Source Selection for Analogical Reasoning
An Interactionist Approach

by
William Anthony Stubblefield

B. A. in English Literature, Stanford University, 1970
M. S. in Computer Science, The University of New Mexico, 1984

Doctor of Philosophy in Computer Science

May, 1995

I dedicate this work to my wife, Merry,
and to my parents, William and Maria.

Acknowledgements

Most of all, I owe the success of this work to my wife, Merry Christina. Her love and faith make everything possible.

This work would not have been possible without the patience, intelligence and dedication of my dissertation chairman, George Luger, and the members of my committee: Donald Morrison, Peder Johnson, Jim Hollan, and Stephanie Forrest. I thank them for seeing something in me worth nurturing, and helping me to bring it forth.

I owe an equal debt to all of the friends and colleagues who have supported and guided me through this process: Edward Angel, Alan Apt, Staughton Bell, Joanne Buehler, Jan Claus, Anna Dolan, Scot Drysdale, Donald Johnson, Kathleen Luger, Fillia Makedon, Rose Mary Molnar, Jim Moor, Pat Morrison, Samuel Rebelsky, Pete Sandon, Michael Sims, Carl Stern, Lucy Torres, Hy Yarchun and any others for whose unintentional omission from this list I apologize.

I thank my parents, William and Maria Stubblefield, who are responsible for whatever virtue and ability may be found in me, and all of my family, especially Barbara Thomas, Pete and Berte Peterson, and Bill and Diane Riggs and all of their children.

To Richard and Hela Miller, my dearest friends: thank you for your encouragement, support, and decades of uncompromising friendship.

Finally, I thank my students for allowing me to share the passion, symmetry and clarity of their developing minds. They constantly remind me that the quest to understand the patterns of mind is the greatest adventure of all.

Source Selection for Analogical Reasoning
An Interactionist Approach

by
William Anthony Stubblefield

Doctor of Philosophy in Computer Science

May, 1995

Source Selection for Analogical Reasoning
An Interactionist Approach

by
William Anthony Stubblefield

B. A. in English Literature, Stanford University, 1970
M. S. in Computer Science, The University of New Mexico, 1984
Ph.D. in Computer Science, The University of New Mexico, 1995

Abstract

Analogy is a process of discovering and reasoning about similarities: if two objects or situations are known to have certain properties in common, analogy lets us infer additional similarities between them. Analogical inference transfers knowledge from a well understood *source* domain to a poorly understood *target*. An essential first step in analogical reasoning is the selection of an appropriate source for the analogy. Most analogical reasoners rely upon a number of simplifying assumptions in selecting sources: these include restrictions on the choice and representation of information that may be considered in selecting sources, the use of monotonic measures of similarity, and a reliance on clustering techniques to construct index hierarchies through an analysis of potential sources.

Building on a theory of metaphor known as the *interaction theory*, this dissertation argues that these assumptions unnecessarily restrict source selection. These criticisms lead to the three conjectures investigated in this work: (1) source retrieval may compensate for missing information about target problems by making reasonable assumptions, and using these inferences to distinguish candidate sources; (2) the similarity of candidate targets and sources should be measured according to the source's anticipated effects on the systematic structure of the target problem; (3) inductive learning can improve source selection based on the reasoner's experience in solving target problems.

Table of Contents

1. Introduction and Overview **1**

1.1 The standard model of analogical reasoning 2

1.2 An interactionist critique of the standard model 5

1.3 An interactionist model of source selection 10

1.4 The goals and focus of this research 13

1.5 Overview of the dissertation 15

2. Analogical Inference and Source Retrieval **16**

2.1 Analogical inference 17

2.2 Source selection and Memory Organization 34

2.3 An interactionist critique of computational approaches to analogical reasoning 48

2.4 Conclusion 54

3. An Interactionist Model of Source Selection **56**

3.1 The analogical interpretation problem 57

3.2 Why this is an interesting problem 63

3.3 An overview of the SCAVENGER architecture 70

3.4 An extended example 80

3.5 Testing SCAVENGER 101

4. Evaluating SCAVENGER: Interpreting Tutorial Examples of LISP Methods **106**

4.1 The selection and representation of test data 106

4.2 Learning in SCAVENGER 108

4.3 Scaling in SCAVENGER 122

4.4 Comparing SCAVENGER with source oriented clustering methods 123

4.5 Conclusion 130

5. Evaluating SCAVENGER: Diagnosis of Procedural Bugs **132**

5.1 Representing Target Problems and Analogical Sources 133

5.2 Identifying bugs 140

5.3 Learning and scaling 143
5.4 Conclusion 148

6. Evaluating SCAVENGER: Reasoning About Simulations 151

7. Summary of Results and Future Research 158
7.1 Empirical evaluation of SCAVENGER 158
7.2 SCAVENGER and the interaction theory 159
7.3 Future Directions 163
7.4 Conclusion 165

Appendix 1. The Class and Method Library Used in the Tests of Chapter 4 167

Appendix 2. The Problem Set Used in the Tests of Chapter 4 188

Appendix 3. A SCAVENGER Hierarchy 203

Appendix 4. Sample SCAVENGER Runs 212

Appendix 5. Buggy Operators used in the Tests of Chapter 5 218

References 223

List of Figures

1. The water / electricity analogy ____ 3

2. A hierarchical index ____ 9

3. Analogical inference ____ 17

4. A geometric Analogy Problem ____ 18

5. An initial analogy between the atom and the solar system ____ 19

6. An annotated source used in derivational analogy ____ 24

7. Alternative analogical mappings ____ 26

8. Analogical reasoning as search through T-space ____ 28

9. Spreading activation in PARADYME ____ 42

10. A graph of a target transcript ____ 62

11. The SCAVENGER algorithm ____ 70

12. An analogical mapping ____ 71

13. A simple SCAVENGER index hierarchy ____ 72

14. A partial interpretation of the target problem ____ 73

15. A set of partial analogies ____ 75

16. Completing a partial analogy ____ 76

17. An updated index hierarchy ____ 78

18. A simple SCAVENGER index hierarchy ____ 81

19. An initial interpretation of a target problem ____ 83

20. A partial Interpretation ____ 87

21. A partial Interpretation ____ 88

22. Explanation of target problem based on INTERPRETATION-2 ____ 89

23. Explanation of target problem based on INTERPRETATION-3 ____ 90

24. Partial analogies under INTERPRETATION-2 ____ 92

25. A high-level interpretation and its set of similar analogies ____ 93

26. Another high-level interpretation and its set of similar analogies ____ 94

27. Partitions of target interpretations produced by two candidate specializers ____ 97

28. An updated version of the index of 18 ____ 98

29. Improvement in mean execution times per problem across training runs ____ 109

30. Reduction in number of analogies tested across training runs ____ 110

31. Reduction in median problem execution times across training runs ____ 112

32. Distribution of solution times on training set before training ____ 113

33. Distribution of solution times on training set after training ____ 114

34. Learning curve showing early fluctuations in solution times ____ 115

35. Comparison of heuristic and non-heuristic

versions of SCAVENGER ____ 119

36. Increase in average solution times as source base grows ____ 123

37. Alternative index hierarchies ____ 128

38. Adding digit types to the LISP type hierarchy ____ 137

39. Improvement in average problem solution times across repeated trials on training set ____ 145

40. Distribution of solution times on test problem set ____ 146

41. Scaling test results ____ 147

42. A simple flow system ____ 151

List of Tables

1. Source classed and methods listed by domain ____ 107

2. Speedup on test problems ____ 111

3. Distribution of sources among nodes of a typical index ____ 115

4. Solutions found at each level of the index ____ 115

5. Performance times (in seconds) on the test problems ____ 119

6. The effect of learning on the SCAVENGER's failure performance ____ 121

7. Solution times for SCAVENGER and the comparison algorithm ____ 126

8. Profile of the index hierarchies for SCAVENGER and the comparison algorithm ____ 127

9. Distinct analogies generated by SCAVENGER and the comparison algorithm ____ 127

10. Summary of SCAVENGER's Diagnoses ____ 147

1
Introduction and Overview

> *And so handled, the quest of physical causes merges with another great Aristotelian theme - the search for relations between things apparently disconnected and for 'similitude in things to common view unlike.' Newton did not show the cause of the apple falling, but he showed a similitude . . . between the apple and the stars. By doing so, he turned old facts into new knowledge . . .*
>
> *D'Arcy Wentworth Thompson*
> On Growth and Form

Metaphor and analogy are processes of discovering and reasoning about similarities: if two objects or situations are known to have certain properties in common, these processes let us infer additional similarities between them. Through metaphor and analogy, we use knowledge of one situation to shape our understanding of another.

The ability to reason with metaphors and analogies is a necessary component of many high-level cognitive activities. For example, natural language has long been recognized to be a vast web of metaphors (Lakoff and Johnson 1980): this is true not only of literature, but also of news articles, common conversation and even technical or journalistic writing. The opening sentence of a recent Wall Street Journal article on trade agreements is built around a "politics is war" metaphor:

> *After a grueling two-month battle, the Clinton administration seemed poised to win a cliff-hanger triumph on the North American Free Trade Agreement (WSJ, 11/17/93).*

Metaphors and analogies permeate not only natural language: My computer interface uses the visual metaphor of a trash can icon to represent file deletion. Lawyers interpret and apply case law through processes of analogy. Doctors exploit analogies between cases to make diagnoses and plan treat-ment. Even mathematics and physical science rest on a metaphoric foundation: Computer algorithms search through "trees" or "spaces," using heuristics to "prune branches" or "climb hills." Light moves in "waves," like water or sound. Subatomic particles have "spin," "charm" and "color." Artificial intelligence has developed formal models of metaphor and analogy, and applied them in such areas as case-based reasoning (Kolodner 1993), natural language understanding (Way 1991), inductive learning (Russell 1989) and scientific discovery (Falkenhainer 1990).

An essential problem for any formalization of metaphor or analogy is the selection of a basis or *source* for the comparison. Why did the author of the newspaper article on trade agreements choose a "politics is war" metaphor over a "politics is cooking" metaphor? How do lawyers select the cases on which to build their arguments? How did physicists come to interpret the behavior of light in terms of waves (like sound or water)?

This dissertation examines the problem of selecting an appropriate source for analogical reasoning. Based on a theory of metaphor[1] known as the *interaction theory*, it uncovers several shortcomings of existing retrieval techniques and proposes an alternative approach that addresses these problems. It then implements this retrieval mechanism in a computer program, and evaluates its ability to construct relevant, useful metaphors and analogies.

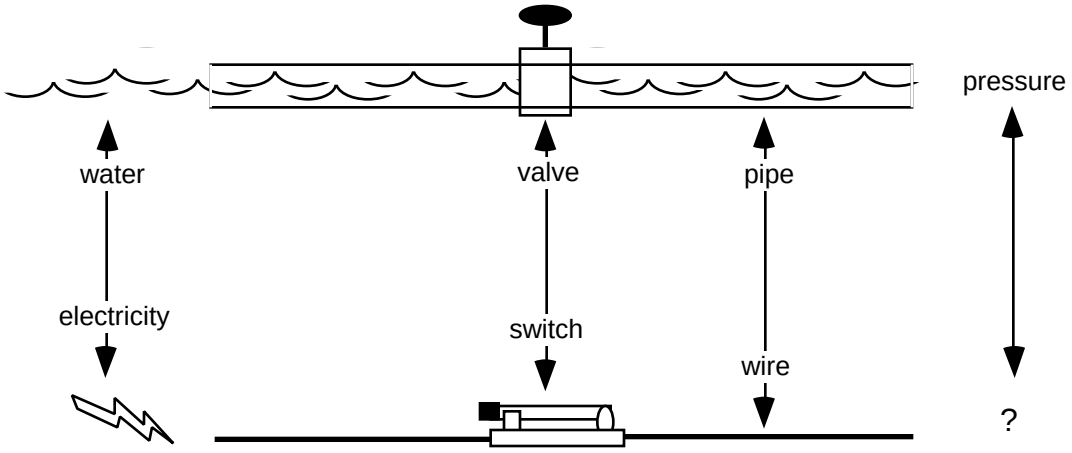
1.1 The standard model of analogical reasoning

Analogical inference refers to any process of reasoning about similarities: if two situations are known to have certain properties in common, we may infer by analogy that they are likely to share additional properties. For example, physics teachers often describe the behavior of electricity through an analogy with the flow of water (figure 1). They begin with a simple comparison between water flowing through a pipe and electricity flowing through a wire, and help their students extend the analogy to include comparisons between voltage and water pressure, amperage and the quantity of water flowing through the pipe, etc.

---

[1] Although the literature regards metaphor and analogy as distinct phenomena, there seems to be little agreement on exactly how they differ. The most common formulation holds that analogies are comparisons of the form: "A is to B as C is to D," while metaphors are direct comparisons of the form "A is B." However, the initial selection of an analogical source requires establishment of a direct comparison between A and C, which seems a lot like a metaphor. Conversely, models of the way in which metaphors effect systems of relations tend to be strongly analogical in flavor, with many authors treating analogy as an underlying mechanism for constructing and reasoning about metaphor (although this idea is somewhat controversial). It is also possible that distinctions between the terms are largely cultural: the term metaphor seems to be most common in research in natural language, while analogy seems to dominate work in reasoning and problem solving.

This dissertation is about analogical reasoning. However, I believe that there are essential, underlying commonalties between these two forms of inference, and I have drawn freely on both literatures in developing my theory of source retrieval.



The water/electricity analogy

Figure 1

Formally, we may think of an analogy as a *mapping*, or set of associations between elements of a *target* domain and those of a *source*. The source of an analogy is a well understood problem, theory or situation: in the above example, a theory of water flow is the source. The target is a problem to be solved, a theory to be extended or a situation to be understood: in this case, the target is the behavior of electricity. Analogical reasoning begins with an initial mapping between elements of the target and source: electricity maps onto water, wires onto pipes, etc. It then extends the mapping to include additional elements of each domain: What is the electrical counterpart of water pressure? Of water flow? The result is a transfer of knowledge from the source to the target.

1.1.1 The structure of analogical reasoning

A general framework for analogical reasoning has been proposed by a variety of authors, including (Hall 1989; Kedar-Cabelli 1988; Wolstencroft 1989). These formulations describe analogical reasoning as consisting of the following steps:

1. An analogy begins with the *retrieval* of an appropriate source. In the water/electricity example, the initial comparison was suggested by the teacher. In the absence of a teacher, analogical reasoners select a source by matching properties of the target with those of candidate sources in an effort to select the most similar. Central issues in source retrieval

include the choice of properties to use in selecting sources, techniques for measuring the similarity between targets and sources, the organization of memory to support efficient retrieval, and the use of learning to improve retrieval over time.

2. The selection of a source establishes an initial mapping between it and elements of the target. *Analogical inference* extends this mapping to include new properties and relations. In the electricity example, an initial mapping might associate water with electricity and pipes with wires; analogical inference leads students to search for an electrical counterpart to water pressure (voltage).

3. Most forms of analogical inference are not guaranteed to be logically sound. It is possible for a student to infer wrongly that electricity may be stored in a wire, like water in a pipe. Consequently, an analogical reasoner must *evaluate* its inferences, using additional knowledge or empirical tests to do so. When an analogical inference proves incorrect, the reasoner may either *repair* the defective aspects of the analogy, or begin again with a new source.

4. Finally, an analogical reasoner should *learn* from this process. The simplest type of learning saves valid extensions to the target domain. More sophisticated approaches also improve retrieval, inference and evaluation.

Nearly all analogical reasoners fit this basic framework.

1.2 An interactionist critique of the standard model

The *comparison theory of metaphor* (Black 1962; Way 1991) makes explicit many of the assumptions underlying the standard computational model of analogical reasoning. The comparison theory takes the common sense position that metaphor is an act of comparing two objects in order to clarify and emphasize similarities between them. Way (1991) characterizes the comparison theory as:

> *. . . relying on some pre-existing similarity between the characteristics possessed by two 'like' objects. These similarities are then made explicit by comparing all the characteristics of the tenor* [target] *and vehicle* [source] . . . (page 34)

According to the comparison theory, metaphors serve primarily as a means of selecting properties in the target; the meaning of those properties is fundamentally independent of the comparison. The semantics of electrical voltage does not depend in any way upon the analogy with water pressure; the analogy only helps to call our attention to it.

In contrast, the *interaction theory* (Black 1962) gives metaphor a fundamental role in establishing the meaning of concepts. The interaction theory views metaphor as a complex interaction between systems of relations in the target and source that can lead to fundamental changes in our understanding of both. The interaction theory contrasts with the comparison view in a number of ways:

1. Under the interaction theory, the meaning of concepts is not fixed, but evolves through their participation in a succession of metaphors. Metaphor can actually cause the meaning of a concept to shift by changing the system of relations surrounding it. Metaphor does not just detect similarities; it can also create them between objects previously held to be dissimilar.

2. The interaction theory treats metaphor as involving entire systems of relations in the target and source, with the relational structure of the source acting as a filter to reorganize our understanding of the target. This contrasts with the comparison view, which suggests that source properties can transfer to the target on a one-at-a-time, individual basis.

3. Whereas the comparison view treats information transfer as moving exclusively to the target, the interaction theory also allows for information to flow back to the source, resulting in changes to both components of the metaphor.

Examination of a common metaphor reveals its fundamentally interactionist structure. Calling a sexist man a "male chauvinist pig," does not build on pre-existing, "objective" similarities between sexists and pigs. Instead, the metaphor creates new notions of similarity that change our understanding of sexism by associating it with other forms of crude behavior. The metaphor does not work at the level of individual properties, but rather transfers a whole system of negative connotations from pigs to sexist men. Further-more, the negative connotations we associate with pigs do not stem directly from pigs themselves, but have evolved from a whole history of comparisons with unpleasant human behaviors. In truth, pigs are highly intelligent creatures. Aside from a fondness for cool mud on hot days, they are not particularly dirty, and their legendary greed for food is really no different from that exhibited by chickens, geese, cattle and other barnyard animals. However, a long history of metaphors comparing pigs to sloppy eaters, greedy businessmen and political extremists of every ideological ilk have transferred a whole constellation of undeserved negative traits to these unfortunate beasts. The history of "human as pig" metaphors is an excellent illustration of the interactionist theory.

The interaction theory raises a number of challenges to the standard com-putational model of analogy. This dissertation focuses on its implications for the problem of source selection, questioning three common assumptions:

1. The separation of retrieval and inference.

2. Monotonic measures of similarity.

3. Memory organization as clustering.

The next three sections discuss these issues in more detail.

1.2.1 The separation of retrieval and inference

The standard model treats source retrieval and inference as separate stages in analogical reasoning: retrieval uses *known* similarities to select an appropriate source, and inference extends the analogy to include additional correspondences. If the interaction view is correct, and metaphors and analogies can alter the semantics of both the source and target, then the separation of retrieval and inference rests on an unresolved circularity: How can an analogical reasoner use similarity to select a source if metaphor and analogy actually create notions of similarity?

Many computational models of source selection ignore this problem by using a restricted set of properties to select sources and assuming that these properties will be known for all sources and targets. These properties com-prise a fixed *retrieval vocabulary*. Retrieval finds sources that have a sufficient number of these properties in common with the target. Analogical inference may transfer additional properties to the target, but it cannot add to, remove from or change the interpretation of this retrieval vocabulary.

This approach leaves source selection excessively reliant on the initial choice of a retrieval vocabulary. It limits the flexibility and adaptability of the retrieval mechanism, and makes it hard to build analogical reasoners in domains like empirical discovery, where too little may be known about the target domain to define an adequate retrieval vocabulary.

In contrast, the interaction theory suggests that analogical reasoning should not assume that all properties in a retrieval vocabulary are known for target problems, nor should it assume that such vocabularies are fixed. Instead, it should allow the analogical reasoner to project relevant properties of promising sources onto target problems *during retrieval*. Beginning with currently recognized similarities between the target and candidate sources, retrieval mechanisms should be able to infer new similarities or re-evaluate existing similarities in choosing among potential sources. Instead of relying on a fixed retrieval vocabulary, this process should be free to use any properties of sources and targets that may lead to the selection of useful analogies.

1.2.2 Monotonic measures of similarity

Most current models of analogy choose sources using the heuristic that the more similar the source and target, the more likely the source is to support useful, relevant analogical inferences. Although this is a reasonable approach to source selection, most analogical reasoners compute similarity as a monotonic function of the number of properties two situations have in common: the more common properties, the greater the similarity.

In holding that metaphors work on entire systems of relations, the interaction theory suggests a greater role for context in determining similarity. The similarity of two situations depends upon complex interactions of goals, background assumptions, inference strategies and interpretations. Monotonic measures of similarity cannot capture these contextual effects. For example, there are situations in which the importance of a property in determining similarity decreases given additional knowledge. If I tell you that two people each weigh 200 pounds, you are likely to infer that they have similar physiques. If I also tell you that they are married to each other and share the biological parentage of a child, you are less likely to make this assumption.

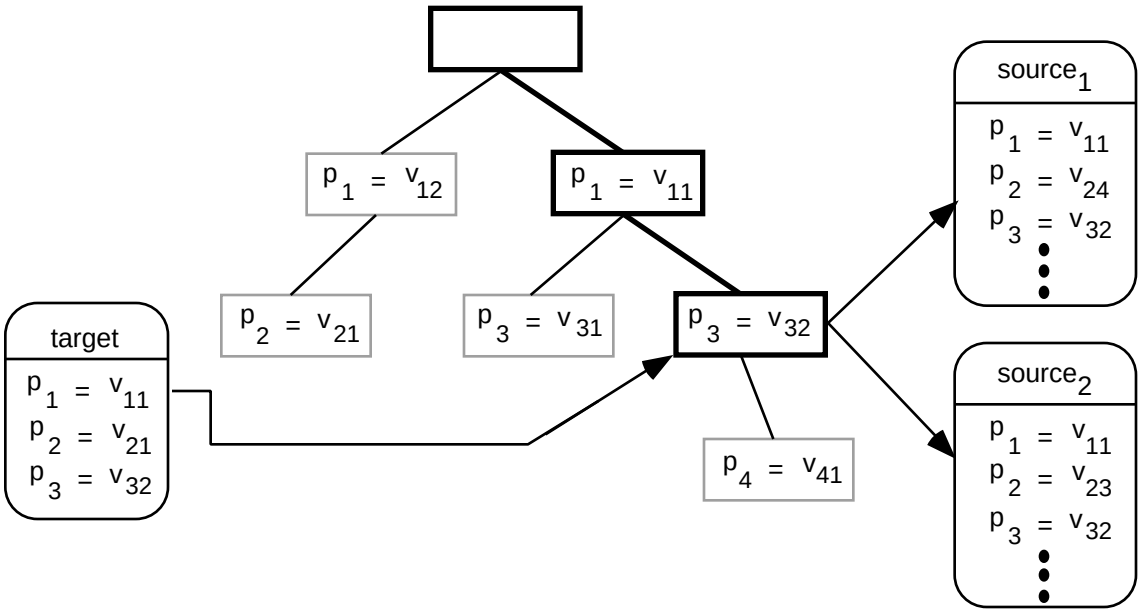
This potential non-monotonicity suggests that similarity should be inferred through qualitative reasoning about target and source interactions in the context of a given target situation. Source selection should take these contextual effects into account.

1.2.3 Memory organization as clustering

Analogical reasoners organize memory to improve the quality and efficiency of source retrieval. This generally takes the form of a hierarchical index system or discrimination network (figure 2) in which each index defines common properties of a set of similar sources. Indices found deeper in the hierarchy add constraints to their parents and, consequently, point to more restricted sets of sources.

Source selection matches target properties with those recorded in indices, retrieving the sources stored under the best match. In figure 2, retrieval has

matched the target with the indicated index node based on their identical values for p_1 and p_3; this leads to the retrieval of $source_1$ and $source_2$. This is more efficient than trying all possible sources and, given an appropriate index hierarchy, can be highly effective in selecting relevant sources.



A hierarchical index

Figure 2

There is a close relationship between index hierarchies and category taxonomies in that each index defines a category of similar sources. Many analogical reasoners use clustering algorithms (Fisher and others 1991; Michalski and Stepp 1983) to create and maintain indices. Clustering algorithms attempt to construct taxonomies that maximize the similarity of items in the same category. Typically, clustering algorithms update the index hierarchy when adding new sources to memory. A common approach searches the existing hierarchy for the best place to insert the new source, modifying the hierarchy if this addition causes any degradation in the quality of the taxonomy.

This source-oriented approach to memory management ignores the reasoner's successes and failures in solving target problems. This works well if sources and

targets can be described using a fixed, bounded number of properties: Clustering finds those properties best able to distinguish sources, and creates indices that provide the same results as an exhaustive search of candidate sources, although in a more efficient fashion. What happens, how-ever, if sources have an unbounded number of given and inferable properties that may potentially be useful in retrieval? Clustering algorithms that require a priori bounds on the properties they must consider in building index hierarchies cannot scale to such a situation.

Here again, the interaction theory suggests a solution: use successful metaphors and analogies to determine which source properties were relevant in past problem solving contexts, and construct indices using these properties. This uses experience to construct a retrieval vocabulary *dynamically*, and shifts the focus of memory management from source-oriented clustering to experience-driven forms of empirical learning.

1.3 An interactionist model of source selection

The above criticisms not only question the cognitive validity of standard computational approaches to source selection, but also raise questions concerning their scalability, generality and usefulness for building analogy-based problem solvers. Such analogical reasoners are excessively reliant on biases in the selection and representation of retrieval vocabularies; they do not work well if these vocabularies are inappropriate, or too little is known about the target to define them. Many application areas, such as learning and discovery, seem to require far greater flexibility then can be provided by traditional approaches.

This dissertation proposes an interactionist model of source selection that addresses the limitations of the standard approach. The major components of this model are:

1. The interleaving of retrieval and inference through *assumption-based retrieval*.

2. The replacement of monotonic similarity measures with *the use of context in determining similarity*.

3. A shift from source-centered, clustering based memory organization strategies to *empirical memory management*.

The next three sections discuss these points in more detail.

1.3.1 Assumption-based retrieval

Instead of treating retrieval and inference as distinct stages in a sequential process, the approach developed in this dissertation interleaves them in a cycle of making and evaluating plausible analogies. Retrieval consists of the steps:

1. Choose an initial set of candidate sources by comparing the target with the nodes in a hierarchical index. Instead of requiring a complete match with the index, match according to the following criteria:

   If a property represented in the index is known for the target, they must match.

   If an index property is unknown for the target, transfer this property to the target and allow the match.

2. Step 1 may produce multiple candidate matches, involving different, potentially contradictory inferences about the target. The second step ranks these initial matches heuristically (see section 1.3.2).

3. Retrieve the sources stored under the index that produced the best match, and construct analogies between them and the target. Each analogy may transfer additional source properties to the target as needed for its solution. If a particular source cannot be extended to include an essential component of the target, eliminate it from consideration and try the next candidate.

4. Evaluate the analogies produced in step 3 empirically. If one of them solves the source problem, quit; otherwise, try the sources stored under the next matching index. Repeat until finding a successful source or exhausting the index matches.

This approach, which I have called *assumption-based retrieval* can improve the quality of source selection in situations where little is known about the target. It also lessens the retrieval mechanism's reliance on a priori restrictions on the set of properties that may be considered in source selection, since it allows indices to use properties that may not be known for the target. Finally, as suggested in the interaction theory, it allows the analogy to create similarities where none previously existed through the inferences made in step 1.

1.3.2 The use of context in evaluating similarity

The above methodology relies heavily on heuristics to constrain initial matches, guide analogical inference and rank candidate sources. We may view these heuristics as measures of target/source similarity. Rather than using simple quantitative similarity metrics, the interactionist view suggests that analogies should be evaluated according to their effects on the overall, systematic structure of the target problem. For example, one of the heuristics used in this dissertation attempts to construct a (partial) solution to the target problem using the knowledge afforded by each candidate analogy, and ranks the sources according to the quality of these partial solutions. This replaces monotonic similarity measures with a systematic approach that evaluates the similarity between targets and sources in the context of potential solutions to the target problem.

