. At this point, it repeatedly drops objects until all grippers are free, then shifts to moving objects to a different room.

We observed in that, in the Gripper domain, both the heuristics offered substantial help to the problem solver in making progress towards a solution. However, the system carried out a lot more search in runs where it did not utilize any heuristic and consequently scaled poorly with increasing number of goals. In fact, as the plots in Figure 5.5 show, the problem solver indeed takes less time, presumably because it

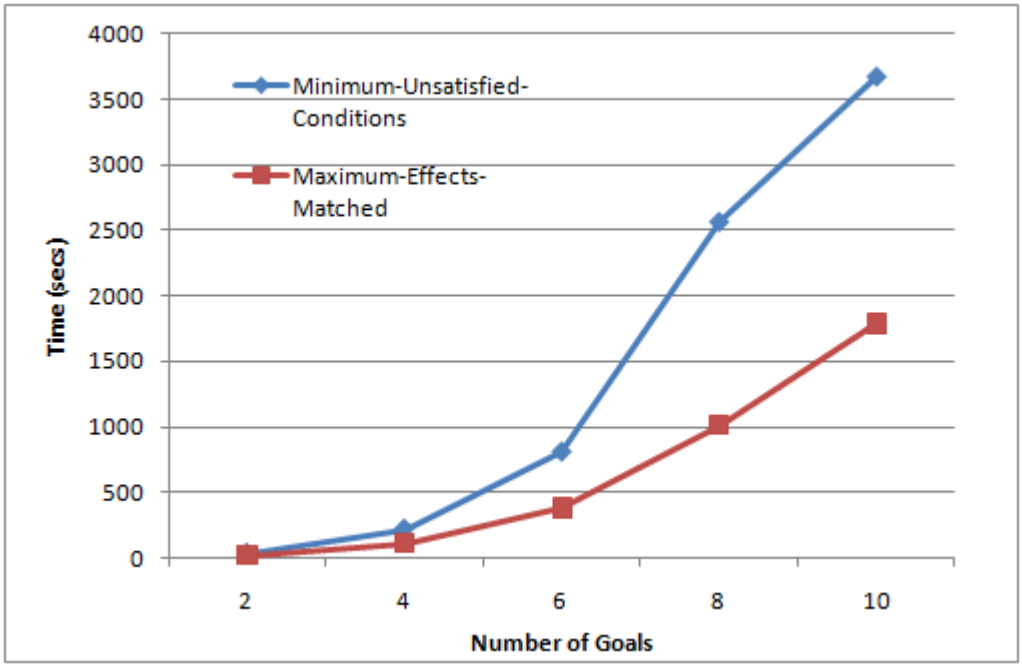

Figure 5.5: System Runtime in Gripper Domain as a function of Problem Complexity

performs less search, when using the maximum-effects-matched heuristic in the Gripper domain, and it scales better with respect to increasing problem complexity. These results support our claim that the effectiveness of heuristics is linked to the type of operators available to choose from and in particular the maximum-effects-matched heuristic proves helpful when the system has access to non-primitive skills.

5.5 Summary Remarks

In this chapter we described experimental evaluations of ICARUS' extended problem-solving mechanism. We saw that, with its ability to chain off negations, it could solve problems in the Tower of Hanoi domain with a less handcrafted representation. We also demonstrated that the new system carried out less search in comparison to its ancestor and hence scaled better with increase in problem complexity. Moreover, we established that use of constraints cuts down the search even more. Finally, we showed how the effectiveness of the heuristics varies with the type of the knowledge available for a problem domain. Thus the extensions proposed in this thesis not only extend the

architecture's account of problem solving but also improve its performance, bringing it closer to functionality observed in humans.

Chapter 6

Related Research

There has been a long history of research on computational models of problem solving and hence it should not be surprising that some ideas similar to those we have proposed have appeared elsewhere. We believe that we are the first to combine them in a unified framework, but we should still mention earlier efforts that have addressed similar issues. In this chapter, we discuss the research related to the extensions we proposed in chapter 3 in the same order as we discussed there.

Recall that our first extension was the introduction of a distinct memory structure called "problems" around which we organize the representation of the underlying problem space. This idea is not new; traditional planners represent goals as conjunctive sets and search in the space of these sets and the Soar [11] framework relies on the notion of search through a problem space, although their approach differs from ours in many details. Our approach also resembles that taken by the Prodigy [3] architecture, which incorporates a representation of problems and subproblems in which goals are embedded.

The introduction of problems lets ICARUS create subproblems when it chains off operators or concepts. This idea appears first in the classic work on GPS [19] by Newell and Simon as part of the "apply-operator" step. Laird et al. refer to backward chaining as "operator-subgoaling" in their work on Soar. Our approach differs from both in that it subgoals not only on operators but also on concepts, and its representation of the resulting subproblems is more flexible since, as described earlier, it encapsulates both conjunctive and disjunctive problems.

Our distinction between multiple problem formulations in terms of bindings appears to be more novel. Prodigy also incorporated choice points for bindings, but

it relied on domain-specific control rules to guide selection in the absence of which it made an arbitrary choice. ICARUS utilizes domain-specific constraints when available, but it also incorporates a domain-independent process that compares problem elements against the current state to generate partial matches, from which it selects one that is maximal. No other work that examines the role of problem formulation in creating problem spaces embeds such a process within problem solving itself.

Our idea of using domain-specific constraints to guide search is also not novel. Prodigy included control rules that would select, reject, or prefer an operator, goal, or binding. In a similar way, TLPlan [2] represents control knowledge in temporal logic and utilizes it to control a forward-chaining planner. The recent work on landmarks [7] by Hoffmann et al. also supplies planners with control knowledge which, like ours, specifies ordering on problem goals based on their interactions.

Of course, the use of heuristics for guiding problem-space search dates back to the field's beginning, but these have typically incorporated domain-specific knowledge rather than generic calculations about relations between operators, goals, and states. Traditional planning methods have used a set-difference heuristic to perform a reachability analysis and guide the selection of operators. This is similar to our maximum effects heuristic, an earlier version of which was reported by Jones and Langley [9]. Nejati [16] has explored the minimal unsatisfied conditions metric in a similar setting, but we believe that we are the first to combine them in a single system.

To our knowledge, the use of disjunctive goals to handle chaining over complex negated conditions is entirely novel. Most traditional planners do not operate in a closed system and avoid this kind of reasoning with negations since it can possibly lead to infinite loops. But ICARUS unifies problem solving with execution and problem state updation and this closed loop allows it to avoid the possible pitfalls of reasoning with unsatisfied negations.

Chapter 7

Directions for Future Work

In the previous chapters, we described ICARUS' account of problem solving and demonstrated its generality across domains. We also compared it with related approaches and showed that the architecture offers broader capabilities than many of these efforts. However, there remain some limitations in the current framework that we should address in future work.

One drawback is that the current version of ICARUS does not include a learning mechanism that reflects on traces of the problem solver's efforts to achieve a given set of goals. Humans display a clear ability to learn generalized structures from their past behavior that aid them in solving similar problems in future. Indeed, the analysis of a successful solution trace reveals information that can be used to learn generalized goal constraints like those we described in chapter 3. More specifically, such a trace contains information about order in which the problem goals were achieved, as well as about the extra goals that the architecture generated. [1] In fact, Nejati [16] reports a system which can learning such information by annotating successful traces. We can adapt her approach to learn structures that can be used by the new problem solver.

However, it is also possible to learn new knowledge from problem-solving failures. To this end an extended version of the architecture could construct generalized bindings constraints every time a system fails on a set of bindings. This process could examine the traces to assign and propogate blame, terminating when it finds the conditions responsible for the failure—an analysis similar to dependency directed backtracking [10]. The system could then encapsulate them into bindings constraints to eliminate similar choices on future tasks.

[1] Recall that these goals describe a "landmark" [7] belief that would become true on every solution path for the given problem.

Another limitation arises from the systematic nature of ICARUS' problem-space search. It has been observed that humans seldom perform a depth-first search, partly due to the memory limitations that make it difficult to maintain a complete chain of problems. de Groot [5] contends that chess players instead use *progressive deepening*, a search strategy that explores a single path at a time and only remembers the first decision and the quality of the outcome. Jones and Langley [9] use a similar technique of *iterative sampling* in their EUREKA architecture to organize problem-space search. We should include similar methods in our accounts of problem solving to enable memory-limited, non-systematic search.