1.3.3 Empirical memory management

Rather than updating memory structures when adding new sources, the retrieval system developed in this dissertation updates its indices after it has constructed a successful analogy. It does so by examining the properties that were involved in the analogy to find those that distinguish the successful mapping from the failed candidates. It then constructs a new index that uses these properties to access the successful source[s].

This reduces the need for prior biases on the selection and representation of the retrieval vocabulary. Instead it uses the structure of the target problem to define the relevance of source properties and bound the information that must be considered when updating memory. Note that this approach effects a limited form of bi-

directional transfer as suggested by the interaction theory of metaphor. Learning conveys knowledge of the relevance of various source properties from the target back to the source retrieval mechanism.

1.3.4 Conclusion

This dissertation implements these ideas in a computer program and evaluates its performance on several test domains. In designing and evaluating this program, I have continued to be guided by the specific ideas and general spirit of the interaction theory. In particular, the assumption-based retrieval mechanism described in 1.3.1 allows candidate analogies to create similarity in light of unknown knowledge. If these hypothesized similarities lead to a satisfactory problem solution, they may be saved in the index hierarchy (section 1.3.3) for future use.

In developing the heuristics outlined in 1.3.2, I was guided by the interaction theory's view that metaphors and analogies focus on entire systems of relations. Rather than using monotonic similarity metrics, the heuristics examined in this work evaluate candidate analogies in the context of the whole target problem.

Perhaps the most interesting aspect of the interaction theory is its recognition of bi-directional information transfer. By using the experience gained in solving the target problem to select properties that may be useful in choosing future sources, the memory management scheme described in section 1.3.3 transfers knowledge of the relevance of different properties from the target back to the source. This knowledge assists in the construction of effective index hierarchies.

1.4 The goals and focus of this research

The above discussion outlines an interactionist source retrieval algorithm, and raises several questions that form the heart of this dissertation. These questions include:

1. How can we limit and assure the relevance of the inferences made about the target during assumption-based retrieval?

2. How can systematic properties of candidate analogies be used to measure similarity? In particular, how can a *partial* analogy be evaluated in the context of a target problem?

3. What knowledge representation techniques can improve the efficiency of this retrieval / inference cycle?

4. How can the resulting program learn from its experience? How can it best select properties for inclusion in its index structure?

5. Can we expect features that were relevant to one situation to be effective in choosing a source for a different target?

This dissertation develops a model of source retrieval that provides one set of answers to these questions. It implements this model as a computer program, and tests it on the problem of using analogies to interpret empirical observations. This problem is related to the *case-based interpretation* prob-lem described in (Kolodner 1993). I have chosen it for its inherent challenges, its relevance to a number of specific applications, and the particular demands it places on a retrieval mechanism.

The dissertation evaluates this program / theory along the following dimensions:

1. The effectiveness of the learning algorithm. This evaluation follows the standard machine learning approach of using independent training and test data to test the learning algorithm's ability to generalize acquired knowledge to new situations.

2. Comparisons of the performance of the retrieval algorithm with a more traditional, clustering approach to memory management.

3. Measures of the algorithm's ability to scale as the source base grows.

4. Qualitative evaluations of the quality of analogies produced by the retrieval mechanism.

5. The algorithm's generality. This has been measured by evaluating the algorithm's performance on three different test domains.

1.5 Overview of the dissertation

This dissertation develops and evaluates the ideas put forward in this introduction. Chapter 2 begins with an overview of computational work in metaphor and analogical reasoning. It focuses on the problem of source retrieval, discussing current approaches in more detail and further developing the interactionist critique of standard approaches to analogy.

Chapter 3 defines a computational model of an interactionist source retrieval mechanism. It begins with a discussion of one of the the application areas addressed in this dissertation: the use of metaphor and analogy to interpret empirical observations. It then describes the architecture of SCAVENGER, a computer program that I have written to formalize and test these ideas.

Chapter 4 evaluates SCAVENGER's performance on the problem of interpreting tutorial examples of LISP function behavior. Tests include measures of its efficiency, scalability, correctness and ability to learn. The evaluation also compares SCAVENGER to a more traditional approach to memory organization.

Chapter 5 demonstrates SCAVENGER's application to a diagnostic problem, that of finding bugs in children's basic arithmetic skills. This chapter repeats many of the tests done in chapter 4.

Chapter 6 provides a simple "proof-of-concept" demonstration of SCAVENGER's use in reasoning about physical simulations.

Chapter 7 concludes the dissertation by summarizing the lessons of the SCAVENGER experiments and considering the broader ramifications of this research.

2
Analogical Inference and Source Retrieval

...And I hear it again:
It's in Lu Ji's Wên Fu*, fourth century*
A. D. "Essay on Literature" - in the
Preface: "In making the handle
Of an axe
By cutting wood with an axe
The model is indeed near at hand."

Gary Snyder
Axe Handles

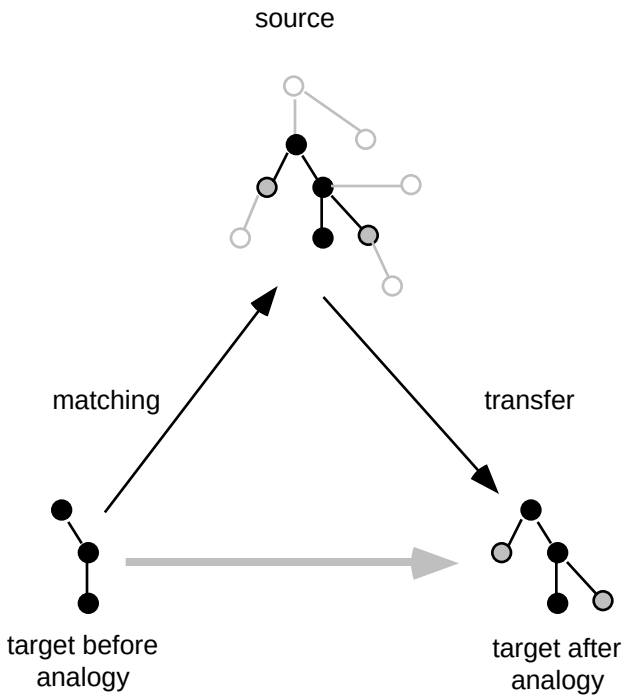
In its most general sense, *analogical reasoning* refers to any process of reasoning about similarities: if two things are known to be similar in certain respects, we may, by analogy, infer additional similarities between them. Analogical reasoning has been applied to a variety of applications including planning (Birnbaum and Collins 1988; Carbonell 1983; Sycara 1988), program synthesis (Amarel 1986; Dershowitz 1983; Williams 1988), problem solving (Kolodner 1993), natural language understanding (Lakoff and Johnson 1980; Lakoff and Turner 1989; Way 1991), machine learning (Falkenhainer 1990a; Falkenhainer and Michalski 1990) and scientific discovery (Shrager and Langley 1990; Thagard 1988; Hesse 1966).

This chapter examines the foundations of the interactionist model of source retrieval and inference[2] outlined in chapter 1. It begins with an overview of analogical inference (section 2.1). This discussion stresses the techniques used to limit analogical inference to plausible, relevant conclusions. Section 2.2 examines current approaches to source selection and memory organization, emphasizing the determination of similarity and the organization of memory to improve the efficiency of retrieval. The final section of the chapter presents the interaction theory of metaphor in greater depth, and uses it to further develop the dissertation's criticisms of conventional retrieval and inference techniques.

[2] Although the evaluation and repair of analogical inferences are important topics, they are outside the central focus of this work. Their treatment in the computer program that tests my theory is straightforward, and I do not discuss them in depth.

2.1 Analogical inference

Analogical inference (figure 3) begins with a *target*; this may be a set of observations to be explained, a problem to be solved, a plan to be completed, an incomplete theory that we would like to extend, etc. The *source* of the analogy is a similar explanation, problem solution, plan or theory. *Analogical inference* transfers additional components of the source (the shaded circles) to the target, attempting to make useful, relevant and supportable inferences.



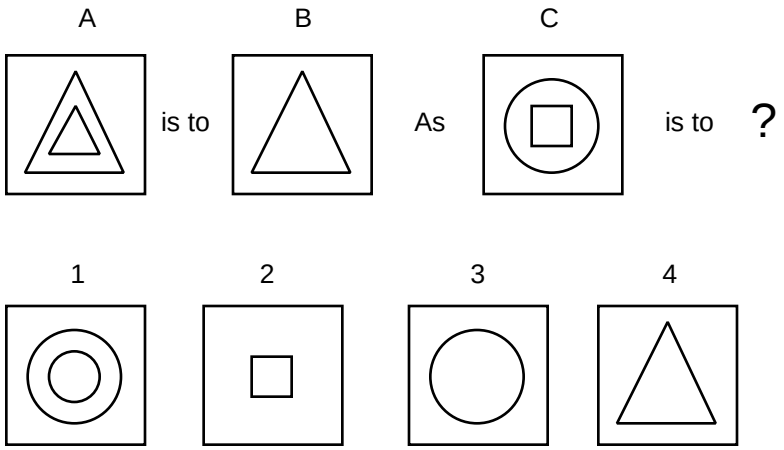
Analogical inference[3]

Figure 3

Evans's (1968) ANALOGY was the first analogical reasoning program, and it introduced many of the ideas that remain central to the field. ANALOGY solved simple geometric analogy problems such as might be found on an intelligence test (figure 4). It solved these problems by determining the series of operations (such as adding

3 Although the figure suggests that sources and targets are represented as graphs, using graph isomorphism as a matching criterion, the ideas discussed in the text apply to other representations.

or deleting components of a figure) needed to transform shape A into B (this sequence of operations becomes the source of the analogy). It then attempts to use this sequence of operations to convert C (the target) into each of the possible answers, selecting the alternative that involves the fewest modifications to the original series of operations (3 is the correct answer in figure 4).



A geometric analogy problem

Figure 4

Evans's work established the viability of automating analogical reasoning, and made use of several ideas that have proven important to the field in general and this research in particular. The most important of these is *systematicity* (section 2.1.2). In choosing the best answer, ANALOGY operates on the entire sequence of steps needed to explain the relationship between A and B. This concern with systems of relations is a central characteristic of analogy and a valuable source of constraints on analogical inference. By focusing on the original "A to B" transformation, ANALOGY also exploits such *pragmatic* constraints as goals and context to guide its inference (section 2.1.3).

2.1.1 Varieties of analogical transfer

An analogy is a *mapping*, or set of correspondences, between the system of relations in the target and those of the source. For example, the analogy between the atom and the solar system might begin with the mapping of figure 5. Analogical inference extends this mapping to include additional properties, such as the

existence of attractive forces in the atom as a counterpart to gravity in the solar system.

Source		Target
solar system	→	atom
sun	→	nucleus
planet	→	electron

An initial analogy between the atom and the solar system

Figure 5

Extending a mapping may lead to the inference of previously unknown target properties and relations. The view of analogies as creating and extending a target/source mapping is extremely general, and can be instantiated in different ways. Kolodner (1988a) proposes a taxonomy of transfer techniques:

1. Transfer the system of relations from the source to the target on an "as is" basis. Here the operators and relations of the target can only map onto identical source elements, but the analogy can match source *objects* with any target object. Essentially, this extends conventional pattern matching algorithms to allow matches between unlike constants. Structure mapping (section 2.1.2) takes this approach.

2. Modify the source as necessary to adapt it to the target. Simple versions modify components of the source, effectively allowing relations and operations to map onto different target elements but leaving the structure of the source unchanged. More sophisticated versions change the relational structure of the source in such ways as adding steps to a plan, reordering operations in a problem solution, etc. Here, the analogical mapping is implicit in the sequence of operations that adapt the source to the target. Evans's ANALOGY performed this type of transfer, as does transformational analogy and similar approaches (section 2.1.5).

3. Transfer the inference method used to solve the source. Here, the problem solver computes a solution to the target, using a source

solution as a guide. This guided reconstruction may be simpler than trying to adapt the source solution or solving the target from scratch. Derivational analogy (Carbonell 1986; Carbonell and Veloso 1988; Minton 1988) exemplifies this approach (section 2.1.3). Although derivational transfer obscures the notion of analogies as mappings, it does establish correspondences between goals, subgoals and initial problem conditions.

4. Create an abstraction that describes both the target and the source, and apply it to the target. Reuse of this generalization can also improve future inferences. Here, the target/source mapping is implicit in the generalization of corresponding objects.

An important characteristic of nearly all forms of analogical inference is its lack of soundness: analogy cannot guarantee the truth of its inferences. Although the ultimate test of validity must be empirical, it is seldom feasible to generate and test all possible analogies. Analogical reasoners use heuristics to produce *plausible* inferences: those with some support in experience or theory. Another property of analogical reasoning is its complexity: Unconstrained analogies can produce an explosion of potential inferences. The design of heuristics to assure the tractability and plausibility of analogical inference is a central problem for theories of analogical reasoning. The remainder of this section discusses a number of heuristics that are commonly used to constrain analogical inference. These are:

1. The preservation of the systematic structure of the source.

2. The use of pragmatic concerns to select analogical inferences that will be likely to be useful in solving the target.

3. Type constraints on analogical mappings.

4. The use of heuristic operators to modify sources to fit the target.

5. The use of learned abstractions to constrain future analogies to fit patterns that have proven successful in the past.

2.1.2 Systematicity and analogical inference

An essential strength of metaphors and analogies is their ability to transfer whole systems of relations from the source to the target in a single comparison. Lakoff and Johnson (1980) point out that the purpose of metaphor is to convey the global structure or gestalt of a domain: when we use the meta-phor, "argument is war," we transfer the systematic structure of war to the process of argument. Metaphors and analogies play a similar role in establishing a conceptual framework for scientific discovery (Hesse 1966; Rothbart 1988). Systematicity is central to the evocative power of linguistic metaphors. It is widely discussed in the literature on metaphor and analogy (Falkenhainer 1990a; Falkenhainer and others 1989; Gentner 1983b; Gentner and Stevens 1983; Helman 1988; Lakoff and Johnson 1980; Lakoff and Turner 1989; Vosniadou and Ortony 1989). The interaction theory of metaphor focuses heavily on systematicity (section 2.3).

The systematic structure of analogies not only offers expressive benefits, but also can be a source of contextual constraints on analogical inference. *Structure mapping* theory (Falkenhainer and others 1989; Gentner 1983a; Gentner 1983b) is one of the earliest and clearest examples of the use of systematicity to constrain analogical inference.

High-quality analogies transfer richly connected systems of relations from the source to the target. For example, an analogy such as "the atom is like the solar system," transfers a rich system of causal relations to the target; this richness is what makes it useful in explaining the target domain. In contrast, the comparison "this flower is like the sun" would generally be taken to communicate superficial similarities: the flower, like the sun, is round, yellow and associated with summer. Structure-mapping theory constructs high-quality analogies in an efficient manner by focusing on those components of the source that are most indicative of its semantic structure. It implements this heuristic through a set of syntactic biases on representation and inference. These include:

1. A preference for mappings that transfer relationships over those involving properties of objects. Structure mapping formalizes this as a distinction between unary predicates (properties) and predicates of a

higher arity (relations). In the atom/solar system example of the previous section, relations such as larger-than(sun, earth) and orbits-around(sun, earth) would more likely be useful in constructing an explanation of observed behavior, and are preferred over unary predicates like hot(sun).

2. Preservation of relations. If a relation exists between two objects in the source, then infer the same relation between corresponding objects in the target:

orbits-around(sun, earth) -> orbits-around(nucleus, electron)

3. Systematicity. Higher-order relations (relations that take other relational sentences as their arguments) have a preference in the mapping. Causes is typical of higher-order relations. For example, analogical inference would focus on the source rule:

causes(larger-than(sun, earth), orbits-around(sun, earth))

leading to the inference:

causes(larger-than(nucleus, electron), orbits-around(nucleus, electron))

Winston (1980; 1986) has also explored the use of explicitly represented systematic constraints on analogical inference through a focus on causal relationships in constructing mappings.

Discussion: Syntactic approaches to systematicity

Systematicity, through its use of contextual, structural information about the source (and target), is a powerful source of heuristics to constrain analogical inference. However, many attempts to formalize systematicity constraints are excessively rigid and reliant on syntactic biases. In the previous example, structure mapping eliminated hot(sun) from consideration under the heuristic of favoring relations of arity greater then 1. On the face of it, this is a reasonable interpretation of systematicity; unfortunately, it is excessively dependent on a priori representational biases and leaves critical issues unexplained. For instance, if we had represented this property as temperature(sun, hot), structure mapping could produce different

inferences. Also, while the property hot(sun) is not relevant to descriptions of the structure of the atom, large(nucleus) is. Structure mapping theory does not specify an adequate criterion for selecting one representation over another.

A theory of analogy should not rely on unexplained representational biases, but should produce the same results for any reasonable representation of target and source. By "reasonable," I mean that the representation should be sufficiently expressive to capture any semantically important distinctions: an analogical reasoner cannot transfer causal relationships to the target if the representation language does not include a notion of causality. Systematicity constraints should not be encoded in a rigid syntactic fashion but should be derived from problem solving contexts or past experience. The other approaches considered in this section explore these alternatives.

2.1.3 Pragmatic approaches

The discussion of analogical reasoning in (Holland et al. 1986) criticizes structure mapping for failing to account for goals and other contextual constraints in guiding analogical inference. Although acknowledging the importance of systematicity, Holland et al. are critical of attempts to encode it in the syntax of a representation, arguing that this ignores these pragmatic concerns.

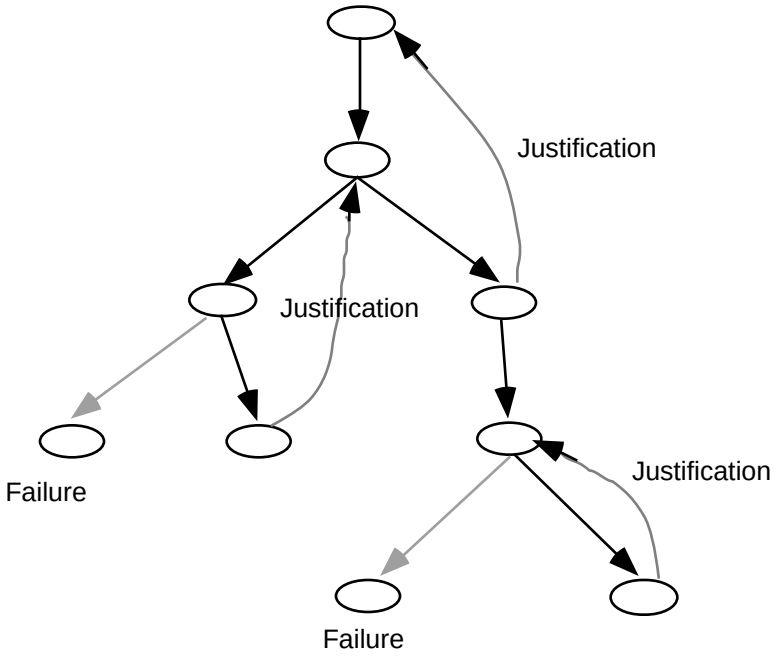
A common way of exploiting pragmatic constraints is to transfer the goal of the target problem to the source domain, solve the goal for the source, and transfer the solution back to the target. An alternative approach selects a previously generated source solution by matching on goals. This focus on goals selects sources that may be useful in solving the target problem.

On the face of it, pragmatic approaches would seem to select sources with greater relevance to the target. However, proponents of structure mapping counter that goal-based constraints are just higher-order relations that can be syntactically encoded in a representation. Novik (1988) further examines this debate.

Kedar-Cabelli (1988) formalizes the pragmatic approach, integrating EBL and analogical reasoning in her purpose-directed analogy. In explaining an observation in a target domain, purpose-directed analogy first constructs an explanation of a

similar instance in a source domain; transfer then operates on this explanation. Greiner (1988a) incorporates a form of pragmatic constraints in his requirement that analogies be useful. An analogy is a useful extension to a theory if there exists a goal that the theory cannot prove, but that can be proved by the union of the theory and its analogical extensions.

Carbonell has argued that analogy should consider the reasons why decisions were made in solving the source problem (Carbonell 1986; Carbonell and Veloso 1988). *Derivational analogy* treats source solutions as sources of strategic information. This requires that each node in the source be annotated with the justification for applying its chosen operator, a description of the initial portions of paths that were tried and failed, and the reasons for these failures (figure 6). Construction of a solution in a target domain proceeds stepwise, first mapping the start state of the source solution onto the start state of the target problem. At each stage, the decision made in the source, along with these justifications, provides guidance for the next step in constructing a solution of the target. Derivational analogy applies a strong form of pragmatic constraints to analogical transfer.



An annotated source used in derivational analogy

Figure 6

Case-based reasoning (CBR) is a variant of analogical reasoning that is widely used in problem solving, planning and expert systems (Hammond 1989b; Kolodner 1988; Kolodner 1993). A case-based reasoner solves new problems by adapting and reusing solutions to previously solved problems. CBR differs from analogical reasoning in two ways: One is its concern with *instances* of a problem solution, rather than more general knowledge. Case-based reasoners also tend to work within a single domain, where analogies often cross domains.[4] Case-based reasoners often exploit pragmatic constraints. One reason for this is their frequent emphasis on problem solving; another is the fact that it is straightforward to match target and source goals if transfer operates within a single domain. Case-based reasoners exploit both systematic and pragmatic constraints through their focus on entire source cases in analogical transfer, and their reliance on target goals in source selection. Schank's (1982) dynamic memory theory places heavy emphasis on the use of goals in source retrieval, and many CBR systems build explicitly on this theory (Kolodner 1993; Kolodner 1983). Section 2.2 further discusses the use of goals and context in retrieval.

Discussion: Goals and relevance

Through their focus on goals in developing an analogy, pragmatic approaches add relevance criteria to the structural constraints exploited by structure mapping. It is telling that so many successful efforts to exploit goal-based constraints have been in case-based reasoning. Because CBR tends to work within a single domain, it is more feasible to retrieve a source by matching on goals. Cross-domain analogies require more complex determinations of similarity, and may have a harder time choosing a relevant source.

There are other ways of implementing pragmatic constraints. Section 2.2.1, for example, discusses two systems (MEDIATOR and HYPO) that use broader contextual knowledge (in addition to goals) to determine source relevance. This dissertation explores yet another approach to determining the relevance of analogies. The problem of interpreting empirical data, which is addressed in this research, does not

---

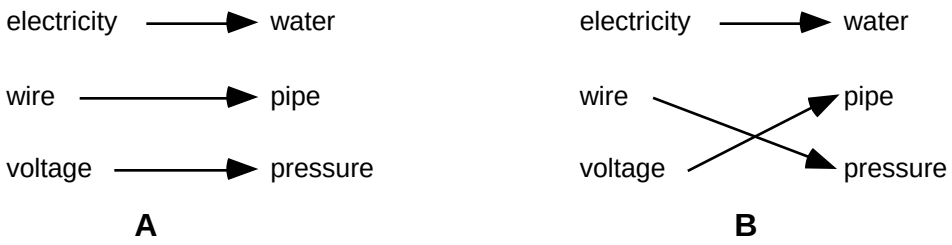
[4] Is analogical reasoning is simply a higher-order form of CBR?

offer a single, clearly defined goal. Instead, the "goal" of an analogy is a high-quality explanation of a set of target observations. This replaces a focus on explicit goals with a preference for analogies that yield simple, unifying and empirically verifiable explanations of target phenomena. One of the central issues explored in this work is the problem of combining empirical validation, qualitative heuristics (such as a preference for simple answers), and goal-centered strategies in the selection of analogical sources.

2.1.4 Type Constraints on Analogical Mappings

There is a strong relationship between categorization and analogy in that both depend upon similarity: Analogies transfer knowledge between similar objects, while categories gather similar objects together. It is reasonable that analogies should use taxonomic information to determine similarity.

For example, assume we are exploring an analogy between electricity and water. Mapping A of figure 7 is clearly correct. A plausible criterion for eliminating mapping B is that it associates a quantitative value (voltage) with a non-numeric object (pipe).



Alternative analogical mappings

Figure 7

We may implement this constraint by favoring mappings between objects that have a common super-type low in a type hierarchy. The closer the common parent, the more similar are the objects, and the more plausible is the mapping.

Way (1991) holds that the meaning of a metaphor is found in the most specific generalization of the metaphor's components. For instance, the meaning of the metaphor "Bubba is a pig," results from the fact that both Bubba and pigs belong to the category of sloppy eaters. She addresses the problems of tangled hierarchies by

using a form of pragmatic constraints: the context of the metaphor acts as a filter to select a relevant portion of the type structure.

There is a strong two-way relationship between analogy and categorization. Turner (1988) has pointed out that the same notions of similarity underlie both phenomena. We do not recognize similarities between elements of a category as analogical because the category structure reifies the comparison: what we recognize as an analogy is an expression of *previously unrecognized* similarities between members of different categories. Because it redefines category boundaries, analogy leads to a new understanding of the target situation. When an analogy, such as "argument is like war," is used often enough, it eventually forms a new category of experience whose analogical origins are forgotten.

Discussion: Types and context

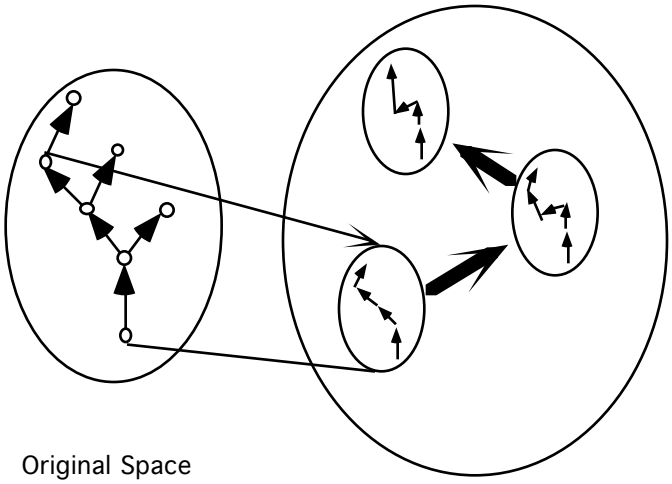
The use of type information in constraining analogical inference reflects the view of class taxonomies as highly compressed representations of more complex, systematic notions of similarity. There are, however, a number of difficulties in exploiting taxonomic information for analogies. One of these is accommodating notions of context in using types. Way (1991) addressed this problem by using problem specific information to select portions of a complex, tangled hierarchy for use in interpreting metaphors.

Another problem is the chicken-and-egg relationship of categorization and analogy. Categories capture notions of similarity underlying analogies, and analogies lead to the formation of new categories. How can we implement this interaction formally? Machine learning techniques have the potential to mediate this interaction by building taxonomies that reflect successful analogies; this is the approach taken in this dissertation (chapter 3).

2.1.5 Operator based source adaptation

A powerful form of analogical inference uses transformation operators to adapt a source to a target. Here, the analogical mapping is implicit in the sequence of modifications performed on the source. *Transformational analogy* (Carbonell 1983) is one of the first programs to solve new problems by adapting successful solutions

to similar source problems. Operators modify solutions in such ways as inserting or deleting steps, re-ordering steps, splicing new solutions into the source, changing the bindings of parameters, etc. These operations define a space (T-space) in which states are entire problem solutions (figure 8). Transformational analogy searches this space, using means-ends analysis to choose the transformation that moves the solution closer to the target problem. This work exerted a strong influence on the development of analogical reasoning by showing how complex problem transformations could be implemented as search, and how weak heuristics, such as means-ends analysis, could be applied to analogical reasoning.



Analogical reasoning as search through T-space (Carbonell 1983)

Figure 8

Case-based reasoners often rely upon rich sets of case-adaptation operators to apply source cases to new problems. Kolodner (1993) proposes a taxonomy of transformations; these include:

1. Substitution methods. These are operators for replacing components of a source solution with items from a target problem. They include general operators for re-instantiating sources, rules for adjusting values of numeric parameters, and similarity-based substitution rules such as

allowing an object to be replaced with an object that is close to it in a semantic network.

2. Transformation methods that make changes in the structure of a source solution. "Commonsense" transformation operators perform weak, syntactic modifications on a source, such as adding, deleting or re-ordering operations (transformational analogy uses such operators). This contrasts with methods that exploit domain specific adaptation and repair operators. Model-guided repair is the strongest transformational method, reasoning with a causal model of the target domain to adapt sources to targets.

3. Derivational replay is a variation on the derivational analogy method of section 2.1.3.

Different case-based reasoning systems provide variations on these basic themes. See Kolodner (1993) for a more thorough discussion.

There are a number of systems which, while not strictly analogical in nature, have explored the use of transformations on problem solutions. AbE (O'Rorke et al 1990) and COAST (Rajemony 1990) have applied the transformational approach to the theory-formation problem; working in the domain of qualitative process theory (Forbus 1984), a general language for describing the qualitative changes and interactions of physical processes.

The PRODIGY learning system uses *operator refinement* to repair incomplete theories in planning domains (Carbonell and Gil 1990). Operator refinement analyzes failed plans or plans that suffer from inefficiencies or goal conflicts due to poor ordering of operations. It repairs the theory by acquiring new pre- and post-conditions on misapplied operators. Where it detects multiple possible modifications, it plans experiments to select the best of these.

Mitchell's (1993) Copycat system takes a novel approach to operator based analogy. Copycat interprets analogies between letter strings, solving problems of the form **abc -> abd** as **ijk** -> ?. The program constructs solutions through a large number of applications of extremely fine-grained operators. The application of

relevant operators is partly governed by various stochastic values that cause the algorithm to vary its solution strategies across repeated trials on identical problem instances. This approach makes Copycat's behavior flexible, and capable of human-like variations in problem solving strategy and the solutions it finds.

Discussion: Domain specificity in operator-based approaches

Although operator-based methods are a powerful form of analogical reasoning, they are hard to apply to cross-domain analogies, since even seemingly weak operator-based methods make domain specific assumptions. For instance, PRODIGY's operator refinement method is specific to planning problems. AbE and COAST's operations work with qualitative process theory, which incorporates assumptions about qualitative physics. Wilkins (1990) uses meta-operators to propose modifications to an incomplete, inconsistent or incorrect theory, noting that while these meta-operators do not incorporate domain-level semantics, they are specialized to a single class of domains: those performing *heuristic classification* (Clancy 1985). Copycat encodes domain specific assumptions in its "slipnet" a large semantic net-like representation of program knowledge.

This reliance on domain knowledge is both a source of the power of operator-based approaches, and limitation on their generality as a theory of analogy. Machine learning may offer a means of bridging this gap by explaining the empirical origins of such operators. The next section examines the role of learning and abstraction in analogy.

2.1.6 Analogy and Abstraction

Perhaps the most powerful source of constraints on analogical inference is the generalization of patterns of successful analogy. In a series of experiments on humans, (Gick and Holyoak 1983; Holyoak 1984; Holyoak 1985) discovered that an abstract understanding of significant features of an analogy improves its detection and use. Gick and Holyoak conjectured that humans develop abstract schemas for classes of similar problems which help them find and use productive analogies. Subsequent experiments corroborated this conjecture. The use of abstractions in

analogy has been formalized in several ways, including the use of meta-logics and schemas.

Russell and Davies offer a formal model of abstractions in analogy in their theory of *determinations* (Davies 1988; Davies and Russell 1987; Russell 1988; Russell 1989). Determinations represent knowledge of relevance as a form of functional dependency. For example, if I know that Pierre comes from France and speaks French, I may conclude that since Catherine is also from France, she will speak French as well. How did I know that I could use the analogy between Pierre and Catherine's country of origin to infer her native language, while ignoring the difference in their gender? Davies and Russell argue that analogies require prior knowledge of the relevance of problem features to specific goals: country of origin determines language, but gender does not.

A determination is a logical schema that specifies the relevance of certain features to others. Formally, the knowledge that p determines q is written (underlined variables indicate sets of variables):

$$p(\underline{x}, \underline{y}) \succ q(\underline{x}, \underline{z})$$

and is defined as:

$$p(\underline{x}, \underline{y}) \succ q(\underline{x}, \underline{z}) \text{ iff}$$
$$\underline{w}, \underline{x} \, [[\quad \underline{y} \; p(\underline{w}, \underline{y}) \quad p(\underline{x}, \underline{y}) \qquad \underline{z} \, [\, q(\underline{w}, \underline{z}) \Leftrightarrow q(\underline{x}, \underline{z}) \,]]$$

In defining the form of object-level inferences, determinations represent meta-level constraints on allowable analogical inferences. This allows the logically sound inference:

$$[p(\underline{x},\underline{y}) \succ q(\underline{x},\underline{z})] \quad p(\underline{s}, \underline{a}) \quad p(\underline{t}, \underline{a}) \quad q(\underline{s}, \underline{b}) \quad q(\underline{t}, \underline{b})$$

For example, assume that we have the prior knowledge that nationality determines native language:

$$\text{nationality}(x, w) \succ \text{native\_language}(x, z)$$

Assume we know that Pierre is from France and speaks French, and that Catherine is from France:

nationality(pierre, france)
native_language(pierre, french)
nationality(catherine, france)

By expanding the determination and making appropriate substitutions we obtain:

nationality(pierre, france) nationality(catherine, france)
[native_language(pierre, french) ⇔
native_language(catherine, french)]

This leads to the inference: native_language(catherine, french).

Determinations are abstract constraints on analogical inference. They also provide a computational perspective on the use of background knowledge in induction proposed in (Goodman 1954). Ironically, the primary problem with determinations is their goal of providing a logically sound model of analogy. This excessively restricts analogical reasoning: To be effective in many domains, analogies do not need to be logically sound, they need only to be an effective source of plausible, testable and (if needed) repairable conjectures.

A number of researchers have taken abstraction-based approaches to analogy. Greiner's (1988a; 1988b) work in abstraction-based analogy uses abstract specifications of classes of laws to mediate construction of an analogy between instances of the abstraction. For example, an abstract schema for flow theories may mediate future analogies between hydraulic systems and electrical circuits. MEDIATOR (Kolodner and others 1985) is a case-based reasoner for resolving disputes. It organizes source cases under generalized episodes, abstract descriptions of classes of similar source cases. The generalized episode describes the common structure of the sources; pointers to individual sources describe differences between them and the generalization. Mediating the mapping through the generalized episode provides strong constraints on inference. Note that the frame structure of generalized episodes also implements a form of systematicity constraints.

Shrager's work with views showed how abstractions of concepts may be useful in constructing domain theories (Shrager 1987; Shrager and Langley 1990). Shrager demonstrated their use on the problem of discovering how to program a Big-Trak. The Big-Trak is a programmable toy tank that allows the user to enter sequences of

instructions, such as go-forward, turn, stop, etc. A program called IE constructed a model of the Big-Trak's behavior by instantiating and combining views: these are abstract definitions of computer concepts including memory, stack manipulation, clock, etc. IE selects relevant views and integrates them into the theory by instantiating them with elements of the problem domain.

Discussion: Abstraction and learning

Abstractions mediate analogy by providing templates for a preferred mapping; this improves efficiency and eliminates many erroneous analogies. A plausible form of learning in an analogical reasoner is to store abstract patterns of analogy, solving future problems through instantiation of appropriate abstractions, rather than construction of a new analogy.

The acquisition of useful abstractions raises a number of interesting questions. Abstractions of successful analogies must capture their systematic structure: What syntactic operations can be used to assemble appropriate structures? How can a learner determine which elements of an analogy to leave out in constructing an abstraction? What heuristics can help it to construct abstractions that will be of use in future analogies? How can such abstractions be integrated into a source retrieval system? The learning mechanism of chapter 3 addresses these problems.

2.1.7 Summary and conclusion

This section examined a variety of approaches to analogical inference. Although these techniques differ in their specific formulations, a set of common themes run through nearly all models of analogy.

Systematicity is one of the most important characteristics of analogical reasoning: A single comparison conveys an entire system of assumptions and relations from the source to the target. Systematicity permeates nearly all approaches to analogy: Structure mapping favors higher order predicates in constructing a mapping; MEDIATOR uses frame like structures to guide inference; other approaches, such as purpose-directed analogy, transformational analogy and many forms of case-based reasoning implement systematicity constraints by analogizing on entire explanations or problem solutions.

The pragmatic criticism of structure mapping theory is less a rejection of systematicity then a recognition that purely syntactic approaches cannot account for context-sensitive aspects of problem structure: analogies operate on systems of relations, but the structure of those systems depends upon a problem solving context.

Systematicity underlies other constraints on analogy: type hierarchies represent, in a shorthand manner, families of similar objects. While the underlying justification of the similarities underlying them might be a complex, systematic explanation, the taxonomy conceals this rationale and allows a reasoner to apply it in a highly efficient manner. Transformation operators also encode more complex, systematic constraints implicitly.

The last type of constraint examined, abstractions of common patterns of analogies, synthesizes systematic and pragmatic constraints. Abstractions represent systems of operators or relations as derived from problem solving experience.

The SCAVENGER algorithm (chapter 3) exploits all of these constraints in its approaches to both retrieval and inference.

2.2 Source selection and Memory Organization

The selection of an appropriate source is a central problem for analogical reasoners. Since it deals with similarity instead of identity, analogy cannot rely upon straightforward pattern matching algorithms such as unification. In the most general sense, similarity depends upon influences as diverse as the representation of source and target problems, the past experience of the reasoner, and the goals and assumptions of the target problem. Since targets are, by definition, incompletely understood, retrieval must work with partial knowledge. Retrieval can be made more efficient by a memory organization that supports detection of similarities, but any such organization must deal with these complexities. Source selection raises a number of problems:

1. Intuitively, similarity would seem to depend upon the number of features two situations have in common, but how do we account for the fact that some features may be more important than others? How

do we take the goals and context of the target problem into account in interpreting similarity?

2. A retrieval mechanism cannot consider all properties of the source and target in computing similarity; how can it select an appropriate subset of properties to consider in a given problem? Is it better off using more easily evaluated, but weakly predictive properties, or should it pay the cost of managing a richer but more complex retrieval vocabulary? Are we justified in efforts to "hard wire" a retrieval vocabulary in an analogical reasoner, or is the selection of appropriate properties inherently dynamic and context dependent?

3. What role does the category structure of source and target domains play in retrieval? Conversely, what role do analogies play in the formation of taxonomies?

4. While similarity underlies source selection, evaluating a target's similarity to every possible source is not computationally efficient. Analogical and case-based reasoners generally use indexing systems to access candidate sources more efficiently. How can a retrieval mechanism best select and represent indices? How can it adapt these memory structures to improve performance over time?

The remainder of this section examines these issues in more detail.

2.2.1 Retrieval vocabulary

For reasons of efficiency, retrieval cannot afford to consider all properties of the target and source problems during retrieval. In realistic domains, objects can have an unbounded number of known or inferable properties. As stated in (Kolodner 1993):

> *. . . a case's indexes are a subset of the case's description or representation. Thus, index vocabulary is a subset of the vocabulary used for full symbolic representations of cases.* (page 196)

Although some restrictions on a retrieval vocabulary are necessary, in many analogical reasoners, the set of properties that can be used for source retrieval is initially defined by the system's designers, and does not change through use. It is reasonable to criticize such systems for their reliance on biases embedded in the retrieval vocabulary.

Many theories of analogy justify this approach as in terms of a distinction between *surface* and *deep* knowledge (Gentner 1989; Ross 1989). Surface features, often taken to correspond to direct perceptions, are readily accessible properties used for selecting source cases. Deep knowledge reflects the more complex, systematic structure of a domain; this structure is often implicit, and only becomes apparent when knowledge is applied to actual problem solving. For example, an explanation or proof of a theorem makes a portion of the deep structure of a theory explicit; a plan reveals part of the deep structure of a planning domain.

There is evidence that humans use more readily accessed, perceptual features of target problems for source retrieval (Gentner 1983b; Gentner 1989). However, while surface-deep distinctions may be meaningful in specific situations, they do not reflect a global, a priori division of knowledge. Such distinctions are highly context-specific, depending upon the background knowledge, goals and assumptions currently available to the reasoner. Uemov (1970) has criticized a priori distinctions between deep and surface knowledge , stating:

> *When an analogy is successful it is called "deep", "strict", "scientific" etc. When it does not work it is called "superficial", "loose", "unscientific" etc. Naturally there is a question as to how those two types of analogy can be distinguished before the practical realization takes place. What are the criteria which permit this distinction?*

Although surface features are generally associated with perception, perception requires interpretation. Different people see different things in the same situation. For instance, if an experienced user sees a new computer system, their "surface features" might include observations of the type of operating system, user interface or mass storage; someone who was not computer-literate might initially notice the size and color of the monitor. Experts develop a more selective and informed notion of surface features. Hammond (1989a) argues that an analogical reasoner should index sources under functional information about their role in problem

solving, rather than on the basis of a priori distinctions between deep and surface knowledge. Gick and Holyoak's work on human analogy supports this insight (Gick and Holyoak 1983; Holyoak 1984; Holyoak 1985). After being exposed to multiple analogies and being encouraged to abstract out their common features, subjects become better at applying the sources to future problems. This implies that these abstractions of "deep" structure become easily accessible in memory and serve as retrieval cues for subsequent analogies.

The inadequacy of a priori deep / surface distinctions is even more evident in light of the use of analogies to transfer knowledge across problem domains (such as the water / electricity example of chapter 1). There is no strong likelihood that different domains will share common surface features: source selection must exploit the semantic structure of both the target and source. Birnbaum and Collins (1988) argue that cross-domain retrieval is enhanced by acquiring abstract structural information, and using this to index cases; they give the example of using abstract categories of strategies to enable the transfer of strategies from one game, such as chess, to another, such as baseball. Seifert (1988) suggests that cross-domain source retrieval should use abstractions of target goals for retrieval. ACCEPTOR (Leake 1991) applies case-based reasoning to the problem of detecting anomalies in stories; it uses a rich vocabulary of story failure types to index source stories. This taxonomy of failures reflects deep knowledge of the problem domain.

Discussion: The problem of fixed retrieval vocabularies

A retrieval vocabulary cannot be defined in an a priori fashion, but must be learned through experience. Retrieval mechanisms should be constantly on the lookout for properties that can be used to select relevant sources, and add these to its index hierarchies. Instead of relying upon priori syntactic biases, source selection must be able to consider any knowledge, whether represented as relational, systematic knowledge or as simple features.

Several systems have addressed this problem. For instance, MEDIATOR (Kolodner and others 1985) organizes instances under generalized episodes: these are frame structures that describe the organization of a class of cases. Thompson and Langley (1991) have further explored the use of structured representations in

concept formation. In an approach that is related to the retrieval mechanism described in chapter 3, HYPO (Ashley 1988; Ashley and Rissland 1988a; Ashley and Rissland 1988b) integrates retrieval and inference in a process of explicitly reasoning about the relevance of candidate sources.

These observations raise a number of questions that are central to this research:

1. What are the criteria for acquiring and changing the retrieval vocabulary?

2. How can newly discovered retrieval cues be integrated into an indexing system?

3. Many highly predictive source-properties may not be among the information given in the statement of target problems. Such properties may be the product of extensive inference in the source domain, and may not be known for a poorly understood target. Are there benefits to indexing sources under properties that may not be known for target problems? How can such properties be used in retrieval?

2.2.2 Numeric measures of similarity

There is no way to guarantee that a source will be relevant, complete and correct in solving a target problem, except to use it to solve the target, and evaluate the results. However, analogical reasoners limit the sources so considered under the heuristic that sources that share known similarities to the target have a strong likelihood of sharing additional similarities.

A reasonable definition takes the similarity of two objects to be a function of the number of features they have in common. A simple approach represents objects as attribute vectors; these are sequences of attribute / value pairs: $[a_1=v_1, a_2=v_2 \ldots a_n = v_n]$. Each attribute vector designates a point in an n-dimensional attribute space; the closer two points are in this space, the greater their similarity. The distance between two attribute vectors is a function of the individual distances between their values for each attribute: For numeric attributes the distance between two values is simply

the absolute value of their arithmetic difference. A common measure of the distance between qualitative attributes gives a value of 1 if the values are different, 0 otherwise.[5] Given the distances between individual attributes, we may compute the overall distance between two attribute vectors. An obvious approach computes this as the Euclidean distance in attribute space. A simpler method sums the distances between individual attributes. Niiniluoto (1988) has measured similarity as $k/(k+m)$ where k is the number of features two objects have in common and m is the number on which they differ.

Numeric measures of similarity work adequately for simple domains but are limited in their ability to address several important issues. One of these concerns the salience of features. All features are not equally important; some may be more predictive of a source's usefulness than others. A simple model of salience measures total similarity as the weighted sum of the similarities of individual features, giving higher weights to more salient features.

Discussion: Similarity and context

Although numeric models of similarity have grown quite sophisticated, there are limits to what can be done without using qualitative reasoning. In particular, defining similarity to be a function of the number of properties objects have in common does not extend to domains in which objects can have an unbounded number of properties (Watanabe 1969). Using such similarity metrics requires that the universe of possible properties be bounded by some notion of relevance, usually to the goals of the target problem. This implies that similarity measures must depend upon some form of qualitative inference concerning the relevance of various properties. However, in implementing these inferences, many analogical reasoners rely upon a priori representational bias to bound properties that must be considered. One of the primary goals of this research is to explore the use of empirical learning to define a retrieval vocabulary dynamically.

---

[5] Approaches that combine numeric and qualitative attributes usually normalize the distances between numeric attributes to a range between 0 and 1.

2.2.3 Qualitative approaches to inferring similarity

Several systems use domain-knowledge to construct a qualitative explanation of why things are similar. MEDIATOR (Kolodner and others 1985) uses generalized episodes, abstract descriptions of a family of cases, as indices in source retrieval. Individual cases are associated with the appropriate generalized episode, along with a record of any differences between the case and the generalization. This approach, influenced by a theory of human memory organization (Schank 1982), incorporates goal and contextual knowledge in case selection. Learning creates new generalizations to represent similarities between successful cases. On failure, MEDIATOR explains the reasons for the failure, and uses these explanations to find features that discriminate failed and successful sources. It then adds these properties to a new generalized episode.

HYPO (Ashley 1988; Ashley and Rissland 1988a; Ashley and Rissland 1988b) integrates analogical retrieval, reasoning and justification in a process of reasoning about the similarity of cases. HYPO is a case-based reasoner working in the domain of trade secret law. It maintains a library of legal precedents, indexing cases by a set of dimensions. A dimension is one way of arguing about a case: it identifies features of the case that can define similarities and differences with a new problem in the context of a particular argument. In effect, a dimension defines the salience of features in the context of a specific argument. HYPO retrieves precedent cases by matching dimensions of the new problem with dimensions of precedents. Since a dimension can match a case partially, HYPO organizes these matches into a claim lattice, a partial ordering of the "pointedness" of precedent cases: the greater the number of features matching a dimension, the more on point is the precedent. HYPO reasons with this lattice to support its case and rebut counter arguments.

PI (Thagard 1988) uses *spreading activation* through a network of concepts to construct analogies. Spreading activation is a strategy for searching an associative memory, such as a semantic network. Beginning with a goal concept, "activation" spreads from concept to concept along associations, searching the network in a generally breadth-first fashion. Given a target concept that the system would like to explain, PI searches outward from that concept through the links of its semantic

network. Activation spreads through the network until it finds a useful source: the closer the source, the greater the similarity. In an example given in (Thagard 1988), PI might propose a wave theory of sound by tracing associations in a semantic network:

sound -> music -> stringed instrument -> vibration -> up and down motion -> wave

Protos (Bareiss 1989) and GREBE (Branting 1989) employ a similar approach in using an explicit theory to construct an explanation of why domain items are similar.

Many instantiations of the spreading activation model of analogy suffer from an excessive reliance on deterministic search through fixed networks of associations. The Copycat program (Mitchell 1993) addresses these problems by allowing the strength of the activation of nodes in the network to vary in response to aspects of the current problem solving context, and stochastic influences on the strength of these factors that render the spread of activation fundamentally non-deterministic. In doing so, she has captured many features of human analogies in a computer program.

PARADYME (Kolodner 1988b; Kolodner 1989) manages case retrieval for the JULIA meal-planning system. PARADYME uses a number of different similarity heuristics designed to measure various pragmatic criteria. These are:

1. A preference for source cases whose conclusions match the goal of a target problem. This is a measure of the relevance of the source to the target (see section 2.1.3).

2. A preference for highly salient features. PARADYME computes salience based on the history of a feature's effectiveness in selecting useful sources.

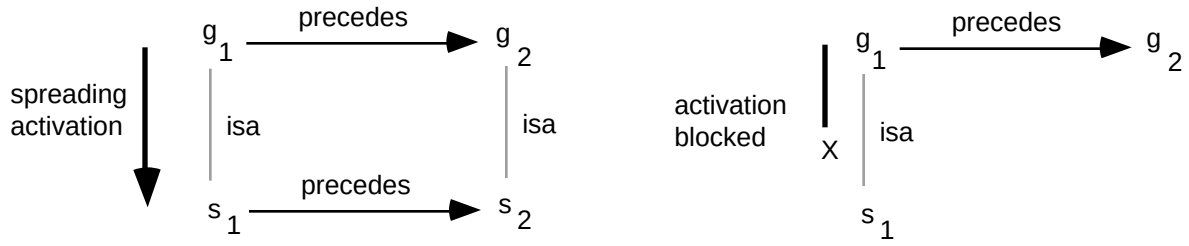
3. A preference for sources that closely match the target. One aspect of this is the number of matching features.

4. Frequency constraints that prefer cases that have been used often in problem solving.

5. Recency measures, a preference those that have been used recently.

6. Ease of adaptation measures.

PARADYME (Kolodner 1988b) is also interesting in its use of systematic patterns in source retrieval. PARADYME searches for sources in a hierarchical organization of concepts. Retrieval begins by matching the target with the most general index, and working down the tree to find an appropriate case. It further constrains search through a form of systematicity, preferring analogies that maintain the structure of the source. PARADYME cases are sequences of events; events can be specialized. If there are events g_1 and g_2, such that g_1 precedes g_2 in some sequence of events, and these have specializations s_1 and s_2 respectively, then activation spreads from g_1 to s_1 only if s_1 precedes s_2. Activation spreads to specializations only if the specialization does not omit part of a case's sequential structure (figure 9). This is particularly interesting in that it combines systematicity constraints with hierarchical organization.



Spreading activation in PARADYME

Figure 9

Discussion: Similarity and learning

Similarity depends upon contextual and systematic information about targets and sources. Rather than using simple numeric similarity metrics, more powerful methods use a domain theory to infer explanations of why a target and source are similar. In order to be effective, such approaches cannot rely upon a priori distinctions between deep and surface knowledge, but must be able to draw upon any known or inferred properties of the target and source.

However, efficient source retrieval must place some constraints on the properties that are considered in source selection. A reasonable approach to this

problem would use empirical learning to favor those properties that have proven useful in selecting sources in solving past problems. This is that approach taken by the SCAVENGER algorithm. Rather than relying on a fixed retrieval vocabulary, it uses inductive learning to infer a retrieval vocabulary from successes and failures in problem solving. The next section discusses the problems of organizing such a vocabulary into an effective retrieval mechanism.

2.2.4 Memory organization

Given a suitable similarity metric, designers of analogical reasoners must still address the problem of memory organization. Even easily computed similarity metrics cannot support an exhaustive search of large numbers of candidate sources. Index structures organize memory into classes of similar sources, as defined by their common features. Schank (1982) calls these patterns *remindings*, where a reminding is some aspect of a memory that is easily triggered by new situations.

A typical memory organization accesses sources through a hierarchical discrimination network (see figure 2 in chapter 1). Each node of the index contains a pattern; child nodes specialize the pattern of their parent by adding constraints. Each index node represents common properties of a class of similar target and source cases. Sources are stored under the most specific index whose pattern they match; retrieval finds the most specific indices that match the target, and retrieves the associated sources.

One of the earliest examples of hierarchical memory organization was EPAM (Feigenbaum 1961). EPAM was designed as a model of the way in which humans store and recall nonsense syllables, such as "pib" or "mur." EPAM stored syllables in a hierarchical discrimination network; each node of the network defined a common feature of a set of syllables: a letter appearing in a certain position. Given a new stimulus, EPAM determines if it has already seen it or a similar syllable by searching this network.

UNIMEM (Lebowitz 1980; Lebowitz 1986) further refines the management of hierarchical discrimination networks. Concepts are sets of features; each node of the network specializes its parent by specifying values for one or more features. Unlike

EPAM, a node does not add just one discriminating feature to its parent: each step in descending the hierarchy may add several features to the parent description. This improves efficiency slightly, although a more important effect is its ability to define interesting generalizations at each node. Each source is indexed by the most specific node that it matches. Retrieval of a concept involves descending this network to find the most specific nodes with a satisfactory match to the new object. Matching does not require that the new object match the node on all features; UNIMEM accepts partial matches.

UNIMEM places new sources in memory by finding the most specific node they match. It then examines the unexplained features of the source: these are features whose values failed to match the index node or any parent of the index. If enough instances stored under the node have unexplained feature values in common with the new source, UNIMEM creates a new specialization based on those features and indexes these sources accordingly. UNIMEM constructs taxonomies incrementally, modifying its category structure as it encounters new training instances.

Because of the close relationship between analogy and categorization, research in *concept learning*, the induction of general definitions of classes of similar instances, has much to offer a theory of memory management for analogical reasoning. *Supervised induction* begins with a set of instances that have already been classified by a teacher. Here, the learner's task is to infer general concepts that will correctly classify training instances and have a high probability of correctly classifying future instances. *Unsupervised approaches* construct useful categories for a set of unclassified instances, relying on the learner's own heuristics or experience to select quality categories.

ID3 (Quinlan 1986) is a supervised induction method that has greatly influenced the retrieval mechanism of this research. ID3 constructs a decision tree that will correctly classify a set of training instances. It attempts to build the smallest tree that correctly classifies the training instances, under the assumption that such a tree will make the fewest unsupported assumptions about the data and will be most likely to categorize future instances correctly. In constructing this tree, ID3 uses an information-theoretic analysis of training instances to select the properties that convey the most information about category membership. The memory adaptation

algorithm of chapter 3 adapts ID3's information-theoretic approach to the construction of index hierarchies; I will discuss this technique and its implementation in more detail at that time.

In general, memory management techniques use unsupervised approaches. *Clustering* is the construction of taxonomies from unclassified training data. Traditional clustering methods use numeric similarity metrics and build taxonomies that minimize the distance between instances in a class (Everitt 1981). Other approaches to clustering include Bayesian induction (Anderson and Matessa 1991; Cheesman 1990), connectionism (Hinton 1990; Rumelhart et al 1986), and genetic algorithms (Holland 1986; Holland and others 1986).