A third limitation of the architecture is that, unlike humans, it always carries out backward-chaining search from goals. Observations of human behavior make it evident that they sometimes resort to forward-chaining search with almost no explicit influence from goals. This is especially true for domains like chess, in which goals are very abstract and do not prove to be useful until late in the problem-solving process. To complete ICARUS' account of problem solving, we should incorporate a forward-chaining search capability along with a mechanism that decides when to switch between the two modes of search.

A final limitation is related to insight in finding solutions to problems. The Gestalt psychologists observed that people find certain tasks quite difficult but, upon reconsideration, see the answer very quickly, and no standard theory of problem solving accounts for this behavior. One possible explaination suggests that humans modify the structure of the problem space such that it transforms the task into one that contains a solution. The next version of ICARUS should incorporate mechanisms that address this intriguing phenomenon.

Chapter 8

Conclusion

In this thesis, we reviewed the key aspects of the standard theory of problem solving including, but not limited to, the notions of physical symbol systems, problem space search, heuristic search, and means-ends analysis. We also proposed extensions to the theory and claimed that problem solving typically occurs in a physical context and is interleaved with execution. We also asserted that problem solving remains at an abstract level that insulates the solver from unnecessary details.

In addition, we described how ICARUS incorporates each of these theoretical commitments at the architectural level and examined some limitations that still remained unaddressed. We then outlines four extension that cover these limitations. Some of these extensions appeared in the frameworks representational commitments to problems, goals, and intentions, whereas others were linked to mechanisms for carrying out goal-directed heuristic search through a problem space. We argued that ICARUS implements these claims in a psychologically plausible manner, and we demonstrated the architectures generality on problem solving in a variety of domains.

We will not claim that particular forms in which ICARUS implements these ideas are the only alternatives, but they do provide an operational account that one can use to develop specific models of problem solving on particular tasks. We should also note that none of these ideas, in isolation, are new to ICARUS . All have appeared elsewhere in the literature and many have been used in computational models. However, this does not diminish their relevance to unified theories of cognition, and, to our knowledge, ICARUS is the first such theory to incorporate them all in a coherent manner.

[1] J. R. Anderson. *Rules of the mind*. Lawrence Erlbaum, Hillsdale, NJ., 1993.

[2] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In *Proceedings of Second International Workshop on Temporal Representation and Reasoning (TIME)*, Melbourne Beach, FL., 1995.

[3] J. Carbonell, O. Etzioni, Y. Gil, R. Joseph, C. Knoblock, S. Minton, and M. Veloso. Prodigy: an integrated architecture for planning and learning. In *ACM Special Interest Group on Artificial Intelligence (SIGART) Bulletin*, volume 2, pages 51–55, 1991.

[4] A. Danielescu, D. J. Stracuzzi, N. Li, and P. Langley. Learning from errors by counterfactual reasoning in a unified cognitive architecture. In *Proceedings of the Thirty-Second Annual Meeting of the Cognitive Science Society*, Portland, OR., 2010. Lawrence Earlbaum.

[5] A. D. De Groot. *Thought and choice in chess*. Mouton, The Hague, The Netherlands, 1965.

[6] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Artificial Intelligence*, volume 2, pages 189–208, 1971.

[7] J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[8] A. Horn. On sentences which are true of direct unions of algebras. In *The Journal of Symbolic Logic*, volume 16, pages 14–21, 1951.

[9] R. Jones and P. Langley. Retrieval and learning in analogical problem solving. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 446–471, Pittsburgh, PA., 1995. Lawrence Earlbaum.

[10] S. Kambhampati. Formalizing dependency directed backtracking and explanation based learning in refinement search. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 1 (AAAI'96)*, volume 1, pages 757–762. AAAI Press, 1996.

[11] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. In *Artificial Intelligence*, volume 33, pages 1–64, 1987.

[12] P. Langley and D. Choi. A unified cognitive architecture for physical agents. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, Boston, MA., 2006. AAAI Press.

[13] P. Langley, J. E. Laird, and S. Rogers. Cognitive architectures: Research issues and challenges. In *Cognitive Systems Research*, volume 10, pages 141–160, 2009.

[14] P. Langley and S. Rogers. An extended theory of human problem solving. In *Proceedings of the Twenty-Seventh Annual Meeting of the Cognitive Science Society*, Stressa, Italy, 2005.

[15] J. H. Larkin, J. McDermott, D. P. Simon, and H. A. Simon. Expert and novice performance in solving physics problem. In *Science*, volume 208, pages 1335–1342, 1980.