COBWEB (Fisher 1987) is one of the most sophisticated clustering algorithms, using a global measure of category utility to select among potential modifications of a taxonomy. COBWEB is interesting for its probabilistic representation of categories: it does not represent a category as a collection of features, but as a collection of probabilities that an object in the category has certain properties. It is an incremental clustering algorithm; it inserts new instances into its hierarchy by trying different modifications to the hierarchy, using *category utility* (Gluck and Corter 1985) to evaluate these alternatives. Category utility measures the extent to which the taxonomy maximizes the probability that two objects in the same class share a property, and minimizes the probability that objects in different categories share properties. It is interesting to note that COBWEB uses a global measurement of taxonomic quality instead of the local metrics used in UNIMEM. Thompson and Langley (1991) have extended the COBWEB approach to include structured representations.

In contrast to numeric approaches, *conceptual clustering* (Michalski and Stepp 1983) emphasizes qualitative measures of similarity and the construction of intensional definitions of categories. Conceptual clustering must address two problems (Fisher and Langley 1985):

1. The construction of useful clusters of instances.

2. The construction of a suitable general *characterization* of these categories.

CLUSTER / 2 (Michalski and Stepp 1983) is one of the first conceptual clustering algorithms; it is also interesting for its integration of the clustering and characterization aspects of the problem. CLUSTER / 2 constructs categories iteratively, attempting to create categories that optimize properties of their descriptions. One such optimization prefers categories that lend themselves to short conjunctive descriptions. This notion of using qualitative properties of category descriptions to construct and represent taxonomies is similar to many of the heuristics used in the program of chapter 3 to select among potential analogies.

Discussion: Memory management as clustering

In general, clustering approaches to memory management adapt source memory when saving new sources. Incorporation of a new source into memory usually proceeds by placing it under the most specific index that matches it, and proposing local improvements to the hierarchy. Options are either to store the source under an existing index, create a specialization of an existing index to accommodate the source and any other, similar stored instances, or merge the current node with a parent or sibling and place the source in the result. UNIMEM and COBWEB both exemplify this source-centered approach to memory management: Their goal is an effective partition of the set of analogical sources. This approach is limited in that:

1. Although it may achieve an easily searched partitioning of source memory, there is no guarantee that this will be effective in determining the relevance of sources to specific target problems. Such indices improve the speed of searching the source base, but their ability to predict the relevance of sources depends primarily on the quality of the retrieval vocabulary.

2. All such algorithms construct index patterns by selecting properties from a pre-determined retrieval vocabulary. It is not clear how they can be adapted to construct indices if these properties cannot be

assumed to be known for all targets and sources. This is particularly true if the space of candidate properties is unbounded.

3. The ability of a property to select a relevant source cannot be determined by examining sources alone. The relevance of an analogy must be found in the relationship between the target and the source. Relevance must be learned by examining and evaluating sources in the context of actual targets.

Machine learning techniques offer a solution to these problems, allowing the reasoner to infer generalizations that may then be used to predict what sources might be relevant to future target problems. For example, Barletta and Mark (1988) have developed a technique called Explanation-Based Indexing (EBI) for using a domain theory to learn predictive indices. They have demonstrated this idea on the fault recovery problem: given a failure of a piece of automated machinery, determine the actions needed to recover from the fault. EBI stores new source cases, which are provided by a human trainer. Given a set of observations made on the system at failure and the human's solution, EBI uses a domain theory to explain how the steps of a plan relate to each observation. If the theory can demonstrate that some action in the plan modifies an observation, it infers that the observation is important and saves it for use in retrieving the case. Note that the ability of the system to distinguish relevant features from irrelevant side-effects depends entirely on the domain theory.

By saving general patterns of analogy, a learner can become more efficient in detecting and developing future analogies. This can lead to an improvement in its ability to solve new problems as well as those directly contributing to the abstraction. It may also lead to an improved ability to develop future abstractions. Gregory Bateson (1972) calls the ability to "learn to learn" *deutero-learning* noting that humans improve their rate of learning across a series of similar tasks. The relationship between analogies and abstractions may provide a means of achieving deutero-learning in a computer program.

2.2.5 Summary and conclusions

There are several issues that the design of source selection algorithms must address:

1. It must support efficient retrieval. Most approaches, including the model proposed in this dissertation, do this through a hierarchical indexing system.

2. Indices must be effective in predicting the usefulness of sources. In contrast to approaches that rely upon fixed retrieval vocabularies, more robust memory organization systems must learn through experience.

3. It is important that memory organization strategies avoid excessive proliferation of indices or the maintenance of categories that do not help solve target problems.

4. Retrieval patterns must be adequately expressive. An indexing system must be able to use deep, structural knowledge, as well as simple features, in describing classes of stored instances.

The retrieval mechanism described in the next chapter addresses these issues.

Sections 2.1 and 2.2 have discussed analogical inference and source retrieval in depth, describing and criticizing current approaches. Although these discussions have covered considerable ground, my approach to the literature has been fundamentally influenced by the interaction theory of metaphor. The next section describes the interaction theory in more depth, and makes explicit many of its influences on my interpretation of the current literature and the design of the SCAVENGER analogical reasoning system.

2.3 An interactionist critique of computational approaches to analogical reasoning

This chapter has outlined a variety of approaches to source retrieval and analogical inference. It has also articulated a number of criticisms to these approaches. The *interaction theory* of metaphor provides a unifying framework for interpreting this work and focusing these criticisms.

2.3.1 The interaction theory of metaphor

Black (1962) characterizes three different views of metaphor:

1. The *substitution view* holds that metaphors can be replaced by equivalent literal expressions. Under this view, the metaphor "Man is a wolf" is simply another way of making the literal statement: "Man is dangerous and predatory." The main flaw with this view is its failure to assign any real semantic significance to metaphor: If metaphors are only a round-about way of saying something that could just as well be said literally, why should we bother with them?

2. The *comparison view* is a variation of the substitution view that treats metaphor as a condensed form of an explicit comparison, such as a simile. The comparison view would interpret the metaphor "Man is a wolf" as meaning "Man is *like* a wolf (in being dangerous)": this emphasizes the role of the comparison in conveying the intended meaning. Although it treats metaphor as equivalent to its literal counterpart, the comparison view gives it a necessary function as a vehicle to transmit this meaning. While this justifies metaphor, it still does not distinguish it from its literal re-statement. Black also objects to the comparison view on the grounds that it "suffers from a vagueness that borders on vacuity."[6]

3. The *interaction view* of metaphor is an effort to remedy the limitations of these theories. It defines an essential role for metaphor in language and thought, and lays a foundation for understanding its means of operation.

Black (1962) describes the interaction view of metaphor as consisting of "seven claims":

---

[6] This objection cannot be leveled against computational instantiations of the comparison view.

> *(1) A metaphorical statement has two distinct subjects - a "principle" subject and a "subsidiary" one.*[7]
>
> *(2) These subjects are often best regarded as "systems of things" rather than "things."*
>
> *(3) The metaphor works by applying to the principle subject a system of associated implications characteristic of the subsidiary subject.*
>
> *(4) These implications usually consist of "commonplaces" about the subsidiary subject, but may, in suitable cases, consist of deviant implications established ad hoc by the writer.*
>
> *(5) The metaphor selects, emphasizes, suppresses and organizes features of the principle subject by implying statements about it that normally apply to the subsidiary subject.*
>
> *(6) This involves shifts in meaning of words belonging to the same family or system as the metaphorical expression; and some of these shifts, though not all, may be metaphorical transfers. . . .*
>
> *(7) There is, in general, no simple "ground" for the necessary shifts of meaning - no blanket reason why some metaphors work and others fail.*

In a frequently cited metaphor of his own, Black compares the effect of metaphor to a view of the night sky "through a piece of heavily smoked glass on which certain lines have been left clear." This glass, which corresponds to the analogical source, makes visible those stars that appear in the clear lines; it also imposes a structure of relationships (the lines in the glass) upon the sky. This structure is the system of relations, or "commonplaces" that are transferred from the source to the target:

> *We can say that the principle subject* [the target] *is "seen through" the metaphorical expression - or, if we prefer, that the principal subject is "projected upon" the field of the subsidiary subject* [the source]. (page 41)

Under this view, metaphors cannot be replaced by their literal equivalent: they are a unique form of inference. To quote further:

> *Their mode of operation requires the reader to use a system of implications . . . as a means for selecting, emphasizing, and organizing relations* in a different field [my emphasis]. (page 46)

---

[7] In the terminology of this dissertation, the *principle subject* is the target, and the *subsidiary subject* is the source.

It is this act of transferring a system of relations into a "different field" that gives metaphor and analogy its unique power. The essential feature of metaphors and analogies is in their ability to transfer knowledge across semantic contexts. This results in unpredictable changes in the interpretation of the transferred knowledge: an "equivalent" literal statement would loose the semantic information that arises out of the interaction between the two subjects. In a statement that echoes section 2.2.2's criticisms of numeric similarity measures, Black argues that any literal paraphrase will fail to capture the subtler notions of relevance found in the metaphor:

> *For one thing, the implications, previously left for a suitable reader to educe for himself, with a nice feeling for their relative priorities and degrees of importance, are now presented explicitly as though having equal weight. The literal paraphrase inevitably says too much - and with the wrong emphasis.* (page 46)

The notions of similarity underlying metaphor are too complex and context sensitive to be expressed in any simple, literal fashion.

Perhaps the most intriguing aspect of the interaction model is its recognition that metaphors transfer knowledge in both directions. Although transfer primarily affects the target, it can result in changes in the source as well:

> *If to call a man a wolf is to put him in a special light, we must not forget that the metaphor makes the wolf seem more human than he otherwise would.* (page 44)

Black's analysis of metaphor is not unknown to the analogical reasoning community. Indeed, it has exerted a profound influence on the development of the field. However, as we should also expect, much remains to be done in translating it into computational form. Putting it in terms appropriate to this discussion, the interaction of the philosophical and computational approaches to metaphor and analogy has produced challenges, insights and shades of meaning not present in either domain.

2.3.2 Interactionism and computational models of analogy

Research in analogical reasoning has produced computational counterparts to all three of Black's theories of metaphor. As is often the case, with computational renderings of philosophical or cognitive theories, these implementations reveal

previously unrecognized subtleties in their less formal counterparts, and a greater understanding of their underlying structure.

A number of analogical reasoners seemingly reflect the substitution approach in using analogies to transfer knowledge that could have been also inferred directly ("literally"). For example, derivational analogy (Carbonell 1986; Carbonell and Veloso 1988) uses analogies with old solutions to speed up problem solving. The problem-solving operators are known to the system; it could produce equivalent solutions without recourse to analogy. The analogy is equivalent to the "literal" solution; it only serves to guide search and improve efficiency. This should not, however, mask the significance of the heuristic knowledge that is so transferred. Derivational analogy and similar systems transfer knowledge of *how to* solve target problems; this procedural knowledge is not available outside of the context of the analogy. Indeed, the behavior of these problem solvers is best understood in terms of the interaction of target and source problems.

Many case-based reasoning systems could be (superficially) classified as taking a comparison approach. As with the substitution view, transfer does not change the meaning of source operators: the target solution could be interpreted as a "literal" rendering of the initial problem. Unlike substitution-oriented counterparts, CBR frequently represents all problem-solving knowledge as source cases; there are no general rules or operators that could be used directly in solving targets. This approach also leads to complexities more reminiscent of the interaction model: The representation of source cases must support generalization and adaptation beyond the original source problem. Memory-organization strategies strive for predictive indexing of source cases; implicitly, at least, these strategies must take the structure of the target domain into account.

The interaction view is most evident in analogical reasoners that infer new declarative knowledge of the target problem. Structure-mapping theory (section 2.1.2) incorporates a great deal of the interaction view, particularly in its emphasis on systematicity. The application of structure mapping to empirical discovery emphasizes analogy's ability to infer systematic aspects of target structure (Falkenhainer 1990b; Falkenhainer and Michalski 1990). Transformational approaches (section 2.1.5) recognize the importance of modifying source knowledge in adapting it to a

new semantic context. These case-adaptation operators serve the function of bridging the gap between semantic contexts. The relationship between analogy and categorization discussed in section 2.1.4 is strongly interactionist in flavor: analogies underlie categories, and categories constrain analogies. Way (1991) makes explicit reference to the interaction view in justifying her use of type hierarchies to model metaphor understanding.

Computational research has also made contributions to the interaction view itself. For example, the pragmatic criticism of structure mapping addresses a problem implicit in Black's theory. Interactionism holds that metaphors transfer "commonplaces" about the source to the target, but it is unclear about the source of these commonplaces and how they can be distinguished from "less common" forms of knowledge. Pragmatic approaches (section 2.1.3) argue that problem solving goals provide a context that defines relevant knowledge.

The criticism of static retrieval vocabularies in section 2.2.1 is a direct result of the interaction view: How can we select a retrieval vocabulary without considering its intended application to solving target problems? Frequently, such vocabularies hide assumptions about target problems in implicit linguistic biases; a complete theory of analogy should make these biases explicit. Similarly, the criticisms of numeric measures of similarity in section 2.2.2 further reflect the interactionist approach.

Although interactionism has already exerted a profound influence on theories of metaphor and analogy, it is an extremely rich theory and much work remains to be done in exploring its ramifications for computational models of analogical reasoning. This is particularly true in the area of source retrieval.

2.3.3 The interactionist critique revisited

Chapter 1 used the interaction view to articulate three specific criticisms of current models of source retrieval. These criticisms are further supported by the research discussed in this chapter. These are:

1. The separation of retrieval and inference.

2. Monotonic measures of similarity.

3. Memory organization as clustering.

To elaborate:

The separation of retrieval and inference

Most models of analogical reasoning separate source retrieval and analogical inference. The interaction view undermines this approach by holding that while source retrieval depends upon target-source similarity, it is often the metaphor (or analogy) itself that *creates* similarity. This implies that some form of analogical inference underlies all notions of similarity. The relationship between categorization and analogy (section 2.1.4) underscores this observation. The pragmatic critique of structure mapping in section 2.1.3 argues for the use of *target* goals in determining the relevance of sources; this implies that inferences about the target may play a role in retrieval.

A number of systems have proposed unified approaches to retrieval and inference. MEDIATOR (Kolodner and others 1985) uses generalized cases to both index sources and constrain inference. HYPO (Ashley 1988; Ashley and Rissland 1988a; Ashley and Rissland 1988b) integrates retrieval and inference in a process of reasoning about the relevance of legal cases. PI (Thagard 1988), PARADYME (Kolodner 1988b) and Copycat (Mitchell 1993) use spreading activation through a network of sources to search simultaneously for relevant sources and infer an explanation that accounts for that similarity.

The retrieval mechanism of chapter 3 offers another perspective on this interaction, interleaving retrieval, transfer and the evaluation of partial analogies in a stepwise fashion. A novel feature of this approach is its evaluation of partial analogies to guide inference and retrieval.

Monotonic measures of similarity

The validity of monotonic, often numeric, measures of similarity is a direct casualty of the interaction view: one of the primary justifications Black gives for the

interaction view is its unique ability to convey "relative priorities and degrees of importance" (Black 1962). Interactionism's reliance on systems of relations, and its support for meaning shifts across contexts implies that simple, quantitative measures can only approximate similarity.

This is perhaps the most widely recognized and explored of the three criticisms, with a number of systems accounting for goals and context in reasoning about similarity. MEDIATOR (Kolodner and others 1985), HYPO (Ashley 1988; Ashley and Rissland 1988a; Ashley and Rissland 1988b), PI (Thagard 1988) and PARADYME (Kolodner 1988b) are all examples of this approach. Protos (Bareiss 1989) and GREBE (Branting 1989) also construct qualitative explanations of why objects are similar. Explanation-based indexing (Barletta and Mark 1988) extends this idea further, using a domain theory to construct a qualitative explanation of a specific target-source similarity, and saving a generalized form of the explanation as an index to source cases.

The approach taken in this dissertation measures similarity in context by tentatively transferring source properties to the target, and evaluating their impact on the target problem.

Memory organization as clustering

The final, and perhaps the most significant of my criticisms focuses on the use of clustering to organize source memory. On the surface, such approaches construct reasonable taxonomies of sources. Their central flaw is the construction of taxonomies by examining source properties *in isolation from target problems*, even though it is precisely these problems that determine the relevance and predictive value of those properties. In order to be successful, such approaches must rely on programmer-defined biases in the definition of a retrieval vocabulary.

Explanation-based indexing (Barletta and Mark 1988) is a notable exception, constructing indices out of explanations of the similarity of targets and sources, as is MEDIATOR (Kolodner and others 1985), which adjusts its generalized episodes in response to experience in solving target problems.

2.4 Conclusion

The interaction view of metaphor is an extremely rich source of ideas about the nature of metaphor and analogy. Many of these ideas have been included in computational models of analogy, while many more of the theory's implications remain to be explored. The next chapter proposes a model of retrieval and inference that further investigates the ramifications of the interaction theory for source retrieval in analogical reasoning.

3
An Interactionist Model of Source Selection

I'm afraid that God has no master plan. He only takes what he can use.

David Byrne
The Facts of Life

SCAVENGER is an analogical reasoning program that formalizes the conjectures made in the previous chapter. SCAVENGER uses analogies with existing knowledge to interpret empirical observations; this is a variation of the *case-based interpretation* problem (Kolodner 1993). I have chosen this problem for its intrinsic theoretical interest, its ability to test the ideas proposed in this work, and its relevance to a variety of applications. These applications include the interpretation of tutorial examples of LISP method behavior (chapter 4), the diagnosis of bugs in failed procedures (chapter 5), and the analysis of simulations in qualitative physics (chapter 6).

Interpreting observations requires the classification of objects and events, and the inference of relationships between them. The proper categorization of empirical data is an essential prerequisite for both scientific and common-sense reasoning: Physics forms theories about the class of moving bodies, not about specific rocks. We describe the physiology of species of animals, not individuals. Psychiatry places individual patients in general diagnostic categories. These theoretical categories are sufficiently abstract that they must be inferred from "raw" observations. In addition to categorizing the objects of an observation, interpretation must determine the basic interactions between objects and events. Many essential relationships, such as causality or similarity, are not apparent in an observation, but must be inferred from a combination of background knowledge and observational data.

Many reasoning systems oversimplify the interpretation process, requiring that objects be explicitly categorized through such statements as moving-body(small-brown-stone) or bird(tweety). Others rely upon rules to infer categories from properties of

objects themselves: X has(feathers, X) lays-eggs(X) → bird(X). More sophisticated techniques use fuzzy logic or Baysian techniques to manage noisy or ambiguous information. However, all of these approaches assume that the meaning of an object can be inferred from properties of the thing itself. In doing so, they ignore many essential contextual influences. Depending on the context of its use, a rock may be a hammer, a weapon, an ornament, or a fossil record of an extinct animal. A single phrase or gesture can express praise, contempt or humor in different situations. The interpretation of an observation depends upon both the properties of its components and the patterns of their interaction with other things.

Metaphors and analogies are powerful mechanisms for applying context to interpretation. This follows from the interaction theory's contention that metaphors shape meaning by transferring *systems* of relations between sources and targets. These systems of relations are an effective expression of context. For these reasons, the problem of interpreting observations through analogy provides an ideal test of the interactionist approach to source selection described in this dissertation. I have designed and implemented SCAVENGER as a vehicle for performing these tests.

This chapter begins with a description of the interpretation problem and its representation. Section 3.1 outlines the interpretation problem and introduces the basic representations used in this work. Section 3.2 discusses the merits of this problem. Section 3.3 provides a high-level overview of SCAVENGER's architecture, and section 3.4 discusses each of its primary modules in detail.

3.1 The analogical interpretation problem

I have framed the interpretation problem as follows:

Given:

1. An unexplained observation. An observation is a sequence of events; these can change objects, although the effect of the changes may not be apparent. Observations do not make causal relationships explicit, but they have a sequential structure that can suggest causality.

2. Knowledge about objects and events in a variety of domains. These are candidate sources for explaining observations.

Goals:

1. Select objects and events from an appropriate source domain, and construct an analogy between them and the target observation.

2. Transfer necessary information from the source to the target in order to categorize target items and construct a causal explanation of the observation.

3. Confirm the validity of the interpretation, either through experiments, simulations or inference. Assume that this process is costly, and should be done as little as possible.

4. Based on the successful analogy, acquire knowledge that will improve future problem solving performance.

This formulation captures much of the essential structure of interpretation in realistic domains.

3.1.1 Representing observations and theories using the Common LISP Object System

In choosing a representation for observations and analogical sources, I was guided by the following goals:

1. To use a language that was general and expressive enough to support diverse applications.

2. To enable the program to evaluate its hypotheses empirically, instead of relying on a human user to evaluate its interpretations.

3. To emphasize the role of context in interpreting observations. This meant that the ability to describe a rich set of causal relationships was essential.

4. To reduce the likelihood that the behavior of the algorithm might be overly influenced by unintended representational biases.

These criteria lead to my choice of the Common LISP Object System (CLOS) as a representation language. SCAVENGER's sources are represented as a library of CLOS class and method definitions. Observations are sequences of function evaluations such as would be produced by the LISP interpreter. Some of the functions in these *transcripts* may be unknown; these are targets for the analogical reasoner. Interpretation attempts to find a mapping between them and appropriate sources such that the sources will re-produce the behavior shown in the transcript. This mapping is SCAVENGER's interpretation of the targets: it projects the semantics of corresponding sources onto target problems. An example will illustrate.

Define a *transcript* to be a series of LISP forms and the results that would be produced by evaluating them using the LISP function: eval. Each evaluation is written as: (<evaluable-form> -> <result>). Assume we give SCAVENGER the following transcript to interpret:

(SETQ X (?TARGET-FUNCTION-1)) -> (INSTANCE ?TARGET-CLASS 1)[8]
(SETQ Y (?TARGET-FUNCTION-1)) -> (INSTANCE ?TARGET-CLASS 2)
(?TARGET-FUNCTION-2 X 1) -> ? [9]
(?TARGET-FUNCTION-2 Y 2) -> ?
(SETQ Z (?TARGET-FUNCTION-3 X Y)) -> ?
(?TARGET-FUNCTION-4 1 Z) -> T
(?TARGET-FUNCTION-4 2 Z) -> T

In this transcript, certain method names (?TARGET-FUNCTION-1, ?TARGET-FUNCTION-2, ?TARGET-FUNCTION-3 and ?TARGET-FUNCTION-4) and class names (?TARGET-CLASS) fail to match any classes or methods known to the system. Instead, the analogical reasoner

---

[8] The expression, (INSTANCE ?TARGET-CLASS 1) indicates an instance of the class ?TARGET-CLASS. The number, 1, serves to identify different instances. This is a syntactically sweetened version of LISP's instance labels, which use pointer values to distinguish instances: #<TARGET-CLASS #x2D9921>

must find an analogical mapping between the targets and known classes and methods that can account for the behavior observed in the transcript.

SCAVENGER maintains a library of source classes and methods, along with high-level descriptions of their semantics. For example, the class, BAG[10], is stored along with a list of its methods and its LISP definition:

class-name: BAG
methods: (ADD, MEMBER, UNION, . . .)
lisp-definition: (DEFCLASS BAG . . .)

Similarly, SCAVENGER stores source methods with their LISP definition, specifications of the types of their arguments and result[11], a list of the arguments the method changes as side-effects, and a high level description of method semantics. For example, the ADD method of the BAG class would be represented as:

method-name: ADD
argument-types: (T BAG) -> BAG
result: (ARG-1)
side-effects: ()
definition: ((ADD-TO-COLLECTION ARG-0 ARG-1))
lisp-definition: (DEFMETHOD ADD (ITEM (B BAG)) . . .)

This definition states that ADD takes two arguments, the first is an object of any type (type = T); the second argument and the result are instances of BAG. ADD returns its second argument as a result, and has the behavior of adding its first argument to the collection passed as its second argument. It has no side-effects; i.e. it does not alter any of its other arguments.

---

[9] The "?" indicates an unknown or unspecified value. I have allowed "wild card" results in order to make the problem of interpreting transcripts harder, and the resulting system more flexible in its application.

[10] A BAG is an unordered collection allowing duplicate members.

In interpreting the above transcript, assume that the source base includes such classes as BAG, SET, QUEUE, STACK, etc., along with their methods.[12] SCAVENGER will search for a source class and methods that can successfully run the example. One of the analogies that SCAVENGER constructs is:

```
Class map.
        TARGET-CLASS -----> BAG
Method map.
        ?TARGET-FUNCTION-4 -----> MEMBER
                arg0 --> arg0; arg1 --> arg1
        ?TARGET-FUNCTION-3 -----> UNION
                arg0 --> arg0; arg1 --> arg1
        ?TARGET-FUNCTION-2 -----> ADD
                arg0 --> arg1; arg1 --> arg0            ; note argument reordering
        ?TARGET-FUNCTION-1 -----> MAKE-BAG
```

Note that the analogy between ?TARGET-FUNCTION-2 and ADD re-orders method arguments; many analogical reasoners do not allow argument re-orderings, since doing so adds greatly to problem complexity.

SCAVENGER confirmed this analogy by attempting to run the original transcript using the source functions. The result of this test is:

```
(SETQ X (MAKE-BAG)) -> #<BAG #x4A0E01>
(SETQ Y (MAKE-BAG)) -> #<BAG #x4A0E11>
(ADD-C 1 X) -> #<BAG #x4A0E01>
(ADD-C 2 Y) -> #<BAG #x4A0E11>
(SETQ Z (UNION X Y)) -> #<BAG #x4A0E39>
(MEMBER 1 Z) -> T
(MEMBER 2 Z) -> T
```

The analogy transfers the semantics of the source functions to the targets, inferring the following interpretation for the target methods:

```
?TARGET-FUNCTION-1 takes arguments: NIL
        returns result: RESULT
```

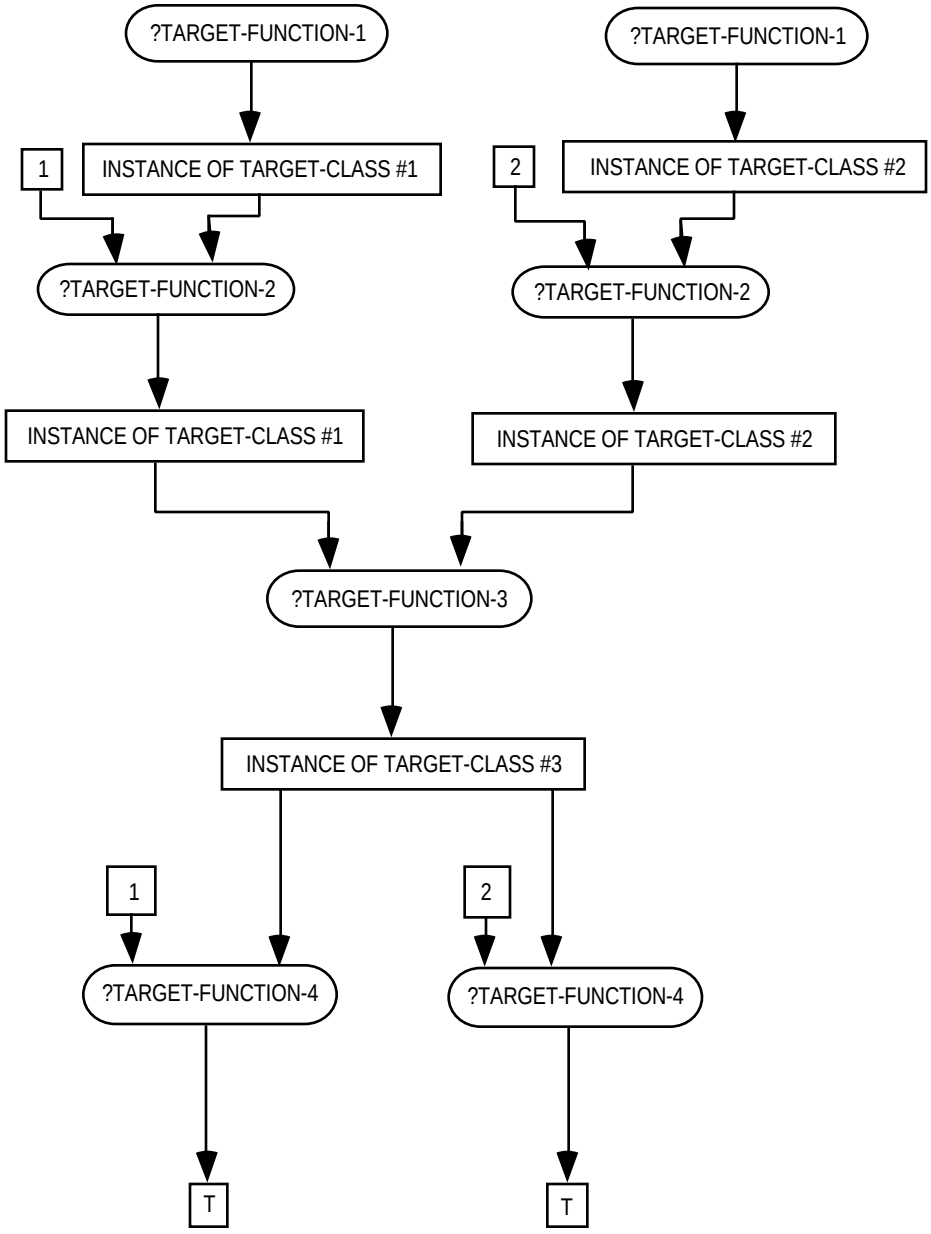
---

[11] This type specification is called the method's *signature* or *prototype*.

[12] The source base used to solve this problem not only included data structure classes, but a wide variety of other CLOS classes and methods, including a rational number package, time and date functions and a simple accounting package. Appendix 1 describes it in detail.

Function has description: ((NEW-OBJECT RESULT))
?TARGET-FUNCTION-2 takes arguments: (ARG-0 ARG-1)
returns result: ARG-1
Function has description: ((ADD-TO-COLLECTION ARG-0 ARG-1))
?TARGET-FUNCTION-3 takes arguments: (ARG-0 ARG-1)
returns result: RESULT
Function has description: ((SUM ARG-0 ARG-1))
?TARGET-FUNCTION-4 takes arguments: (ARG-0 ARG-1)
returns result: RESULT
Function has description: ((IS-IN-COLLECTION ARG-0 ARG-1))

These descriptions state that ?TARGET-FUNCTION-1 (MAKE-BAG) returns a new object as its result; ?TARGET-FUNCTION-2 (ADD) returns its second argument, with a new element added to it; ?TARGET-FUNCTION-3 (UNION) returns the result of a summation of its arguments; and ?TARGET-FUNCTION-4 (MEMBER) is a predicate testing whether its first argument is in the collection specified in the second argument. Using the descriptions inferred through the analogy, SCAVENGER can construct a causal explanation of the target transcript (figure 10).



A graph of a target transcript

Figure 10

This graph, a variation of a data flow diagram, represents causal structure that is not evident in the target transcript itself. For instance, the graph makes it clear that the order of the calls to ADD does not matter, so long as they precede the call to UNION. Similarly, it captures the independence of the two evaluations of MEMBER. It is

interesting to note that the structure of this graph does not follow from either the target transcript or the sources alone: it comes from the interaction of the two. These explanations also play a role in SCAVENGER's retrieval mechanism: in order to rank competing analogies prior to testing them, SCAVENGER will construct and analyze these explanations.

After finding and evaluating an interpretation of the target, SCAVENGER searches for features of the successful analogy that may be useful in solving future problems. It incorporates these features into an index structure that will improve source selection.

This problem captures many features of interpretation as it appears in realistic domains such as scientific discovery: Rather than allowing the system to rely on names to indicate the classification of individual events (function evaluations) and objects (arguments and results), these must be inferred from contextual and behavioral clues found in the transcript. Transcripts have a built-in sequentiality suggests events occurring in time; this, combined with the ability of CLOS objects to preserve state information allows them to represent causal interactions. Although there are causal interactions between statements in a transcript, these interactions and their effects on objects may not be obvious. Observations do not make the interpretation of target objects and events explicit; SCAVENGER must infer them using background knowledge and the information provided in the target transcript.

3.2 Why this is an interesting problem

I have chosen this problem, not only for its ability to challenge SCAVENGER's source selection mechanisms, but also for its generality and intrinsic interest. Interesting aspects of the interpretation problem and its formulation include:

3.2.1 It requires the use of systematic patterns in constructing analogies

The interaction theory recognizes that much of the power of analogical inference lies in its ability to transfer whole systems of relations between sources and targets. However, many models of analogy take an implicitly reductionist view, assuming that sources can be retrieved by matching on individual properties of objects, or that source properties can be transferred to the target individually, on a one-at-a-time

basis. Many measures of similarity ignore the role of context in their assumption that similarity increases monotonically with the number of properties two objects share (section 2.2.2).

As the preceding example indicates, target transcripts provide little information about *individual* functions and objects: it would be nearly impossible to interpret an isolated line of a transcript meaningfully[13]. Interpretations of target classes and methods emerge from the patterns of their interaction. Consequently, effective source selection must attend to such higher level structural features of the target as the numbers and types of target arguments, and the patterns of interaction between target functions.

3.2.2 "Observations" often under-constrain interpretations

Frequently, observations under-constrain their interpretation, requiring that the reasoner use past experience to resolve ambiguities. In the domains I have used to test SCAVENGER, target transcripts often support competing, empirically indistinguishable interpretations. For instance, the example of section 3.1.1 could be interpreted as describing a SET as well as a BAG. SCAVENGER's interpretation of the target, while duplicating the behavior of the transcript, is not guaranteed to be "correct" in any objective sense. Since the transcript alone cannot discriminate among possible interpretations, the analogical reasoner must do so heuristically.[14] It is important to note that this uncertainty is inherent in the problem statement itself; it is not a result of the inference procedures used for its solution.

3.2.3 It reflects the relationship between theories and the worlds they describe

Many AI programs fail to treat the relationship between the world and theories about it realistically. Often, they rely upon the closed world assumption (Luger and Stubblefield 1993; Luger 1994) and treat theories as if they captured all relevant knowledge of the world. In realistic domains, including both scientific and

---

[13] The tests discussed in section 4.4 corroborate this intuition.

[14] In this example, it used the heuristic of preferring the most general interpretation, in this case BAG.

commonsense reasoning, theories are seldom complete. In general, we must reason about worlds that are much more complex than our theories about them could ever hope to be.

As the previous example illustrates, SCAVENGER does not have a strong model of LISP semantics. The interpretation of source classes and methods is hidden in their definitions and the behavior of the interpreter. Target transcripts provide even less information about target methods. SCAVENGER's theories of its "world" do not capture the full complexity of that world; instead, it must select analogical sources using the high-level descriptions stored with its source methods and the limited information provided by the transcripts themselves. This mirrors the problems that arise in many realistic interpretation problems, such as perception, diagnostics and scientific discovery, where our theories are usually incomplete or approximate.

3.2.4 The problem representation eliminates many of the biases found in source retrieval

One of my goals in formulating this problem was to disallow, as much as possible, the unspecified, often unrealistic biases and heuristics that frequently appear in computer models of analogical reasoning. For example, many models of source selection exploit arbitrary restrictions on a retrieval vocabulary (see section 2.2.1), assuming that key predicates from the retrieval vocabulary will always be known for both the target and source, and will be represented in an easily manipulated form.

SCAVENGER's LISP transcripts do not conform to such ad hoc restrictions: There are no lists of target properties that can be matched against indices, no pre-determined retrieval vocabulary that can be used to construct indices, and no a priori restrictions on the knowledge that can be used in retrieval. The only restriction on the representation of targets is that they be reproducible by the LISP interpreter given the correct definitions of the target classes and methods.

Similarly, in selecting sources, SCAVENGER uses the information about argument types and the function descriptions stored with each source. I have placed no excessive restrictions on these descriptions, allowing the use of relations of any arity to be used in describing source methods.

3.2.5 Interpretation requires both background knowledge and empirical data

The interaction theory implies that metaphor and analogy are essential mechanisms for interpreting empirical observations. According to the interaction theory, the interpretation of a concept may be characterized in terms of its relationships to other concepts; metaphors and analogies interpret both targets and sources by transferring such relationships between them. This process involves the continuous interaction of background knowledge in the form of analogical sources and new empirical data, usually drawn from target problems.

The assumption-based retrieval mechanism described in this chapter takes a unique approach to this interaction between background knowledge and observational data in constructing interpretations. Because SCAVENGER's target transcripts provide little information about unknown classes and methods, conventional retrieval techniques do not work well on these problems. SCAVENGER selects sources by transferring aspects of source interpretation to targets and evaluating the implications of these assumptions. If empirical evaluation confirms an analogy, SCAVENGER assumes its interpretation is correct.

3.2.6 The interpretation of observations is an often neglected aspect of inductive learning

In realistic situations, empirical data seldom appears in an easily usable form; it must interpreted and classified. Many learning algorithms ignore this problem, exploiting the *single representation trick* (Cohen and Feigenbaum 1982):

> *If the representations for the rule space* [the space of generalizations to be learned] and *the instance space* [the space of training data] *are far removed from each other, then the searches of the two spaces must be coordinated by complex interpretation and experiment planning procedures. One trick commonly used to avoid this problem is to choose the same representation for both spaces (page 368)*

This assumption may limit a system's applicability in many situations. Cohen and Feigenbaum further state:

> *In more practical situations, the interpretation and experiment-planning routines serve to translate between the raw instances (as they are received from the environment) and the derived instances (after they have been interpreted as specific points in the rule space). (page 369)*

We can view SCAVENGER's solution to the interpretation problem as a technique for bridging this gap in certain circumstances. As such, it may play a role in developing more complete models of empirical learning.

3.2.7 Interpretation can foster insights into the mechanisms of perception

As here formulated, the interpretation problem is a variation of the *high-level perception* problem described in Mitchell's work on analogy (1993). High-level perception is concerned with the way in which intelligent agents construct systems of relations linking perceived objects; it is not concerned with such low-level perceptual processes as feature extraction. Like the simple string analogies studied in Mitchell's research (see section 2.1.5), SCAVENGER's transcripts provide the learner with a set of objects, but do not reveal their underlying relationships. These must be inferred through processes of analogy involving what is given in the observation and what the system already knows. Mitchell has argued that high-level perception is an important but seldom studied aspect of cognition that can legitimately be isolated from lower-level perceptual processes.

In addition, she argues for the use of analogies as a mechanism for high-level perception. Although SCAVENGER takes a different approach to analogy, it shares this underlying assumption. The interpretation of LISP transcripts includes many interesting properties of Copycat's string interpretation problem, particularly the under-constraint of possible interpretations by target problems. The results of the SCAVENGER experiments may provide further insights into the role of analogies in perception.

3.2.8 It is a vehicle for exploring the relationship between analogy, abstraction and learning

Section 2.1.6 discussed the relationship between analogy and abstraction. Although most analogical and case-based reasoners implicitly create and store abstract descriptions of families of similar sources in their indexing systems, many such systems rely excessively on representational biases in constructing those indices. These algorithms emphasize the selection of effective predictors of source relevance from a relatively small, pre-defined set of such properties, and build indices that store these predictors.

The formulation of the interpretation problem in this dissertation precludes any such simple approach to memory organization. LISP transcripts may consist of any syntactically correct, evaluable LISP forms. The descriptions used to construct indices reflect the semantics of the source functions and include relations of any arity. These descriptions reflect the meaning of the sources, and are not selected from bounded, fixed retrieval vocabularies. SCAVENGER must work with these higher level structural descriptions if it is to construct any sort of reasonable, abstract description of successful analogies and use it to improve future inferences.

3.2.9 The interpretation problem underlies a range of potential applications

The generality of LISP and CLOS supports a variety of applications. The example of section 3.1 illustrates SCAVENGER's application to the problem of interpreting examples of LISP function behavior. This problem grew out of an interest in the way humans use examples to communicate more general ideas. Understanding an example requires deciding what is relevant, what is implied and using this information to form an appropriate generalization. For instance, if I were teaching a class about the BAG data structure, I might illustrate the concept with an example similar to that of the previous discussion. How do students extract the appropriate generalization from such an example? Analogical reasoning is a promising mechanism for this process. In addition, certain aspects of this problem, such as the assumption that the example is intended to inform rather than deceive, provide a rich source of heuristics. Although this problem may have applications in the management of object-oriented software libraries[15], I chose it mainly for the inherent interest, and the challenge provided by the diversity of sources that must be managed in this application. Chapter 4 describes SCAVENGER's application to this problem.

Many other problems can be framed as forms of interpretation. For example, diagnosis and debugging could be viewed as the problem of finding an interpretation (a diagnosis) of a set of observations (symptoms). Assume that a system of interacting components is exhibiting abnormal behavior. We could describe the

---

[15] Could a programmer select reusable software modules by giving a system examples of their desired behavior?

system's behavior as a LISP transcript in which objects are the components of the system, and functions describe their behaviors and interactions. The knowledge base for the diagnosis would be a set of CLOS class and method definitions describing both normal and failure behaviors of individual components. Note that the observed behavior of such systems emerges from complex interactions between components: diagnosis must consider the overall pattern of the system's behavior. SCAVENGER will find an analogical mapping between sources describing normal or failure behaviors of components and the components specified in the transcript. It will test these mappings by running simulations to re-produce the observed faults; a successful analogy leads to a diagnosis. This "analogize and simulate" approach to diagnosis differs from the more analytical approach taken by expert systems, and adds another technique to the knowledge engineer's tool box. Chapter 5 describes SCAVENGER's application to the problem of diagnosing bugs in children's subtraction skills.

Another area where I have applied SCAVENGER involves reasoning about simulations. Because it assembles and tests object oriented programs, SCAVENGER is an ideal tool for simulation based problem solving. A brief discussion of this application may be found in chapter 6.

3.2.10 Interpretation is a complex problem

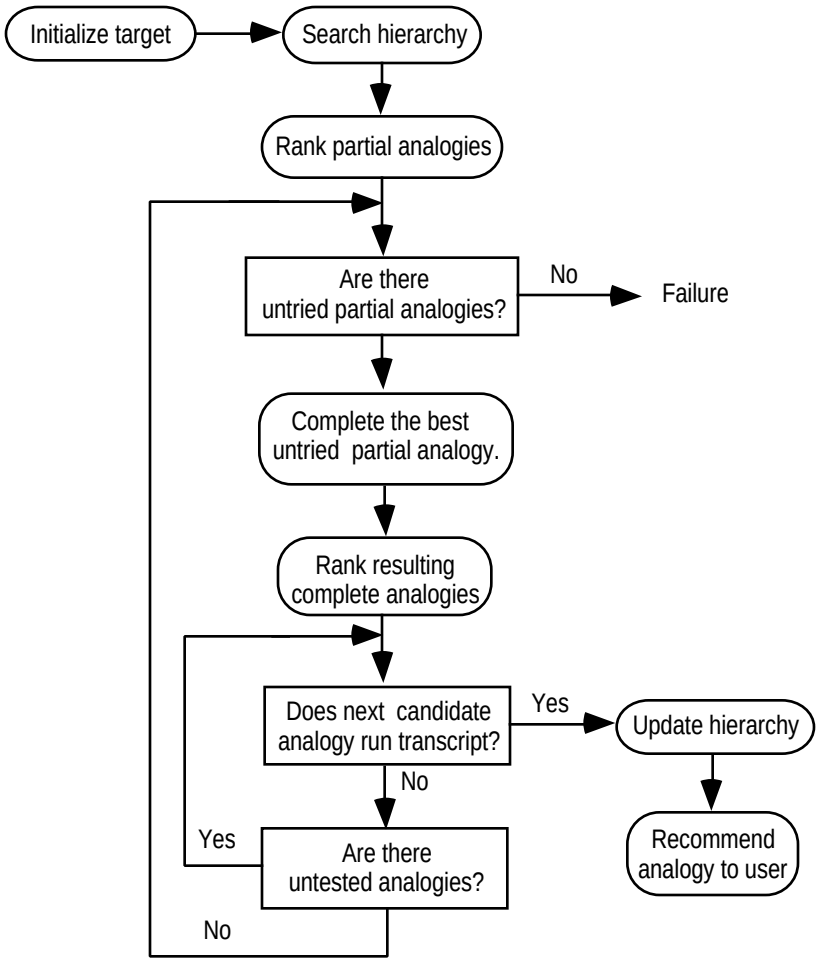
SCAVENGER's analogies allow targets and sources to match under any ordering of their arguments. The only constraints on this match are the requirement that both target and source have the same number of arguments, and that the mapping does not associate items of incompatible types. This contrasts with many more constrained forms of analogical inference (section 2.1.1) that artificially limit these possibilities. As a worst case approximation of the problem's complexity, assume that all functions have k arguments. If we assume n target functions and a source base containing m source functions, there are a total of

$$m^n k!$$

possible analogies that must be considered.

3.3 An overview of the SCAVENGER architecture

This section provides a high level overview of SCAVENGER's architecture; sec-



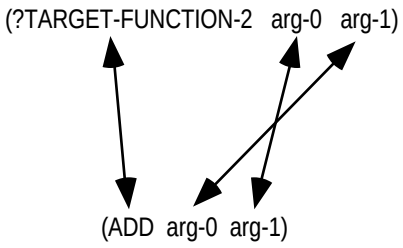
The SCAVENGER algorithm

Figure 11

tion 3.4 discusses it in detail. As illustrated in figure 11, SCAVENGER solves problems through a cycle of generating and evaluating candidate analogies. It begins by searching a hierarchy for sources that match target items. This produces a number of candidate analogies; these are partial in that components of the target may be unmapped. It ranks these partial analogies, completing and evaluating them in order. The remainder of this section briefly describes each phase of the algorithm, drawing on examples from the BAG analogy of the previous section.

3.3.1 Representing analogies in SCAVENGER

SCAVENGER represents analogies as mappings between elements of a target and a source. Figure 12 shows the mapping between ?TARGET-FUNCTION-2 and the ADD method in the example of section 3.1. Note that the analogy allows function names to match freely, and also reorders arguments. The only constraints on analogical mapping are the requirement that target and source have the same number of arguments, and that the types of corresponding arguments be consistent. SCAVENGER does not use operators to change the structure of target problems. By taking a relatively unrestricted approach to analogical mappings, it does not require many of the ad hoc limitations other models of analogy use to make the problem more tractable.

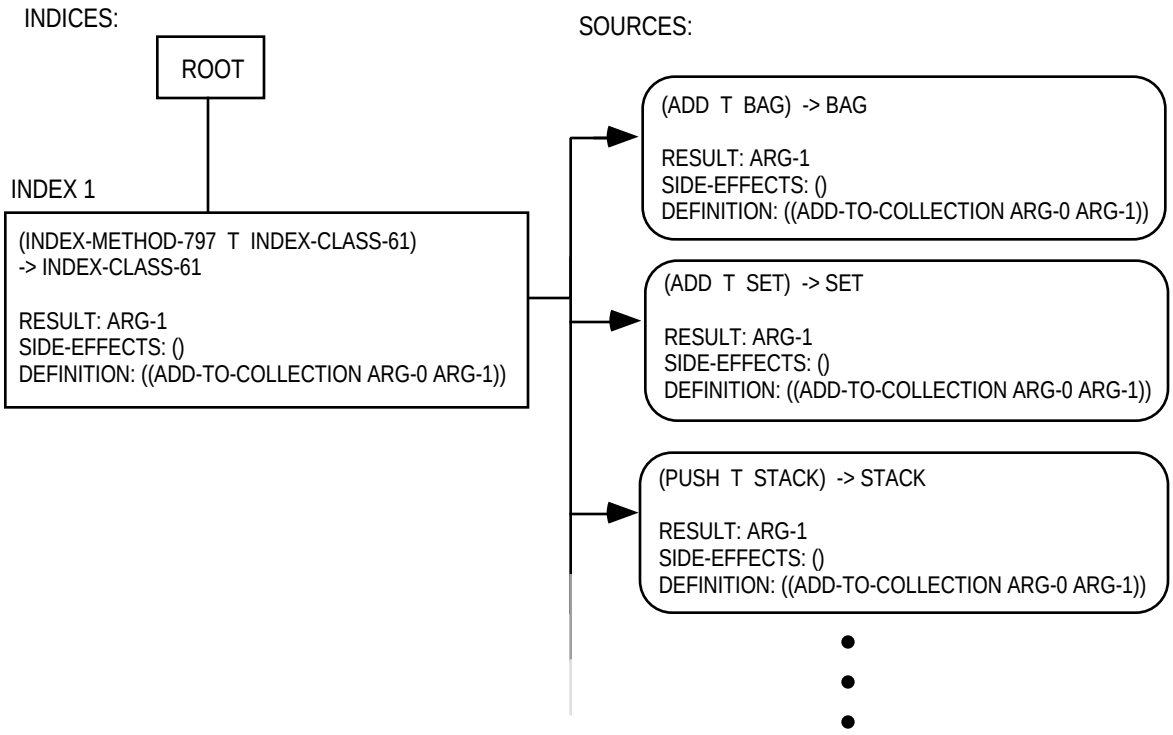


An analogical mapping

Figure 12

3.3.2 Initializing target problems

SCAVENGER begins by creating an internal representation of the target problem. Initialization designates as targets those functions that SCAVENGER does not recognize by name. It assigns a default interpretation to each target function, and makes initial inferences about the types of its arguments.



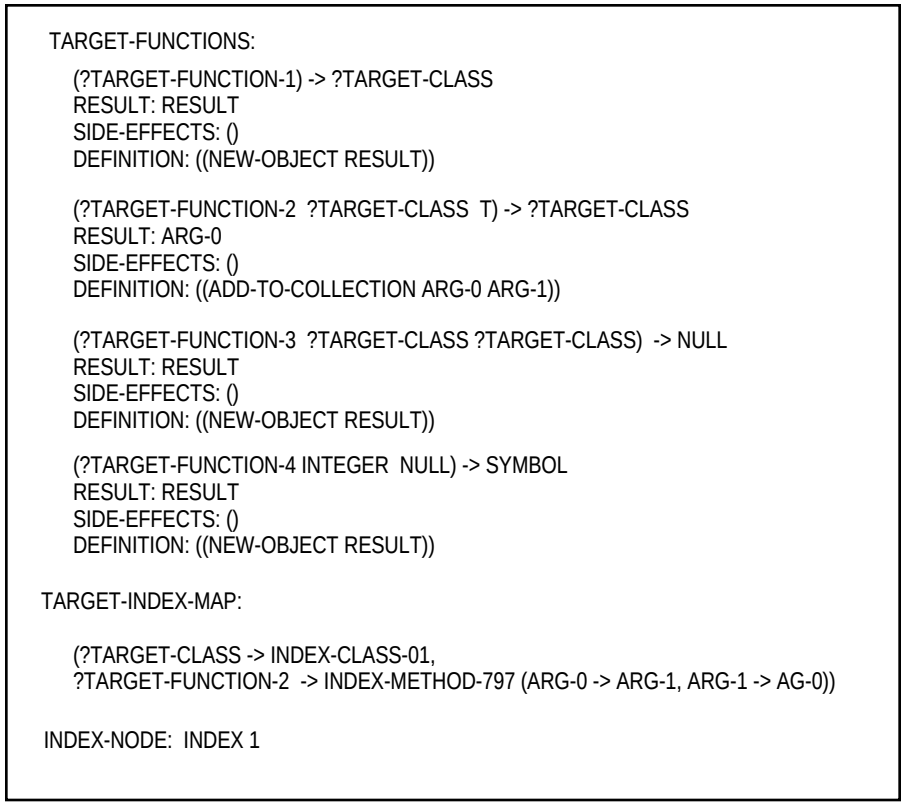
A simple SCAVENGER index hierarchy

Figure 13

In addition, the initializer records dependencies between function arguments and results in the target transcript. For instance, in the example of section 3.1, the initializer notes that ?TARGET-FUNCTION-2's second argument is the value returned by a call to ?TARGET-FUNCTION-1. Although neither type can be inferred from the transcript, it will be possible to infer information about the type of ?TARGET-FUNCTION-2's second argument when ?TARGET-FUNCTION-1 becomes mapped to a source. The initializer stores a record of such dependencies to simplify this inference.

3.3.3 Searching the index hierarchy

Each node of the index hierarchy describes a set of similar source methods. Call these descriptions *method patterns*. A method pattern specifies the argument types and high-level definition shared by a set of similar sources. Figure 13 shows a very simple index hierarchy. Note that the sources stored under an index may differ in details of their behavior.



A partial interpretation of the target problem

Figure 14

Retrieval explores the hierarchy in a standard tree search fashion. Matches between the target and the index are constrained by numbers of arguments and type information. On matching target functions with the method descriptions in an index node, SCAVENGER transfers the index's function descriptions to the matching targets. Each index match will transfer different method descriptions to the target problem. In this fashion, every index node match produces a different *interpretation* of the target problem; these interpretations are more abstract than those afforded by specific analogies. In order to distinguish these two levels of interpretation, the remainder of this chapter will refer to high-level interpretations as *interpretations*. It will refer to the final definitions of targets that are produced by an analogy with specific sources as *analogies*. Under this nomenclature, a single interpretation can be shared by many analogies. This ability to manage multiple levels of interpretation if a basis of much of SCAVENGER's power.

Search of the index of figure 13 produces two interpretations: one results from a match with the root, and does not change the default interpretation. The other results from the match with INDEX-1; figure 3.5 shows the interpretation that results from this match.

In figure 3.5, ?TARGET-FUNCTION-2 matches INDEX-METHOD-797 (note that the match specifies an argument re-ordering). ?TARGET-CLASS matches INDEX-CLASS-01. All functions except ?TARGET-FUNCTION-2 retain their default interpretations: they change none of their arguments, and return a new object as a result.

Information that was transferred from INDEX-CLASS-01 to ?TARGET-FUNCTION-2 includes:

1. Generalizing the type of its second argument from INTEGER (as suggested by the transcript) to T[16].

2. Designating the type of the result of ?TARGET-FUNCTION-2 to be ?TARGET-CLASS. This follows from the observation that the type of the result returned by INDEX-METHOD-797 is the same as the type of the first argument.

3. Specifying that ?TARGET-FUNCTION-2 adds its second argument to the collection specified in its first argument.

3.3.4 Ranking matches

In general, search of the index hierarchy will produce several different matches and resulting high-level interpretations of the target problem. SCAVENGER uses two types of heuristics to prioritize matches for further examination:

1. Favor interpretations that result from matches deep in the index hierarchy.

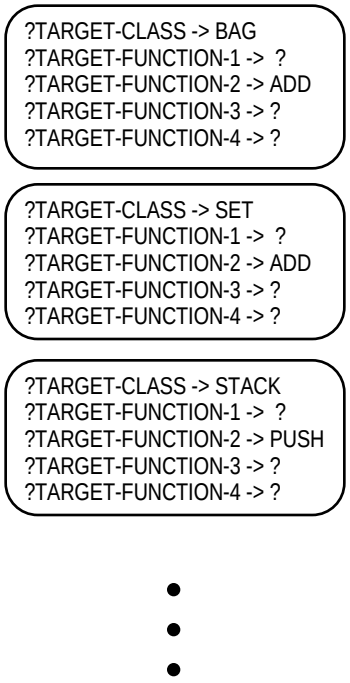
2. Use domain specific heuristics to break any ties remaining after step 1. In the problem of interpreting tutorial examples of LISP function behavior, SCAVENGER uses the function descriptions found under each interpretation to construct an explanation of the target (figure 3.1). It

---

[16] In the LISP type system, T is the most general type.

then evaluates this explanation graph for such properties as connectedness, branching, simplicity, etc. Other domains use different heuristics.