[16] N. Nejati. *Analytical Goal-Driven Learning of Procedural Knowledge by Observation*. PhD thesis, Stanford University, CA., 2011.

[17] A. Newell. *Unified theories of cognition*. Harvard University Press, Cambridge, MA., 1990.

[18] A. Newell, J. C. Shaw, and H. A. Simon. Elements of a theory of human problem solving. In *Psychological Review*, volume 65, pages 151–166, 1958.

[19] A. Newell and H. A. Simon. Gps, a program that simulates human thought. In *H. Billing (Ed.), Lernede automaten*, Munich: Oldenbourg KG, 1961.

[20] A. Newell and H. A. Simon. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[21] A. Newell and H. A. Simon. Computer science as an empirical enquiry. In *Communications of the ACM*, volume 19, pages 113–126, 1976.

[22] K. VanLehn. *Mind bugs: The origins of procedural misconceptions*. MIT Press, Cambridge, MA., 1990.

Appendix A

LISP Code

```
(defun backward-chain-problem-solve (problem)
  (cond

    ;; Step 1. Select bindings
    ((null (problem-bindings-selected? problem))
     (print "Selecting bindings.")
     (let ((result nil))
       (setq result (select-bindings problem))
       (process-select-bindings-result problem result)))

    ;; Step 2. Select focus
    ((or (not (problem-focus problem))
 (goal-literal-satisfied? (problem-focus problem)))
     (print "Selecting focus.")
     (setf (problem-intention problem) nil)
     (setf (problem-focus problem) (select-focus problem))

     (cond ((problem-focus problem)
    (print "Focus selected, proceeding to select a skill for the focus")
    (print (problem-focus problem)))
   (t
    (print "Failed to select focus for the current problem
    with the given bindings. Backtracking.")
    (create-and-store-failure-context (problem-bindings problem))
    (setf (problem-bindings problem) nil)
    (setf (problem-bindings-selected? problem) nil))))

    ;; Step 3. Select Skill
    ((not (problem-intention problem))
     (let ((result-triple nil)
   (intention nil)
   (focus-breakup-triplet nil))
       (print "Selecting skill.")
       (setq result-triple (select-skill problem))

       (cond (result-triple
      (setq intention
    (create-intention (second result-triple) (third result-triple)))
       (setf (intention-problem-parent intention) problem)
       (setf (problem-intention problem) intention)

       (print "Skill selected")
```

```
      (pprint-intention intention)

      (make-problem-from-conditions intention)
      (incf nodes-explored*))

     ;; Select-skill failed. Try concept chaining before recording failure.
     (t
      (print "Failed to select skill for the current problem with given
      bindings and focus. Trying to chain on concept.")

      ;; Concept chaining.
      (setq focus-breakup-triplet
       (find-matching-concept (problem-focus problem)
      :problem-bindings (problem-bindings problem)
   :ignore-failure-context? nil))

      (cond ((and focus-breakup-triplet
  (second focus-breakup-triplet))
     (setq intention
   (create-dummy-intention-for-concept-chaining focus-breakup-triplet))
     (setf (intention-problem-parent intention) problem)
     (setf (problem-intention problem) intention)

     (print "Concept found for chaining. Dummy intention for the
     concept created.")
     (pprint-intention intention)

     (make-problem-from-conditions intention)

     (incf nodes-explored*)

     ;;If chaining of a negative goal, flag the new problem as disjunctive.
     (if (eq (car (problem-focus problem)) 'not)
 (setf (problem-disjunctive-goal-list? active-problem*)
       t)))
    (t
     (print "No concept found for concept-chaining for the focus of
     the current problem. Backtracking.")
     (create-and-store-failure-context (problem-bindings problem)
       (problem-focus problem))
     (setf (problem-focus problem) nil)))))))
    (t
     (warn "You should never be here in backward-chain-problem-solve."))))

(defun select-bindings (problem)
```

```
  (let ((pos-literals nil)
(neg-literals nil)
(intermediate-bindings nil)
(results nil)
(selected-match nil)
(return-value nil))

    ;; Extend problem bindings with bindings from unchainable
    ;;goals of the problem.
    (loop for pos-objective in (problem-pos-objectives problem)
  do
  (when (member (car pos-objective) pos-unchainable-conditions* :test #'eq)
    (push pos-objective pos-literals)))