Two aspects of this ranking method are unique: The first is SCAVENGER'S ability to reason with *partial* analogical mappings; it does this by assuming a default interpretation for unmapped targets. The second is its ability to rate whole families of similar analogies by reasoning at the level of their common high-level interpretation; this helps improve the efficiency of the retrieval algorithm.



A set of partial analogies

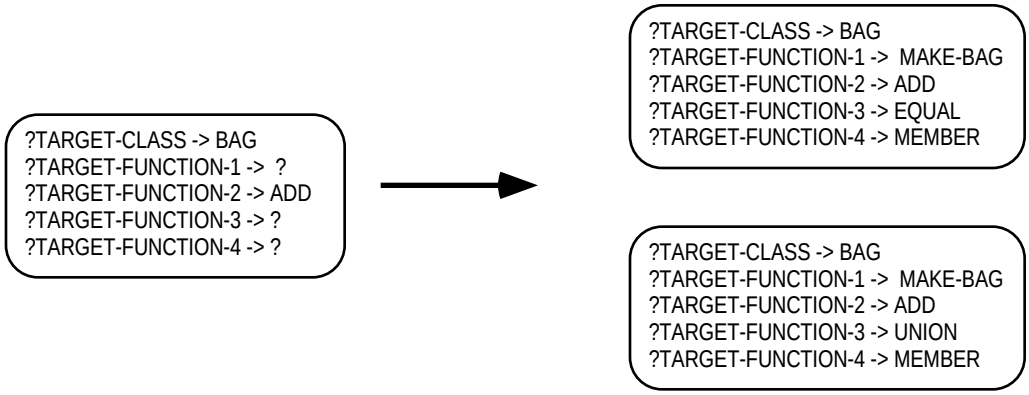
Figure 15

3.3.5 Completing analogies

SCAVENGER completes and evaluates index matches in the order established in 3.2.4. Each index match produces a mapping between a subset of the target methods and the method patterns stored in the index. In turn, the index maintains a set of mappings between its method patterns and all matching source methods and classes. The first step in completing the analogies afforded by a match is to compose

these mappings. This produces a set of partial analogies. Figure 15 shows a few of the partial analogies produced by the interpretation of figure 14. I have omitted the mappings between arguments in order to simplify the figure.

SCAVENGER finds all consistent completions to each of these partial analogies. Although this sounds expensive, a match of a single target generally produces enough information about argument types to constrain possible completions of the analogy. The first analogy of figure 15 binds ?TARGET-CLASS to BAG; since the signatures of ?TARGET-FUNCTION-1 and ?TARGET-FUNCTION-3 include instances of ?TARGET-CLASS as an argument, SCAVENGER is able to limit its efforts to methods of the source class, BAG. Each partial analogy may have multiple completions; SCAVENGER will produce all of them. Figure 16 shows the completions of a the first partial analogy of figure 15.



Completing a partial analogy

Figure 16

Each complete analogy transfers information about its signature and definition to each target method. Note that a single (partial) interpretation produced under step 3.2.2 can produce multiple interpretations at this stage. It is possible for different analogies to transfer different high-level descriptions to the same target methods. SCAVENGER groups those analogies together that have transferred identical signatures and descriptions to the target methods; using the terminology of 3.2.3, these are grouped according to common *high-level interpretations*. SCAVENGER sorts these interpretations using the heuristics described in 3.2.3.

3.3.6 Ranking analogies

SCAVENGER examines the interpretations produced in 3.2.5 in the order of their heuristic ranking. For each interpretation, it further sorts the analogies collected under it in decreasing order of promise. The heuristics used to sort analogies within a common high-level interpretation are different from those used to sort interpretations themselves (see section 3.4.5).

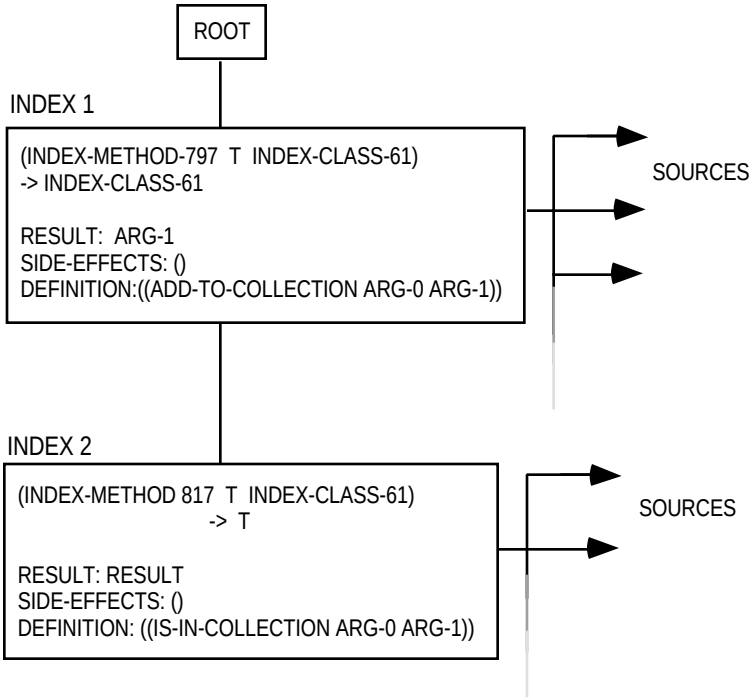
3.3.7 Testing analogies

SCAVENGER tests candidate analogies by substituting source function names into the target transcript, reordering function arguments as indicated in the analogical mapping and evaluating the resulting LISP forms. If this evaluation produces no errors and the results match those specified in the target transcript, SCAVENGER labels the analogy a success.

It is possible to tell SCAVENGER to either quit after finding the first successful analogy or continue to find all analogies that run the target transcript. If it quits after the first success, SCAVENGER assumes that the remaining analogies are failures.

3.3.8 Updating the hierarchy

On finding an analogy that successfully runs the target transcript, SCAVENGER updates the index node that produced the analogy. In general, a single index match will produce many different complete analogies; these may be grouped according to common high-level interpretations of the target. SCAVENGER considers each of these interpretations, labeling an interpretation as *positive* if it contains a successful analogy, *negative* otherwise. It then examines each target method that was not involved in the original match with the index, considering each different interpretation of that method as a candidate specialization of the original index.



An updated index hierarchy

Figure 17

SCAVENGER chooses the method description that best distinguishes the different analogical interpretations using a variation of the ID3 learning algorithm's information-theoretic evaluation function (Quinlan 1986). Section 3.4.6 discusses this in detail. It then creates a specialization of the parent index based on this method description, and stores *all* matching sources under it. Figure 17 shows the specialization produced by adding the inferred description of ?TARGET-FUNCTION-4 (MEMBER) to the hierarchy. Note that index nodes build on the matches with their parents; i.e., any match with INDEX 2 in figure 17 will include and be consistent with the mapping established in the match with INDEX 1.

3.3.9 Recommending analogies to the user

Recommendation displays the successful analogies to the user.

3.3.10 Conclusion: Scavenger and the interactionist approach to source retrieval

SCAVENGER provides an instantiation of the interactionist model of source retrieval introduced in section 1.3. The significant aspects of this model, and their treatment in the SCAVENGER algorithm are:

Assumption-based retrieval

The first feature of the interactionist model of retrieval is the interleaving of retrieval and inference. On matching an index node, SCAVENGER transfers the associated method descriptions to the target, and evaluates the impact of these descriptions on the target problem using domain specific heuristics. It uses these evaluations to select among candidate index matches. This analogical inference is tentative; SCAVENGER will consider a number of interpretations during source selection, ultimately rejecting many of them. It deals with this non-monotonicity by maintaining multiple interpretations of the target problem.

The use of context to evaluate similarity.

The interaction theory of metaphor rejects simple, monotonic measures of similarity. Measures of similarity must account for contextual and goal based effects. After transferring the method definitions stored under an index to the target functions, SCAVENGER evaluates the impact of this information on the target transcript. One approach I have explored evaluates properties of graphs of the target, such as appears in figure 3.1. Two aspects of this approach are interesting: SCAVENGER uses systematic properties of the explanation to rank interpretations; secondly, these properties only have meaning in the context of the target problem.

Empirical memory management.

In contrast to source-oriented, clustering approaches, SCAVENGER adapts its index hierarchy on the basis of its experience in solving target problems. Essentially, it uses the target problem to restrict the information that can be used to specialize the hierarchy. This represents a transfer of knowledge of *relevance* from the target back to the source. This is the riskiest hypothesis considered in this research, since

empirical memory management's ability to improve retrieval depends upon the existence of recurring similarities across target problem instances. If all target problems are completely different, this technique could actually degrade the program's performance.

The next section of this chapter discusses each of SCAVENGER's major components in detail. These discussions include descriptions of the algorithms, justifications for their design, and questions they raise for evaluating the algorithm. Part of that explanation is a detailed trace of SCAVENGER's treatment of a single example problem.

3.4 An extended example

Like the example of section 3.2, the next example problem concerns the interpretation of examples of LISP function behavior. The library used in this example contains a number of classes and methods, including data structure classes, a rational number package, time and date classes and a simple accounting package. It is described in Appendix 1.

This section presents the details of SCAVENGER's performance by tracing its solution to the transcript:

```
(setq x (?target-function-1)) -> ?
(?target-function-2 'a x) -> ?
(?target-function-2 'b x) -> ?
(?target-function-3 x) -> b
(?target-function-3 x) -> a
```

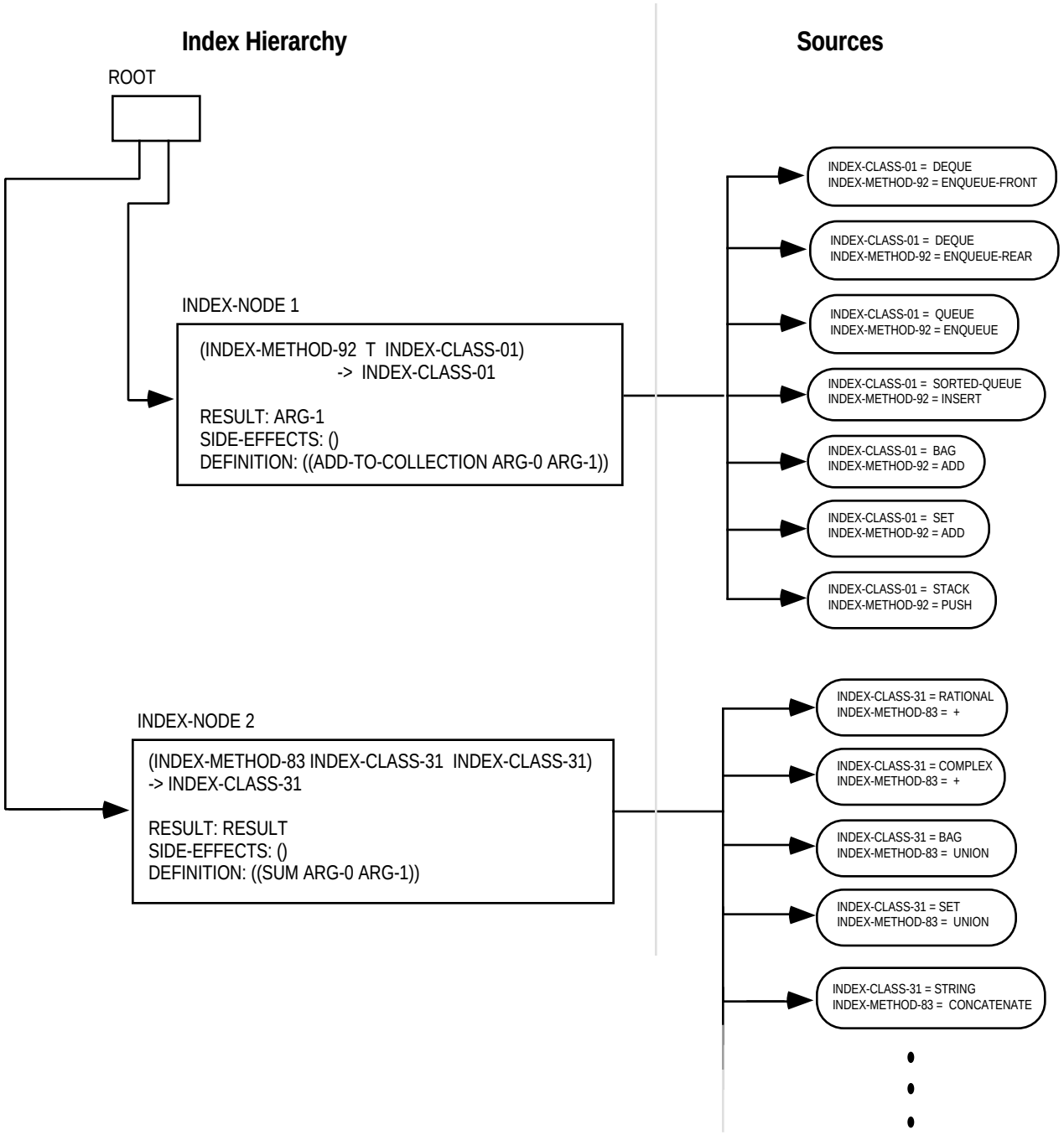
Note that the use of wild-cards ("?") in the target statement makes this a particularly difficult problem instance, since they prevent SCAVENGER from initially inferring the types of the results to ?TARGET-FUNCTION-1 and ?TARGET-FUNCTION-2.

Assume the index hierarchy of figure 18. To simplify the diagram, I have omitted the argument re-orderings in the index-source mappings. This hierarchy was produced by two runs of SCAVENGER: one of these problems was an example of

the addition of rational numbers, the other involved inserting and counting elements in a BAG data structure.

3.4.1 Representing and initializing sources and target problems

SCAVENGER stores source methods with their signatures, LISP definitions, side-effect specifications, and a description of their semantics. This description is



A simple SCAVENGER index hierarchy

Figure 18

not intended to be a detailed specification of the function, but a high-level characterization of its behavior. These descriptions should be general enough that

several source methods will have a common description; indeed, much of SCAVENGER'S power derives from the generality of these descriptions and their ability to represent classes of similar source methods.

As an example, of the description language used in the example understanding problem, consider the description of INDEX-METHOD-82 in INDEX-NODE-1 of figure 18. The result, ARG-1, indicates that the function modifies its second argument. The side-effects slot indicates changes that are made to arguments that do not appear in the function's result: this method has no side effects. The definition of the function behavior, ((ADD-TO-COLLECTION ARG-0 ARG-1)), specifies that it adds its first argument to the collection passed as its second argument. Note that 7 source functions share this method pattern.

The function description language is the major source of inductive bias in SCAVENGER. A good function description language should allow the index hierarchy to divide the source base into groups that are neither too large nor too small. This criterion is difficult to quantify and offers little help in actually designing description languages. In the example interpretation domain, I have used the heuristic of trying to pattern the source description language after the categories people use to describe similar functions.[17] In this example, "functions that insert something in a collection" ((ADD-TO-COLLECTION ARG-0 ARG-1)) is such a category. Similarly, INDEX-NODE-2 describes summation operators, another common category of functions. The description, (SUM ARG-0 ARG-1), indicates that the functions stored under the index return the sum of the other two arguments. Appendix 1 includes a detailed discussion of the language used to describe functions in this domain.

Initializing targets

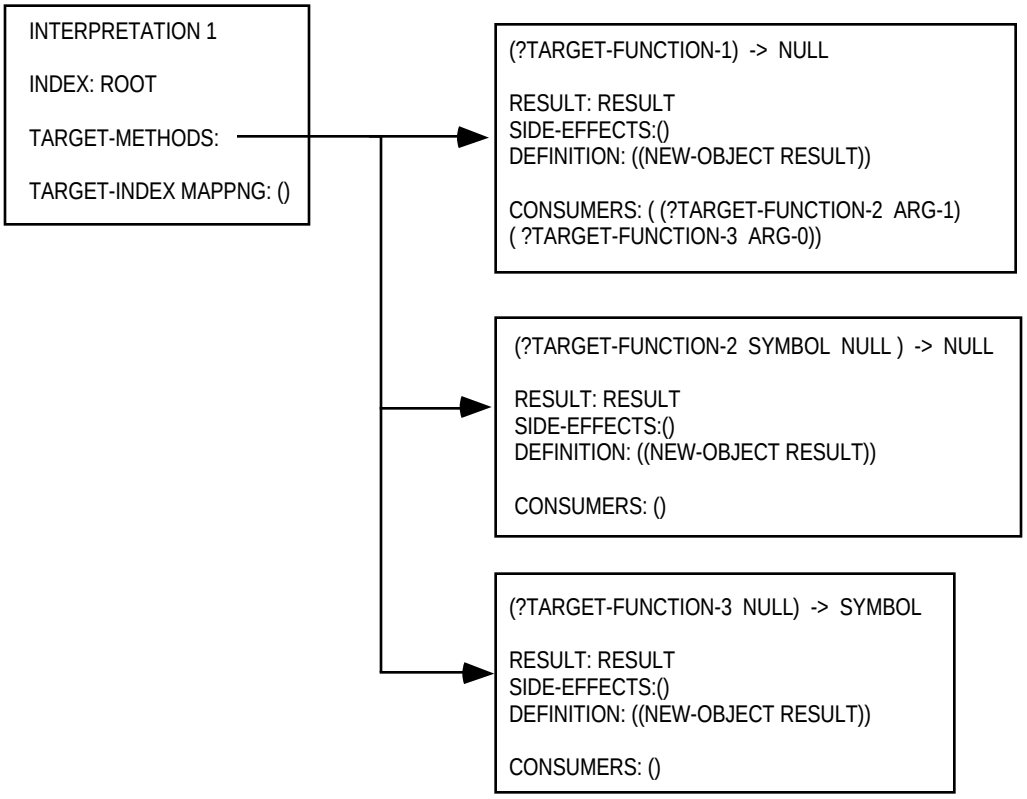
SCAVENGER's first action is to initialize the target problem. Figure 19 shows the initialized representation of the target problem.

---

[17] Empirical tests not only show that this heuristic works well; they also show that the algorithm is relatively insensitive to many aspects of this description language.

Aspects of initialization include:

1. Determining which functions and classes in the transcript are analogical targets. SCAVENGER maintains data bases of user defined CLOS classes and methods, and built-in LISP functions. If a function in the target transcript does not appear in either of



An initial interpretation of a target problem

Figure 19

these data bases, SCAVENGER assumes it is unknown. In the example, the unknown targets are ?TARGET-FUNCTION-1, ?TARGET-FUNCTION-2, and ?TARGET-FUNCTION-3.

2. Proposing an initial interpretation of the semantics of target methods. This default, "vanilla" semantics assumes that each target function returns a new object and has no other side effects. Although this is

generally changed in the analogy, default interpretations are needed to let the algorithm reason about partial analogies.

3. Making initial inferences about the types of target function arguments. SCAVENGER infers information about the type of target arguments from the target transcript. In the example, ?TARGET-FUNCTION-2 takes a symbol as its first argument; ?TARGET-FUNCTION-3 returns a symbol as a result. Note that it is possible that these functions will actually accept more general types than suggested by the target transcript; for example, many functions that operate on integers will also accept floating point numbers. If the type of an argument cannot be inferred, SCAVENGER assigns it the NULL type. Since the NULL type is a subtype of all other types, it will match any other type.

4. SCAVENGER will also store information about which target function arguments are passed the results of other target functions. This enables the program to make inferences about the types of these arguments when the types of objects passed to them are inferred.

3.4.2 Searching the index hierarchy

Search of the index hierarchy begins with the initial interpretation and the root of the tree. It searches the tree recursively according to the algorithm:

search-index (TARGET-PROBLEM, INDEX-NODE)

1) for each method in TARGET-PROBLEM that is not mapped to an index method, construct all analogical mappings between it and the method pattern in INDEX-NODE.

construct a new interpretation of TARGET-PROBLEM for each successful mapping.

if there are no successful mappings, return nil.

2) for every new interpretation produced by a match under step 1
and for every CHILD of INDEX-NODE,
call: search-index(NEW-INTERPRETATION, CHILD)

3) Return all interpretations produced by steps 1 & 2.

Calling SEARCH-INDEX on the default interpretation of the target and the root produces all matches with nodes in the index hierarchy. These are not only matches with leaf nodes, but include internal nodes as well. The reason for retaining all index matches is the uncertain nature of assumption based retrieval. If the initially preferred matches should fail to solve the target, SCAVENGER will need to try other alternatives. It does this by "failing back" to matches with internal index nodes. Since these match fewer methods of the target problem, they allow SCAVENGER to broaden its search of possible analogies.

Analogical matching

SCAVENGER's matcher will find all type consistent matches between a target and the method pattern stored under the index (the source). Analogical matches allow any re-ordering of arguments. Matching arguments and results is constrained according to the following rules:

1. If the type of the target argument is a built-in LISP type[18], then it can match an equivalent, or more general built-in type in the source. The justification for this is that functions often accept or return values that are subtypes of the types specified in the function definition. For example, "+" is specified as taking numbers as arguments; however, it can also accept and return integers. On matching, the target type is generalized to the type of the source.

2. An unmatched target-class can match any index-class or source-class. An index-class is a "dummy" type that matches classes in the source library (see figure 17). A source-class may be any member of SCAVENGER's source base.

3. If an argument is of type NULL, and the source type is a source-class or an index-class, then check if the source-class is already bound to a target-class. If it is, assume the argument is of this target-class; otherwise, create a new target-class, and bind it to the source.

---

[18] Such as integer, symbol, list, etc.

4. If a target-class is already matched to an index-class, it must match the same index class.

5. If a target is already matched to a source class, it may match the same or a more general source class.

On a successful match, SCAVENGER also updates the types of any arguments to other target functions that use the result of the current target. This update will generalize the argument according to rules 2, 3, 4 and 5.[19] If a result and argument do not match according to these rules, the original target / source match fails.

Retrieval in the example problem

In our example, the target matches all three index nodes, producing three different high-level interpretations. One of these, resulting from the root, is simply the initial interpretation of figure 19[20]. The others are shown in figures 20 and 21. In larger index hierarchies, type constraints and differences in the number of function arguments will eliminate many node matches.

3.4.3 Ranking interpretations

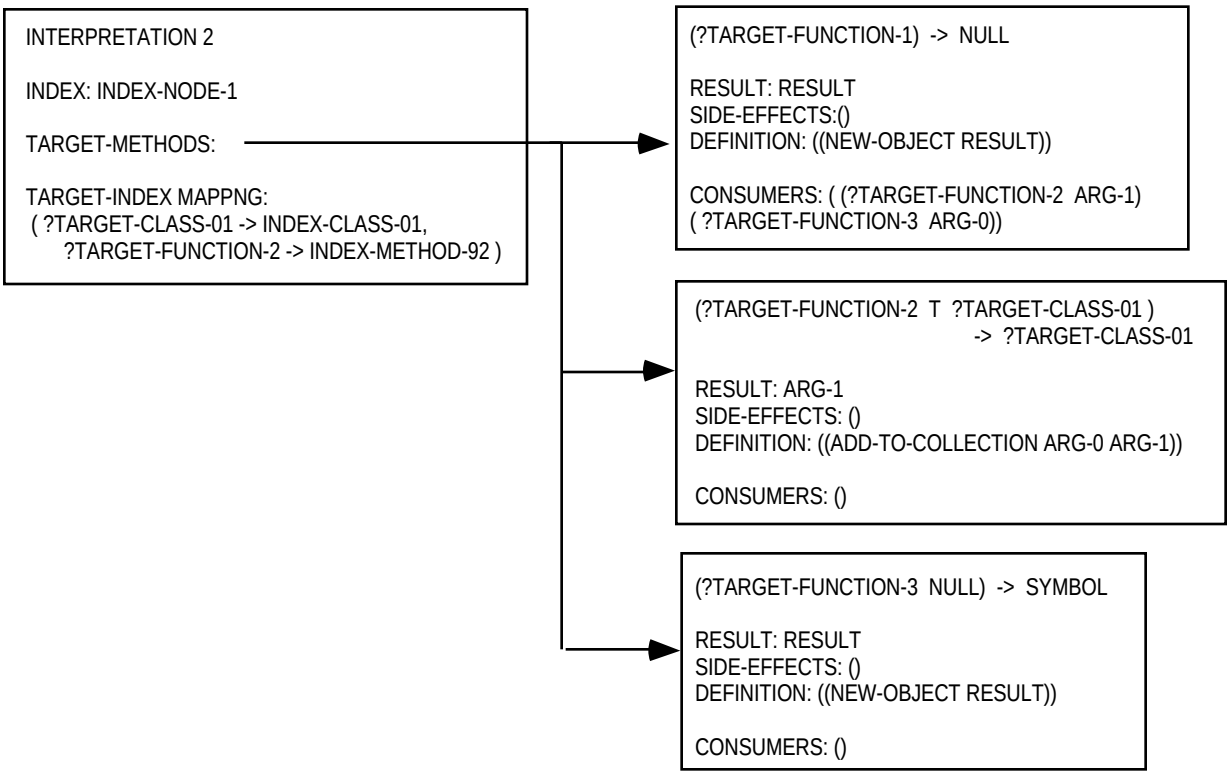
The next step is to rank these interpretations heuristically. The most important heuristic is a preference for interpretations resulting from matches with nodes deep in the index hierarchy. This measures the specificity of an interpretation, since

---

[19] The genericity of many built-in LISP functions, such as CAR (whose return value depends upon the types of the elements of the list it is passed) prevents making reliable inferences about interacting type constraints of built-in LISP functions. This is not as great a problem with classes in the source library, since I have represented situations where the type of a result depends upon the type of a method's arguments explicitly, using multiple function entries for each such situation. The extreme genericity of collection classes does not lead to a proliferation of source entries, since the type of collection elements is simply T.

[20] The source base includes other methods than those in the figure. If retrieval cannot find a match on any other nodes of the index, it will fail back to the root. Completing the interpretation afforded by this "match" leads to an exhaustive search of the source base.

matches with deeper nodes in the hierarchy must satisfy more constraints. This is a general heuristic, and SCAVENGER uses it in all ap



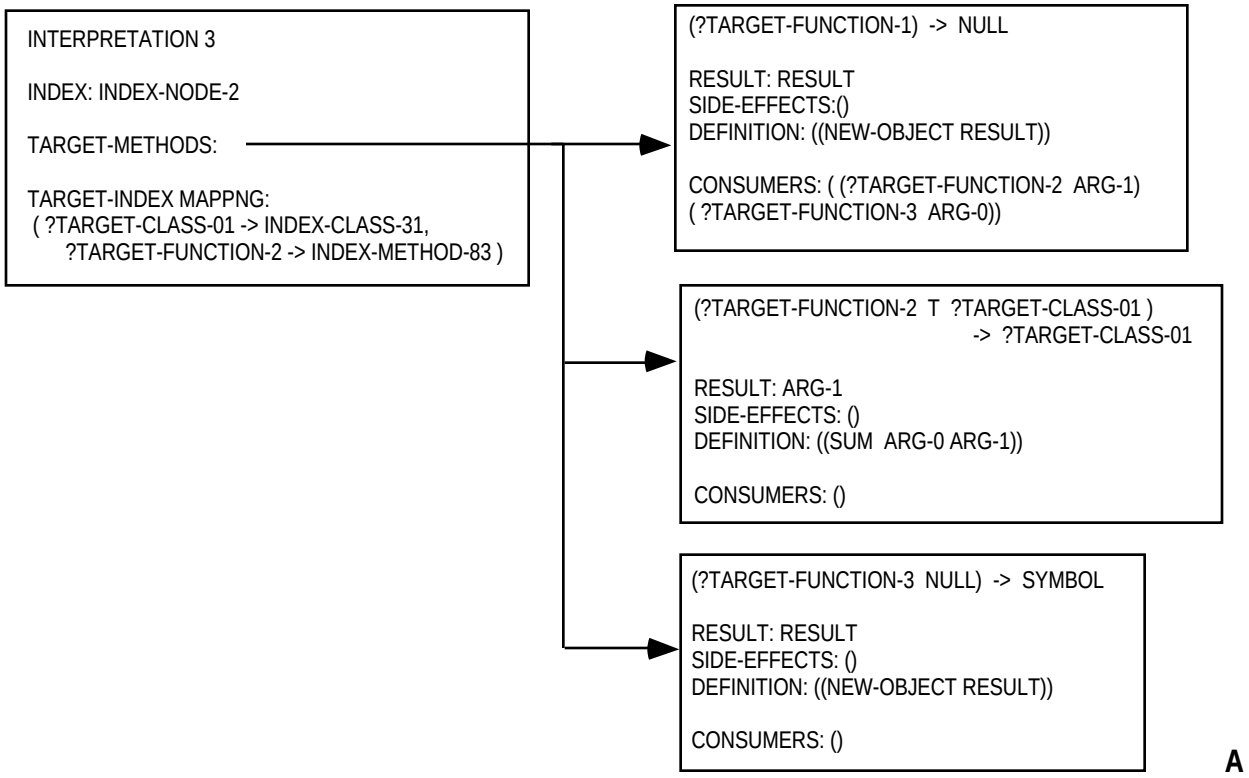
A partial interpretation

Figure 3.20

plications. If this heuristic produces a tie for the best interpretation, SCAVENGER will break it using domain-specific heuristics. These heuristics consider systematic properties of the target under an interpretation, and are often costly to evaluate; consequently, SCAVENGER does not do this unless it is necessary.