    (loop for neg-objective in (problem-neg-objectives problem)
  do
  (when (member (caadr neg-objective) neg-unchainable-conditions* :test #'eq)
    (push neg-objective neg-literals)))

    (setq intermediate-bindings
         (find-all-match-bindings-for-sorted-literals pos-literals
                                      neg-literals
      (problem-bindings
      problem)
      cstm*))

    (setq results
         (find-all-max-size-partial-matches-for-sorted-literals-for-binding-set
                  (problem-pos-objectives problem)
(problem-neg-objectives problem)
intermediate-bindings
cstm*))

    (when results
      (setq selected-match (random-choose results))

      (cond ((and (problem-disjunctive-goal-list? problem)
  (>= (length (car selected-match))
      1))
     (setq return-value (cons t (second selected-match))))
    ((and (null (problem-disjunctive-goal-list? problem))
  (= (length (problem-goals problem))
     (length (car selected-match))))
     (setq return-value (cons t (second selected-match))))
    (t
```

```
     (setq return-value (cons nil (second selected-match))))))
    return-value))

(defun process-select-bindings-result (problem result)
  (cond ((null result)
 ;;This means failure. Store appropriate context in parent and pop-up.
 (print "Failed to select bindings for the current problem. Backtracking.")
 (cond ((problem-i-parent problem)
(setq active-problem* (intention-problem-parent (problem-i-parent problem)))
(create-and-store-failure-context (problem-bindings active-problem*)
  (problem-focus active-problem*)
  (problem-intention active-problem*))
(setf (problem-bindings active-problem*) nil)
(setf (problem-bindings-selected? active-problem*) nil)
(incf nodes-failed*)
)
       (t
(warn "Failed to select bindings for the top level problem.")
(setq active-problem* nil))))

((null (car result))
 ;; This means that bindings were successfully found for a problem
 ;; that has unsatisfied goals.
 (cond ((not (repeated-problem? problem
(cdr result)))
;; This problem has not been encountered till now. Proceed to focus selection.
(setf (problem-bindings problem) (cdr result))
(setf (problem-bindings-selected? problem) t)

;; Set both focus and intention to NIL to make sure new ones get chosen
(setf (problem-focus problem) nil
      (problem-intention problem) nil)

(print "Bindings selected.")
(print (problem-bindings problem)))
       (t
(print "Found a repeated problem! Reporting the current bindings as a failure.")
(create-and-store-failure-context (cdr result)))))

((car result)
 ;; This means that the bindings found are such that all goals of
 ;; the current problem are satisfied.
 (cond ((and (problem-i-parent problem)
     (intention-id (problem-i-parent problem)))
```

```
(cond ((and execution-loop*
    (member (intention-head (problem-i-parent problem))
    (get-repeating-intentions)
    :test #'equal))
       (print "Currrent intention matches a repeating intention.
       Aborting its execution to break execution loop.
       Backtracking.")
       (reset-loop-traces)
       (setq active-problem* (intention-problem-parent
        (problem-i-parent problem)))
       (create-and-store-failure-context (problem-bindings active-problem*)
 (problem-focus active-problem*)
 (problem-intention active-problem*))
       (setf (problem-intention active-problem*) nil)
       (incf nodes-failed*))
      (t
       (print "All skill conditions satisfied, proceeding to execution")
       (set-executing-intention (problem-i-parent problem))
       (setq active-problem* (intention-problem-parent
        (problem-i-parent problem)))

       (setf (intention-bindings (problem-i-parent problem))
     (append (intention-bindings (problem-i-parent problem))
     (cdr result)))
       (establish-correspondence-for-intention-targets active-problem*)

       (setf (problem-bindings-selected? active-problem*) nil))))

       ((and (problem-i-parent problem)
     (null (intention-id (problem-i-parent problem))))
(print "All conditions of the chained-concept satisfied,
selecting new bindings and focus for parent problem.")
(setq active-problem* (intention-problem-parent (problem-i-parent problem)))
(setf (problem-bindings-selected? active-problem*) nil))
       (t
(report-toplevel-problem-satisfied)
(setq active-problem* nil))))))

(defun select-focus (problem)
  (let ((goals-added nil)
(unsatisfied-goals nil)