An example of heuristics for ranking candidate analogies

As mentioned in section 3.1, the domain shown in the current example grew out of an interest in the problem of communicating general concepts using examples. The most obvious factor in such communications is the assumption, made by both parties, that examples will be structured so as to communicate, rather then mislead. This assumption forms the basis for most of the heuristics I have developed for this domain.



A partial interpretation

Figure 3.21

SCAVENGER sorts candidate interpretations according to the weighted sum[21] of the following evaluations (all heuristic values are normalized to values between 0 and 1):

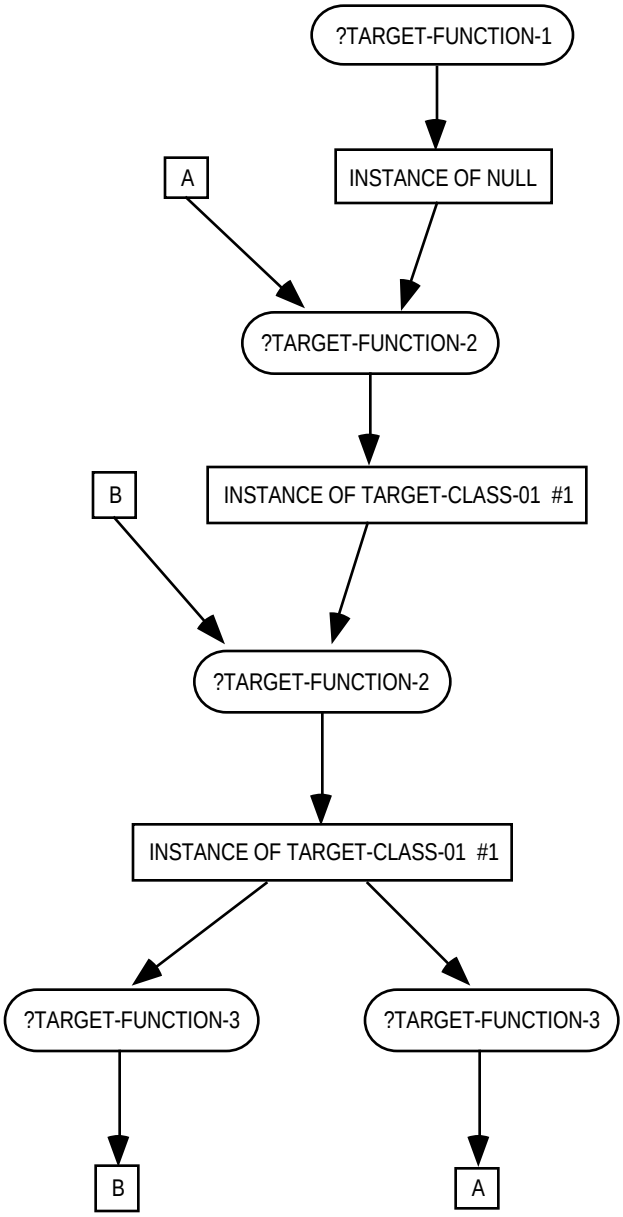
1. Minimize the number of different source classes used in the analogy. This heuristic is based on the assumption that an example is most probably intended to illustrate the behavior of a small number of target classes.

---

[21] I set the weights by hand in an effort to improve results. As I will discuss in chapter 4, I found that the particular settings were more important in the early stages of training SCAVENGER's retrieval mechanism. Once it developed its index hierarchy, performance became relatively insensitive to the heuristics used. The results of its learning algorithm are much more important to SCAVENGER's performance than the heuristics used or their weightings.

2. Prefer interpretations that minimize the ratio of method side-effects to methods. This heuristic assumes that examples would not include more than a few unseen side-effects, since this would be misleading.

3. Prefer analogies that have mapped the largest number of target function arguments. Such mappings will have satisfied more type constraints; consequently, they are more likely to be "correct.".



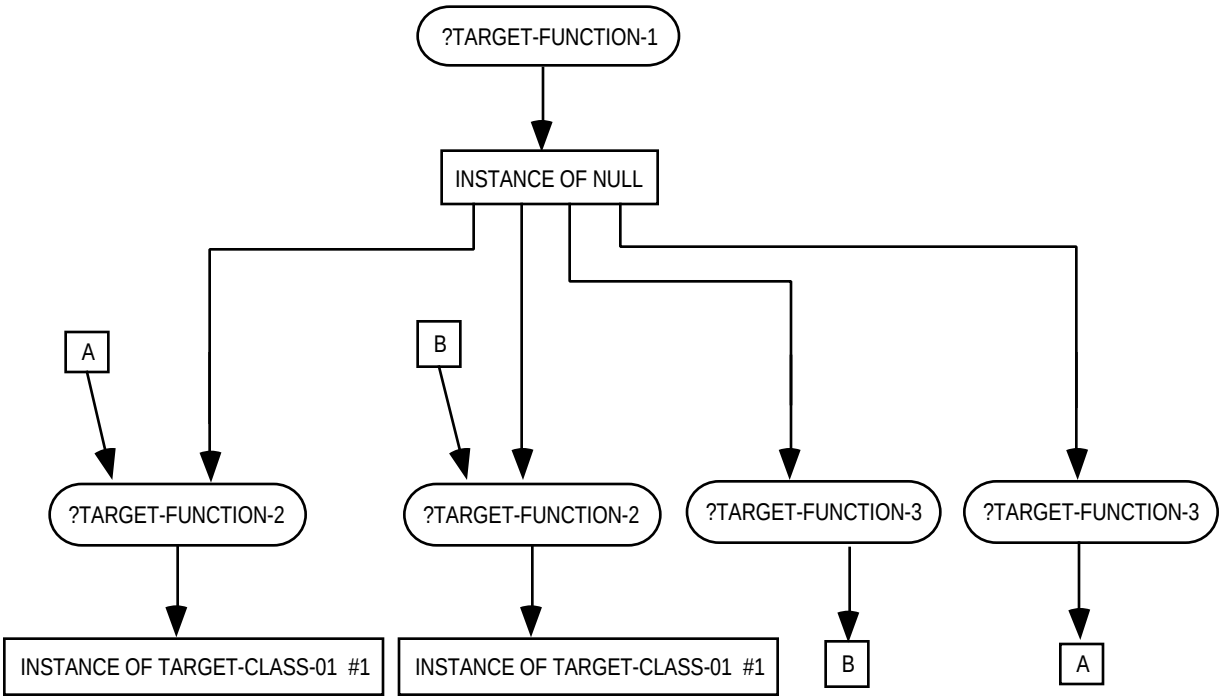
Explanation of target problem based on INTERPRETATION-2

Figure 22

The next two heuristics analyze the explanation of the target problem produced by each interpretation. Explanations are bipartite graphs, with nodes for functions and objects (arguments and results). Each object node reflects a given state of an object; if a function makes changes to an argument, the modified instance appears as

a new object node. Figure 22 shows the graph of the target problem under INTERPRETATION-2. Because ?TARGET-FUNCTION-2 specifies that its result is a modified arg-1, (ADD-TO-COLLECTION ARG-1), the graph shows this function returning a modified instance of TARGET-CLASS-01. In contrast, ?TARGET-FUNCTION-3 is not assumed to change any of its arguments, so both calls to it operate on the same instance.

Figure 23 shows the graph of the target problem produced under INTERPRETATION 3. Note that since the interpretation does not assume that any of the target functions change their arguments: all calls to ?TARGET-FUNCTION-2 and ?TARGET-FUNCTION-3 operate on the same instance produced by ?TARGET-FUNCTION-1.



Explanation of target problem based on INTERPRETATION-3

Figure 23

The remaining heuristics analyze these graphs:

4. Prefer graphs such as that in figure 21, in which functions tend to modify objects, over graphs such as figure 22, where functions do not modify object instances. The rationale for this heuristic is that since programs written in an object-oriented language like CLOS frequently use methods to change the state of objects, examples should illustrate at

least a few such changes. The graph of figure 23 has one object, created by ?TARGET-FUNCTION-1, which is not modified by any subsequent function calls; the graph of figure 22 performs successive modifications to (INSTANCE OF TARGET-CLASS-01 #1). This heuristic maximizes this as the ratio of the total number of different object instances appearing as function arguments to the total number of function arguments. In the graph of figure 22, this ratio is 4/6; in the graph of figure 23, it is 3/6.

5. The final heuristic is graph-connectivity. Prefer interpretations that produce a single, connected graph, over disconnected graphs. The basis of this heuristic is the belief that an example should communicate a single idea, rather than multiple ideas. It is reasonable to interpret a disconnected graph as demonstrating multiple ideas.

In the current example, INTERPRETATION-2 and INTERPRETATION-3 have the same evaluations for all heuristics except #4. This leads to the ranking:

{INTERPRETATION-2, INTERPRETATION-3, INTERPRETATION-1}

3.4.4 Completing analogies

The process of completing analogies proceeds according to the algorithm:

1) Each index stores a number of index-source mappings. Compose the analogy between the target and the index-methods with each of these mappings. This produces a set of partial target-source analogies; these are partial in that some target methods may not be mapped.

2) Complete these partial analogies exhaustively. Note that the resulting analogies may have different interpretations.

3) Group these analogies by common interpretation. Sort these interpretations using the heuristics described in section 3.6.

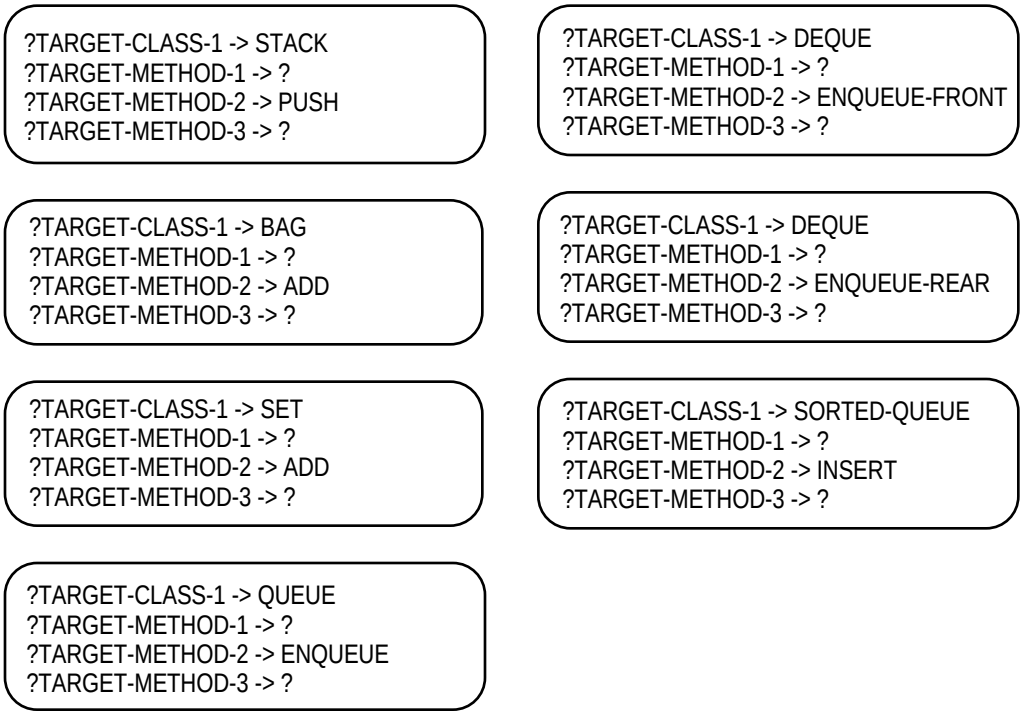
Examples of these steps are:

Composing target-index mappings and index-source mappings

In the target example we are discussing, INTERPRETATION-2 has the following mapping between target classes and methods, and its index classes and methods:

?TARGET-CLASS-1 -> INDEX-CLASS-01
?TARGET-FUNCTION-2 -> INDEX-METHOD-92

In turn, the index stores mappings between its class and method and each stored source. Composing these mappings produces 7 partial analogies; these appear in figure 24 (in order to simplify the figure, I have omitted the argument mappings. These are composed in the obvious way).



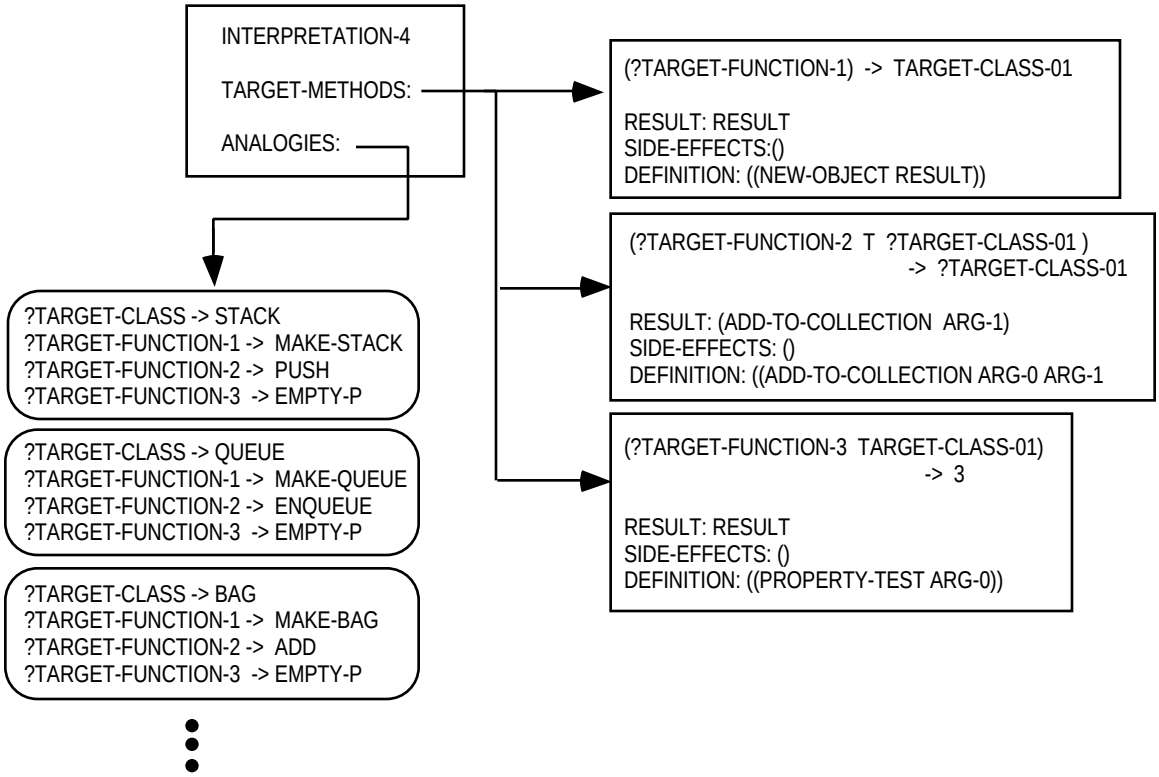
Partial analogies under INTERPRETATION-2

Figure 24

Completing partial analogies

SCAVENGER completes these partial analogies exhaustively. Although this could be an expensive operation, partial analogies usually incorporate enough type constraints to prune the space of possible sources. This is particularly likely after SCAVENGER has developed an extensive index and targets match nodes deeper in the hierarchy. Also, SCAVENGER maintains an additional index of methods by the

classes that appear in their arguments, this simplifies retrieval of relevant sources as target types become known.



A high-level interpretation and its set of similar analogies

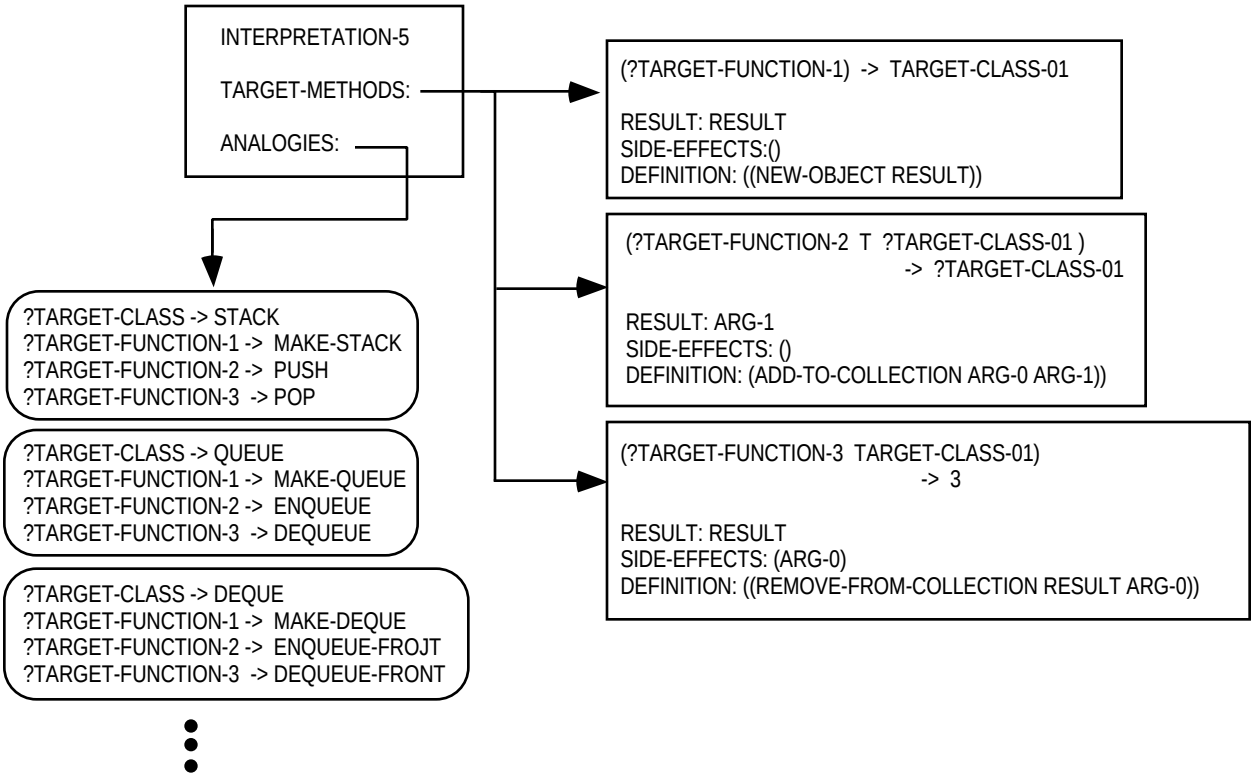
Figure 25

Completing the 7 partial analogies of figure 24 produces 14 complete analogies; a single partial analogy can have multiple completions. SCAVENGER groups these analogies by common interpretation; this grouping improves the efficiency of testing the analogies and updating the index hierarchy. Figures 25 and 26 show the two interpretations produced for the target problem, and a portion of the analogies stored under them.

In figure 25, INTERPRETATION-4 describes a group of data structures, with functions for creating them, adding elements to them and testing if they are empty. INTERPRETATION-5 (figure 26) describes another group of data structure classes, with functions for creating them, adding elements to them and removing elements from them.

Ranking completed interpretations

SCAVENGER next sorts these new interpretations using the heuristics described in section 3.4.3. In this example, SCAVENGER ranks INTERPRETATION-5 over INTERPRETATION-4.



Another high-level interpretation and its set of similar analogies

Figure 26

3.4.5 Testing analogies

Given a prioritized list of complete analogies, SCAVENGER evaluates them in order, stopping when it finds an analogy that runs the target example. SCAVENGER evaluates the analogies in each high-level interpretation produced under 3.4.4. In our example, it tries INTERPRETATION-5 first.

Sorting analogies in an interpretation

INTERPRETATION-5 contains 7 analogies. Notice that the STACK analogy will correctly run the target transcript, as will the mapping of the target-class onto a DEQUE[22], with ?TARGET-FUNCTION-2 mapping onto ENQUEUE-FRONT, and ?TARGET-FUNCTION-3 mapping onto DEQUEUE-FRONT.

In order to choose between such equivalent mappings, SCAVENGER uses the heuristic of preferring analogies that leave the fewest methods of the *source* class[es] unmapped. This is a form of preferring the most general problem solution. In this case, that ranks STACK over DEQUE. SCAVENGER sorts the analogies before testing.

Testing analogies

SCAVENGER tests analogies by trying to run the initial transcript under the analogical substitution, stopping on success. In this case, it stops when it finds that the STACK analogy successfully runs the target example:

```
(SETQ X (MAKE-STACK))  ->  #<STACK #x306629>
(PUSH 'A X)  ->  #<STACK #x306629>
(PUSH 'B X)  ->  #<STACK #x306629>
(POP X) ->  B
(POP X)  ->  A
```

SCAVENGER returns this result as its interpretation of the target transcript:

```
Class map.
   TARGET -----> STACK
Method map.
   ?TARGET-FUNCTION-2 -----> POP
      arg-0 --> arg-0
   ?TARGET-FUNCTION-3 -----> MAKE-STACK
   ?TARGET-FUNCTION-1 -----> PUSH
      arg-0 --> arg-0; arg-1 --> arg-1
   The quality of fit to the source class was 0.600
```

---

[22] Double-ended-queue.

It inferred the following causal structure:
?TARGET-FUNCTION-3 takes arguments: NIL
returns result: RESULT
Function has description: ((NEW-OBJECT RESULT))
?TARGET-FUNCTION-1 takes arguments: (ARG-0 ARG-1)
returns result: ARG-1
Function has description: ((ADD-TO-COLLECTION ARG-0 ARG-1))
?TARGET-FUNCTION-2 takes arguments: (ARG-0)
returns result: RESULT
and produces side effects on: (ARG-0)
Function has description: ((REMOVE-FROM-COLLECTION RESULT ARG-0))

3.4.6 Handling failed index matches

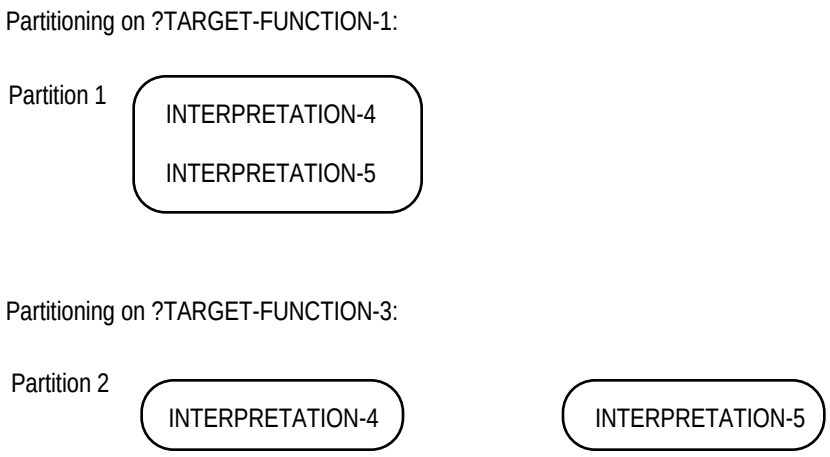
Sometimes, an index match will fail to produce a successful analogy. In such cases, the algorithm will try the next match in the sorted list produced in section 3.4. If matches fail to produce a successful analogy, the algorithm will eventually fail back to the root; since the root matches no target methods or classes, completing this analogy will effect an exhaustive search of the space of all possible analogies.

This process may produce redundant analogies, since each parent node implicitly includes all analogies that will result from its children. Since testing them adds to the program's overhead, SCAVENGER maintains a list of all interpretations it has tried and eliminates any new analogies that match interpretations in this list. By comparing new analogies to the high-level interpretations in this list, SCAVENGER can detect redundant analogies more efficiently than if it had to search all specific target-source mappings previously tried.

3.4.7 Updating the hierarchy

SCAVENGER's final task is to update the index hierarchy. It does so empirically, using the interpretations produced under section 3.4.4 as positive and negative training instances. An interpretation is positive if it contains at least one successful analogy. In this example, INDEX-NODE-2 produced two interpretations: INTERPRETATION-5 is the positive instance, and INTERPRETATION-4 is the negative instance.

SCAVENGER specializes the index that generated these interpretations by examining every target method that was not matched in the original index match. In this example, these are ?TARGET-FUNCTION-1, and ?TARGET-FUNCTION-3. It partitions the interpretations according to the interpretations the successful analogy gives these functions. Figure 3.18 shows the two partitions produced by specializing on different target functions. The description of ?TARGET-FUNCTION-1 was the same in both interpretations, therefore it does not distinguish them. The descriptions of ?TARGET-FUNCTION-3 differ between interpretations, so it creates two partitions.