(goals-to-delete nil))

    ;; Step 1: Get all the goals in the current problem that are
    ;; unsatisfied and that have not failed before.
```

```
    (loop for goal in (subst-bindings (problem-bindings problem)
      (problem-objectives problem))
  when (and (not (goal-literal-satisfied? goal))
    (not (member-failure-context-list? (problem-bindings problem)
       goal)))
  do
  (push goal unsatisfied-goals))

    ;; Step 2: Get all the goals added by add constraints and all the goals
    ;; deleted by ordering constraints.
    (loop for constraint in constraint-memory*
  when (constraint-active? constraint unsatisfied-goals)
  do
  (cond ((constraints-add constraint)
 (setq goals-added (append (subst-bindings (constraints-bindings constraint)
   (constraints-add constraint))
   goals-added))
 (print "Found an active add constraint."))
((constraints-delete constraint)
 (setq goals-to-delete (append (subst-bindings
  (constraints-bindings constraint)
        (constraints-delete constraint))
       goals-to-delete))
 (print "Found an active delete constraint.")))

    ;; Step 3: Remove all goals marked for deletion.
    (loop for goal in goals-to-delete
  do
 (setf goals-added (delete goal goals-added :test #'equal))
 (setf unsatisfied-goals (delete goal unsatisfied-goals :test #'equal)))

    ;; Step 4: Select a focus.
    (cond (goals-added
   (random-choose goals-added))
  (unsatisfied-goals
   (random-choose unsatisfied-goals))
  (t
   nil))))

(defun select-skill (problem &optional (bind-unchainables? t))
  ;; members of candidate-skills are of the form (effect sclause bindings)
  (let ((candidate-skills nil)
(focus-goal (problem-focus problem))
(selected-triple nil))
```

```
    (loop for skill in sltm*
  do
  (loop for effect in (sclause-effects skill)
for (flag . bindings ) = (unify-match focus-goal effect)
do
(when flag
  (push (list effect skill bindings)
candidate-skills))))
    (when debug*
      (print "**** Candidates Matching Focus ****")
      (mapcar #'print candidate-skills))

    (when bind-unchainables?
      (setf candidate-skills
    (loop for (effect sclause bindings) in candidate-skills
  append (find-all-candidates-satisfying-unchainable-conditions effect
sclause
bindings))))
    (when debug*
      (print "**** Candidates with Unchainable Bindings ****")
      (mapcar #'print candidate-skills))

    (loop with results = nil
  for (effect sclause bindings rest) in candidate-skills
  when (not (member-failure-context-list? (problem-bindings problem)
  (problem-focus problem)
  (create-intention sclause bindings)))
  do
  (push (list effect sclause bindings rest) results)
  finally
  (setq candidate-skills results))

    (when candidate-skills
      (case skill-selection-heuristic*
(:MAX-EFFECTS-MATCHED
 (loop with best-quadruplets = nil
       with max-value = -1
       for quadruplet in candidate-skills
       for current-value = (max-effects-matched-heuristic
        (butlast quadruplet) problem)
       do
       (cond
 ((< max-value current-value)
  (setq max-value current-value
best-quadruplets (list quadruplet)))
```

```
 ((= max-value current-value)
  (push quadruplet best-quadruplets)))
       finally
       (setq selected-triple (random-choose best-quadruplets))))

(:MIN-UNSATISFIED-CONDITIONS
 (loop with best-quadruplets = nil
       with min-value = nil
       for quadruplet in candidate-skills
       for current-value = (min-unsatisfied-conditions-heuristic
        (butlast quadruplet))
       do
       (cond
 ((or
   (null min-value)
   (> min-value current-value))
  (setq min-value current-value
best-quadruplets (list quadruplet)))
 ((= min-value current-value)
  (push quadruplet best-quadruplets)))
       finally
       (setq selected-triple (random-choose best-quadruplets))))
(:BOTH
 )))

    ;; Restore original bindings and discard bindings generated
    ;; due to binding unchainable conditions.
    (if selected-triple
      (setf (nth 2 selected-triple) (nth 3 selected-triple)))
    (butlast selected-triple)))
```

This LaTeX document was generated using the Graduate College Format Advising tool. Please turn a copy of this page in when you submit your document to Graduate College format advising. You may discard this page once you have printed your final document. DO NOT TURN THIS PAGE IN WITH YOUR FINAL DOCUMENT! font type: TimesNewRoman font size: 12