Partitions of target interpretations produced by two candidate specializers
Figure 27

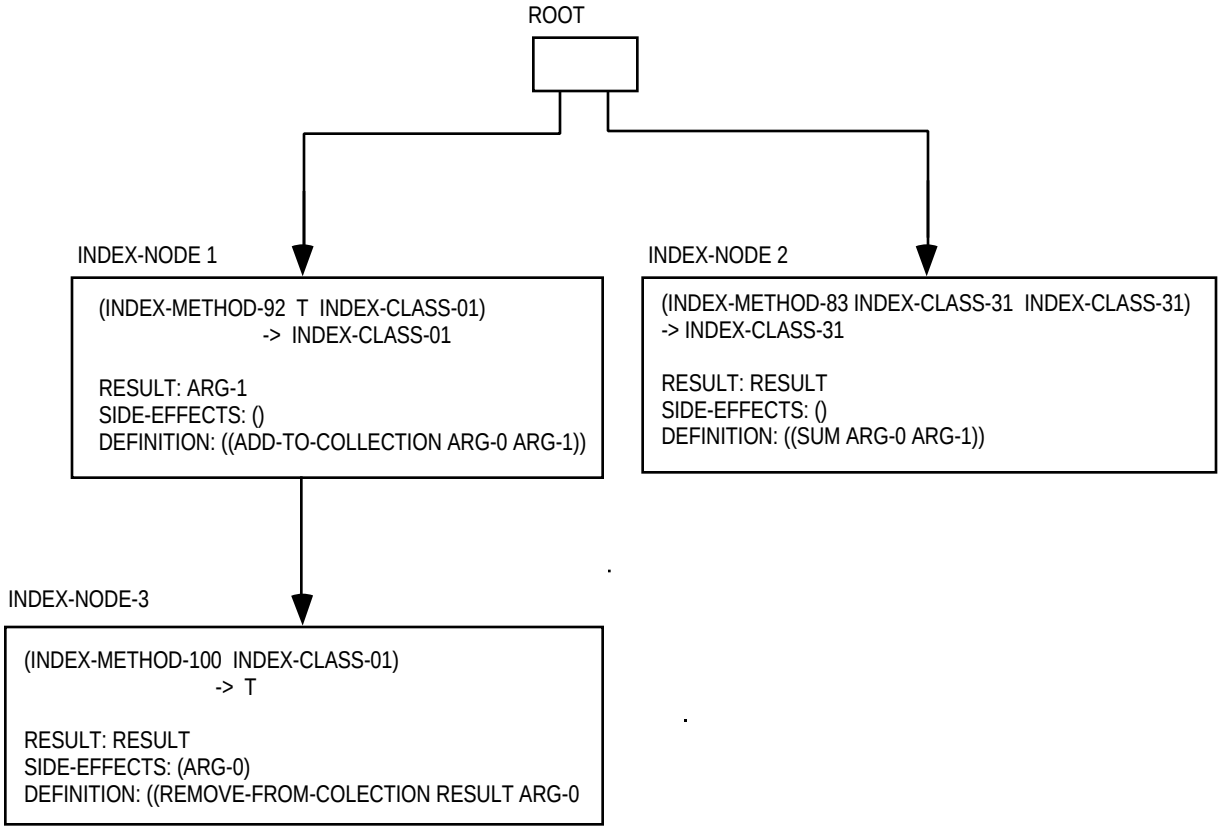
After selecting the best partition, SCAVENGER uses the method description from the successful analogy to specialize the parent index. It stores all matching source functions (whether they ran the target problem successfully or not) under this index. This is necessary, since future target problems may require these other sources. In our example, the best partition arises from splitting on ?TARGET-FUNCTION-3, using the interpretation provided by INTERPRETATION-5. This leads to the revised hierarchy of figure 27.

It is worth recalling that on future retrievals, matches with INDEX-NODE-3 will build upon matches with INDEX-NODE-1. Any match with INDEX-NODE-3 must include matches with INDEX-METHOD-92 and INDEX-CLASS-01.

Information-theoretic partition selection

The proper specialization of the index hierarchy was obvious in the above example; in general, this will not be the case. SCAVENGER chooses among candidate specializations using a variation of the information-theoretic evaluation function used in the ID3 induction algorithm (Quinlan 1986). ID3 constructs a decision tree in a top down fashion; it determines the test to perform at each node by choosing the test that gives the largest information gain in solving the example problem. The ID3 approach has the advantage of producing

Index Hierarchy



An updated version of the index of figure 18

Figure 28

small decision trees; this should help SCAVENGER produce highly efficient and selective index hierarchies.

Shannon (1948) defined the information content of a message as a function of the probabilities of the occurrence of all possible messages. This is computed as the sum across all potential messages, m of:

$$\sum - p(m) \log_2 p(m)$$

where p(m) is the probability of a message occurring. Applying it to SCAVENGER, we can compute the information content of a set of positive and negative interpretations by treating each interpretation as a message. The probability of a positive interpretation is simply the proportion of positive interpretations. In our example, we begin with one positive and one negative interpretation: ?INTERPRETATION-4 and ?INTERPRETATION-5. Consequently, the information required to complete the partial analogy produced by the match with INDEX-NODE-1 is:

$$- p(\text{positive}) \log_2 p(\text{positive}) - p(\text{negative}) \log_2 p(\text{negative})$$
$$= - 1/2 \log_2 1/2 - 1/2 \log_2 1/2$$
$$= -1/2 (-1) -1/2(-1)$$
$$= 1 \text{ bit}$$

We can also measure the information in each candidate partition of the set of interpretations produced by different candidate specializers of the original index. For example, partition 1 of figure 27 has information content of 0 bits; partition 2 has information content of 1 bit.

We can compute the information gain from partitioning the interpretation set on a given method description by subtracting the information content of the original set of interpretations from the weighted average of the information content of all components of the partition. The average is weighted according to the proportion of the original interpretations in each component of the partition.

For partition 1, the information gain = 1 - 1 = 0. For partition 2, the information gain = 1 - 0 = 1. It follows that partition 2 is preferable. This leads to the choice of ?TARGET-FUNCTION-3 as an index specializer. If there is no information gain on any partition, the index hierarchy remains unchanged.

SCAVENGER's use of information-theoretic evaluations differs from ID3 along a number of important dimensions:

1. It does not have a teacher to classify positive and negative instances, but must classify them itself.

2. Because it does not test all candidate analogies, its classification may be wrong in that certain interpretations may be falsely taken as negative instances. In attempting to learn meaningful generalizations from such ambiguous training data, SCAVENGER addresses a problem frequently found in realistic learning problems.

3. Unlike ID3, SCAVENGER does not produce a branch of the tree for each partition of the target problems; it only produces one branch for the partition known to contain the positive interpretation.

4. The classification of sources as positive and negative is only valid in the context of the current target problem. Will the index hierarchies produced in this fashion be useful on future targets that may lead to different interpretations?

Nonetheless, adaptation of ID3's metric was straightforward, and has given good results.

3.4.8 Conclusion

This chapter has described the architecture of SCAVENGER. Although it is a complex algorithm, it has evolved in a natural way from the interactionist model of source retrieval. In particular, many of the design decisions arose from an effort to integrate the three components of the interactionist model into a single program. These components were:

1. The interleaving of retrieval and inference in its assumption based retrieval algorithm.

2. The measurement of similarity using systematic evaluations of the structure of the target problem.

3. Updating of the index hierarchy on the basis of experience in solving target problems. This represents a transfer of knowledge of the relevance of source methods from the target to the source.

Another interesting aspect of SCAVENGER is its ability to represent and use of interpretation at multiple levels of abstraction. By groupings analogies with matching semantics. and reasoning with these high-level interpretations, rather than individual analogies, SCAVENGER can improve the efficiency of many of its operations.

As with any complex AI algorithm, SCAVENGER's behavior is difficult to predict. One danger is that the cost of computing SCAVENGER's heuristics will outweigh the gains made in prioritizing and pruning candidate analogies. Another danger is that target problems will be so diverse as to frustrate all of SCAVENGER's efforts to acquire useful indices. These and other issues can only be tested empirically.

3.5 Testing SCAVENGER

Due to its inherent complexity and the nature of its theoretical foundations, SCAVENGER raises a number of questions that require empirical investigation:

1. SCAVENGER is a complex program. Rather than retrieving sources on the basis of simple features, it matches entire target method descriptions with nodes in the index hierarchy. The heuristics it uses to rank hypothesized solutions construct and analyze graphs of the entire target problem. Will the complexity of matching indices and ranking candidate matches overwhelm any efficiency gains it makes through its learning and retrieval mechanisms? Will SCAVENGER's performance scale well as the size of the source base grows?

2. In selecting sources, SCAVENGER transfers properties to target problems on a non-monotonic basis, evaluating these assumptions to select among competing sources. It is inevitable that some of these assumptions will be wrong. Will the cost of constructing, testing and maintaining these alternative hypotheses nullify the gains of SCAVENGER's indexing system?

3. Like all hierarchical indexing systems, SCAVENGER forms categories of sources. However, instead of forming categories based on *source* properties, SCAVENGER forms them from patterns found in *target* problems. Each index in the hierarchy represents a set of function definitions that have appeared together in target problems. In the example of this chapter, SCAVENGER learned that functions that add items to collections (push, enqueue, etc.) frequently interact with functions that remove them (pop, dequeue, etc.). It follows that SCAVENGER's indices will be of general utility only if a sufficient number of patterns recur across targets. Will SCAVENGER find enough recurring target patterns in real problem domains to justify this approach?

4. SCAVENGER's performance depends heavily on the language used to describe source and target methods. Such a description language should help the indexing system form categories of similar sources, but should support sufficiently fine grained distinctions to discriminate sources. These two criteria must be balanced in the method description language if SCAVENGER is to form useful categories of source methods. Is the design of an effective description language a prohibitively hard problem, or will "natural," intuitive descriptions of source methods lead to effective retrieval?

5. Analogical and case-based reasoning programs must balance store-compute trade-offs: if a reasoner stores too many sources, the cost of retrieval will counteract the gains of re-using source solutions; if it stores too few, it will lack sufficient knowledge to solve a variety of target problems. SCAVENGER addresses this issue in two ways: Instead of saving entire problem solutions, it merely stores components of similar problem solutions, reconstructing a source for each new target. By exploiting similarities across target problems, this should reduce the number of sources that must be stored, but requires that SCAVENGER pay a greater overhead in reconstructing analogies when solving target problems. Second, the information-theoretic evaluation function used to select index specializers only adds an index node if it leads to a gain

in information about the current target problem. If there is no information gain, then SCAVENGER does not add a node; this should keep the index hierarchy from adding extraneous nodes. However, different target problems provide different values for this analysis; will this cause the hierarchy to grow excessively?

In attempting to find answers to these questions, I have tested SCAVENGER on three different problem domains. These are:

3.5.1 Interpreting examples of LISP function behavior

This domain, which provided the examples of this chapter, provided the initial tests of the algorithm; indeed, much of the original motivation for SCAVENGER's design came from this domain. The evaluations performed on this domain include:

1. Speed-up over successive trials. I have evaluated SCAVENGER's ability to improve retrieval using the standard machine learning approach of testing the performance of an untrained version of the algorithm on a set of test problems, training it on a different set of training problems, and measuring the improvement on the original test problems. This tests the algorithm's ability to generalize learned knowledge to new problems.

2. Scalability. I have tested SCAVENGER's behavior as the size of the source base grows. The complexity of exhaustive search grows exponentially with the size of the source base, SCAVENGER's learning ability must be able to manage this growth in complexity.

3. Reliability of heuristics. The algorithm described in this chapter "fails back" to more general index nodes if its first choice does not find a problem solution. Although SCAVENGER will find a solution if one exists, failure to find a solution quickly leads to a performance penalty resulting from the added overhead of maintaining and sorting multiple matches with the index hierarchy. What percentage of the time does SCAVENGER actually solve a problem on the first index match tried? If the hierarchy fails to find a solution, and SCAVENGER must

"fail back" to an exhaustive search, how bad can the added overhead get?

4. Reduction in variations in problem solution time. In many domains, untrained versions of the algorithm can experience several orders of magnitude variation in the times needed to solve different target problems. Can learning reduce the variation in solution times?

5. Comparison to source oriented retrieval methods. It is interesting to compare SCAVENGER to a more traditional, source oriented retrieval mechanism. I have implemented and tested a more traditional retrieval algorithm that constructs an index hierarchy independently of any information about target problems; it does this using a common clustering technique of comparing sources to each other to find effective partitions.

Chapter 4 discusses the results of these evaluations.

3.5.2 Finding bugs in failed procedures

Diagnostic problems are an interesting application for the SCAVENGER algorithm; the second test domain demonstrates SCAVENGER's to find bugs in failed procedures. In this domain, SCAVENGER is given a description of a procedure, along with its anomalous results. Using a source base of functions describing both successful and failed plan operators, SCAVENGER attempts to construct a mapping between the target procedure and these source descriptions that explains the cause of the plan's failure.

Chapter 5 discusses such an application. The domain I have used is the detection of bugs in children's subtraction skills. This builds on work by (Brown and Burton 1978a; Brown and Burton 1978b; Brown and VanLehn 1980) in the development of a theory of children's problems with arithmetic, and demonstrates SCAVENGER's generality by using it to solve an independently formulated problem. In testing SCAVENGER on this domain, I have used actual examples of children's buggy subtraction provided to me by Kurt VanLehn. Using this data, I have been able to

test SCAVENGER's ability to find useful, recurring patterns of analogy in real problem situations.

This test repeats tests 1, 2 and 4 from the previous domain. For reasons I will discuss in chapter 5, the other tests were either not relevant or not possible in this domain. However, I have added additional tests that explore SCAVENGER's ability to duplicate human solutions of these problems, and the problem of designing a language for describing source methods.

This domain demonstrates an interesting alternative diagnostic mechanism: most debugging programs are analytic in their approach, reasoning about problems to identify causes of failure. SCAVENGER, in contrast, does not reason about target problems, it merely learns to spot familiar patterns of source behaviors. It is interesting to note the effectiveness of such a simple problem solving strategy when coupled with a powerful learning mechanism.

3.5.3 Reasoning about Simulations

Because it tests its analogies by evaluating LISP programs, SCAVENGER should prove a valuable tool for simulation based reasoning about physical systems. Qualitative Process theory (Forbus 1984) is a model of the way in which humans reason qualitatively about simple physical systems, and a promising test application for this idea. I have applied SCAVENGER to the problem of interpreting the behavior of simple qualitative simulations. Unlike the two previous problems, I have not developed this domain beyond a small proof of concept. However, this small application further demonstrates SCAVENGER's generality and is discussed in chapter 6.

4
Evaluating SCAVENGER: Interpreting Tutorial Examples of LISP Methods

When you get interested in anything you start looking around for analogies; most scientists do, that's one of the best routes.

Alan Kay
Kay + Hillis
Wired Magazine, January 1994

This chapter describes a group of experiments designed to measure the effectiveness and efficiency of SCAVENGER's retrieval and learning algorithms[23]. These experiments applied SCAVENGER to the problem of interpreting tutorial examples of LISP function behavior; this was introduced and discussed in more detail in chapter 3. Very often, humans communicate general ideas through specific examples; somehow, the listener is able to interpret the example in such a way as to infer the intended generalization. One mechanism for performing this interpretation is the construction of analogies with things the listener already knows.

4.1 The selection and representation of test data

4.1.1 The source base

The tests described in this chapter use a source library of 22 classes and 152 methods. The library includes classes from a diverse selection of application domains. These domains, along with the number of classes and methods in each, appear in table 1. Appendix 1 describes these sources in detail, and also discusses the language used to describe method semantics.

In building the source library, I have attempted to avoid biases that might favor SCAVENGER by basing sources on pre-existing class definitions wherever possible. The data structure classes and methods are the basic abstract data types taught in intermediate computer science courses. The complex number, rational number and

[23] These tests were performed on a Macintosh II computer.

string classes simply embed corresponding LISP functions in CLOS class definitions. The object oriented data-base is part of the SCAVENGER implementation. The thermostat simulation is taken from an example in an AI textbook (Luger and Stubblefield 1993). Other domains, such as accounting, date arithmetic and investments are well constrained by "real world" definitions.

Domain	**# classes**	**# methods**
Accounting	2	13
Complex numbers	1	9
Data structures	8	55
Date arithmetic	1	13
Investments	3	9
Location using x-y co-ordinates	1	6
Object-oriented data base	1	7
Rational numbers	1	10
Strings	1	15
Thermostat-room-heater simulation	3	15
Totals	**22**	**152**

Source classes and methods listed by domain

Table 1

4.1.2 Representing target problems

Target problems are examples of the behavior of unknown target classes and methods. The goal of an interpretation is an analogy between the target and a known source that runs the target example correctly. As described in chapter 3, target problems are transcripts of LISP evaluations, where each evaluation has the form: <evaluated-form> -> <result>. The tests in this chapter used a set of 112 test problems, taken from across all the source domains. A typical target problem (taken from the rational number domain) is:

```
(setq x (?target-function-1 1 3)) -> (instance target-class 1)
(setq y (?target-function-1 1 4)) -> (instance target-class 2)
(setq z (?target-function-4 x y)) -> (instance target-class 3)
```

(?target-function-2 z) -> 7
(?target-function-3 z) -> 12

The intended interpretation is:

target-class = rational
?target-function-1 = make-rational
?target-function-2 = get-numerator
?target-function-3 = get-denominator
?target-function-4 = +

In creating test problems, I was guided by two goals:

1. To reflect a reasonable effort to communicate information about the target. Test problems, like good tutorial examples, may be complex, but they should not be intended to deceive the listener.

2. Like good examples, each problem should attempt to communicate a single idea.

Other than adhering to these criteria, I tried to produce a variety of test problems. Appendix 2 describes the test problems in more detail.

4.2 Learning in SCAVENGER

The first evaluation tests SCAVENGER'S ability to learn. The standard method of testing machine learning algorithms uses two sets of problems: one to train the learning algorithm, and the other to measure its improvement. This measures the ability of the algorithm to learn things that will generalize across problem instances. I have tested SCAVENGER according to the following procedure:

1. Randomly divide the complete set of test problems into separate test and training sets.

2. Initialize SCAVENGER's index hierarchy to a single, empty root node.

3. Disable learning, and measure the algorithm's baseline performance on both the test and training sets.

4. Enable learning.

5. Train SCAVENGER across repeated trials using the entire training set on each trial. Re-order the training set for each trial.

6. Disable learning.

7. Run the test set again to measure improvement.

I ran this procedure 6 times and averaged the results in order to smooth any variations that might result from the selection and ordering of test and training problems. In performing these evaluations, the variations proved relatively minor, and it was clear that this relatively small number of repetitions was sufficient to manage this noise.

4.2.1 Speedup over trials

The graph of figure 29 shows the algorithm's speedup over successive trials. SCAVENGER achieved a 70% improvement on the training set. Note also that the algorithm stabilized after the third run. At this point, SCAVENGER's learning algorithm found no further opportunities to improve the index, and stopped adding nodes.

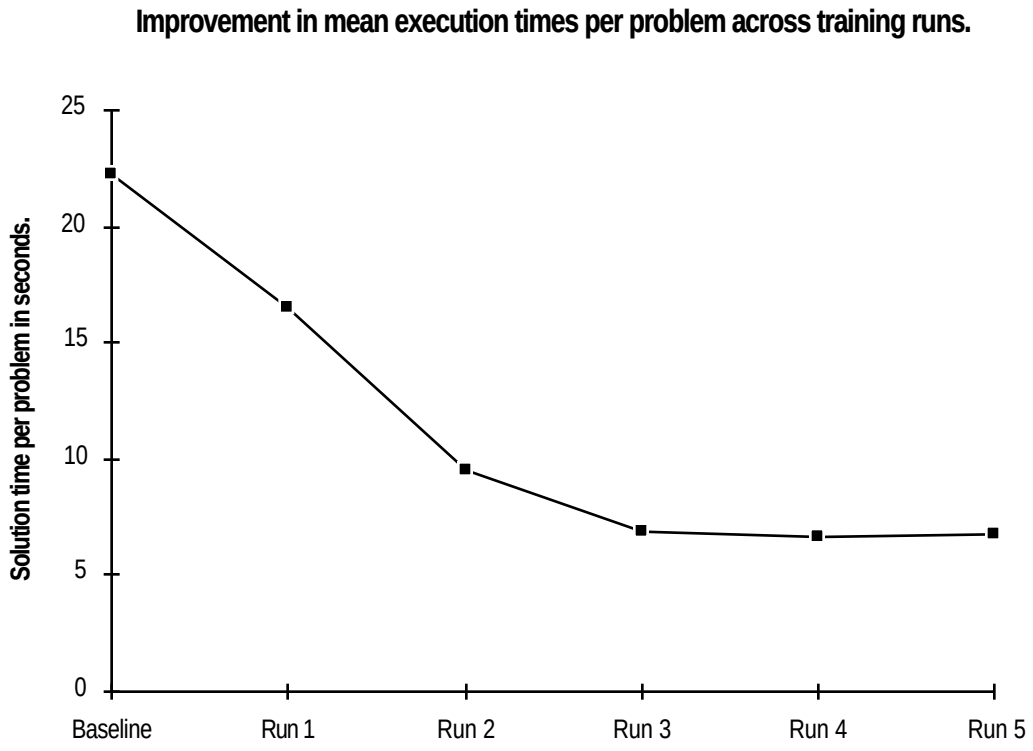


Figure 29

Another useful measure of SCAVENGER'S performance is the number of analogies that it tested before finding a solution. This is important, since many potential applications may require experiments in the physical world, which may be difficult, expensive or even dangerous. The ability to reduce needed experiments is a desirable property of an empirical learning algorithm. As shown in figure 30, the reduction in number of tests across training runs parallels the time speedup.

Reduction in number of analogies tested across training runs.

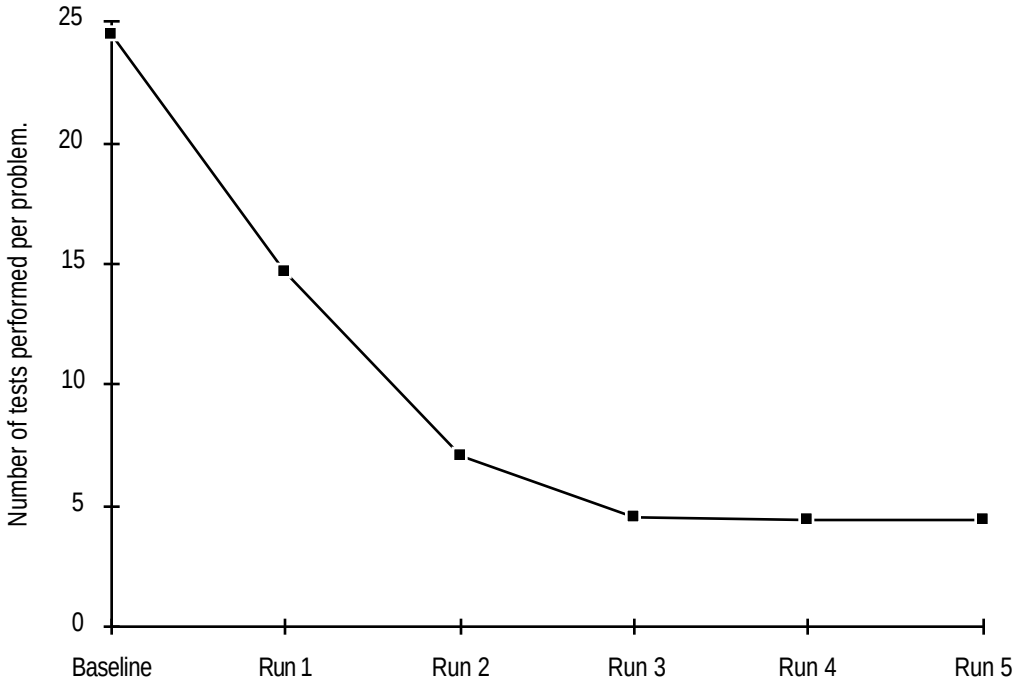


Figure 30

Runs on the test problems measure SCAVENGER's ability to generalize across problem sets. As might be expected, the speedups here are not as great, but remain significant. The speedup in time and reduction in analogies tested on the test problems are given in table 2. It is interesting to note the difference in the algorithm's improvement when measured as the time to solve problems as opposed to the number of tests performed. The explanation for this difference is in the added time required to try and match target problems with index nodes that failed to lead to complete analogies. While partial analogies that could not be completed do not add to the number of tests performed, they do take time to detect. This is more likely to happen if the problem is completely novel, so it should occur more frequently with the test problems.

	Time per problem	Tests per problem
Before training	19.36 sec	24.21
After training	10.5 sec	10.68
Improvement	**45.76%**	**55.88%**

Speedup on test problems

Table 2

4.2.2 Variations in target complexity

An interesting aspect of SCAVENGER's problem domain is the wide variation in execution times across different problem instances. On untrained versions of the algorithm (performing exhaustive search), the longest problem solution time was 684 seconds, while the shortest was 1 second. This variation is due to a number of factors, including the number of methods in the target problem, the number of arguments in target methods, the number of arguments for which the type is known, and the number of targets with identical signatures. An example of a short, but hard problem is:

```
(setq x (?target-function-1 7 4 1994)) -> (instance target-class 1)
(?target-function-2 x) -> 7
(?target-function-3 x) -> 4
(?target-function-4 x) -> 1994
```

The correct solution to this problem maps ?target-function-1 onto the make-date function of the date class, with ?target-function-2, ?target-function-3 and ?target-function-4 mapping onto get-month, get-day and get-year, respectively. It took 514 seconds for an untrained version of SCAVENGER to solve this problem. The main reason for its difficulty is the large number of target methods that take a single class instance as their arguments, and return a result of either integer or a more general type, as do three of the methods in the problem. For example, the date class alone has 8 such methods. There are 8^3 or 512 mappings between these functions and targets 2, 3, and 4. In addition, there are 6 different mappings between the arguments of make-date and ?target-function-1, giving a total of 3072 different syntactically allowable analogies with methods of the date class alone. Furthermore, several other domains can

produce syntactically allowable analogies with these targets which must be evaluated. In contrast, a problem like

(setq x (?target-function-1)) -> (instance target-class 1)
(?target-function-2 1 'a x) -> (instance target-class 1)
(?target-function-2 2 'b x) -> (instance target-class 1)
(cdr (?target-function-3 2 x)) -> b[24]

is much easier, since few sources can take an integer, symbol and target class instance as arguments, as with ?target-function-2.

These hard problems exerted a strong influence on the evaluation of SCAVENGER's performance. Figure 31 shows the improvement in median problem solution times over learning trials. The flatness of this graph in comparison to the graph of average solution times suggests that many of the benefits of learning in SCAVENGER come from its improvement on hard problems.

[24] The correct solution is: target = alist; ?target-function-1 = make-alist; ?target-function-2 = acons; ?target-function-3 = assoc

Reduction in median problem execution times across training runs.

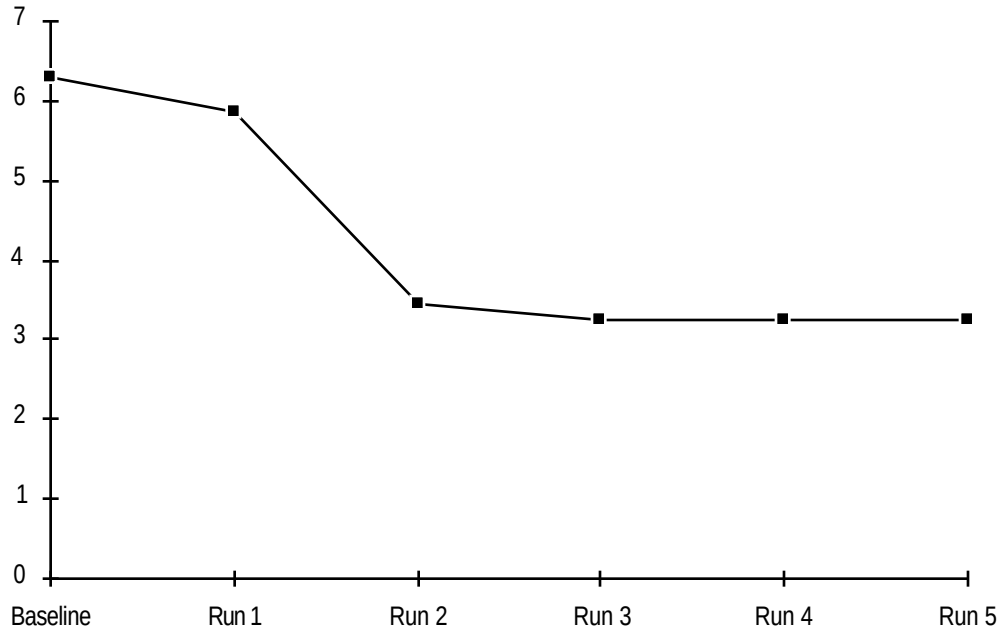


Figure 31

Inspection of individual times further supports this conjecture. Across training runs, the solution time for the hardest problem in any of the trials dropped from 686 seconds to 93 seconds. The reduction in the difference in times between the hardest and easiest problems is also indicated by a reduction in the standard deviation from the mean execution time across target problems. The untrained version had a standard deviation in execution times of 69 for the training problems; this reduced to 11 after training.

This also holds for the test sets. Here the length of time for the hardest problem dropped from 685 seconds to 137 after training. Standard deviation dropped from 48 seconds to 19 seconds.

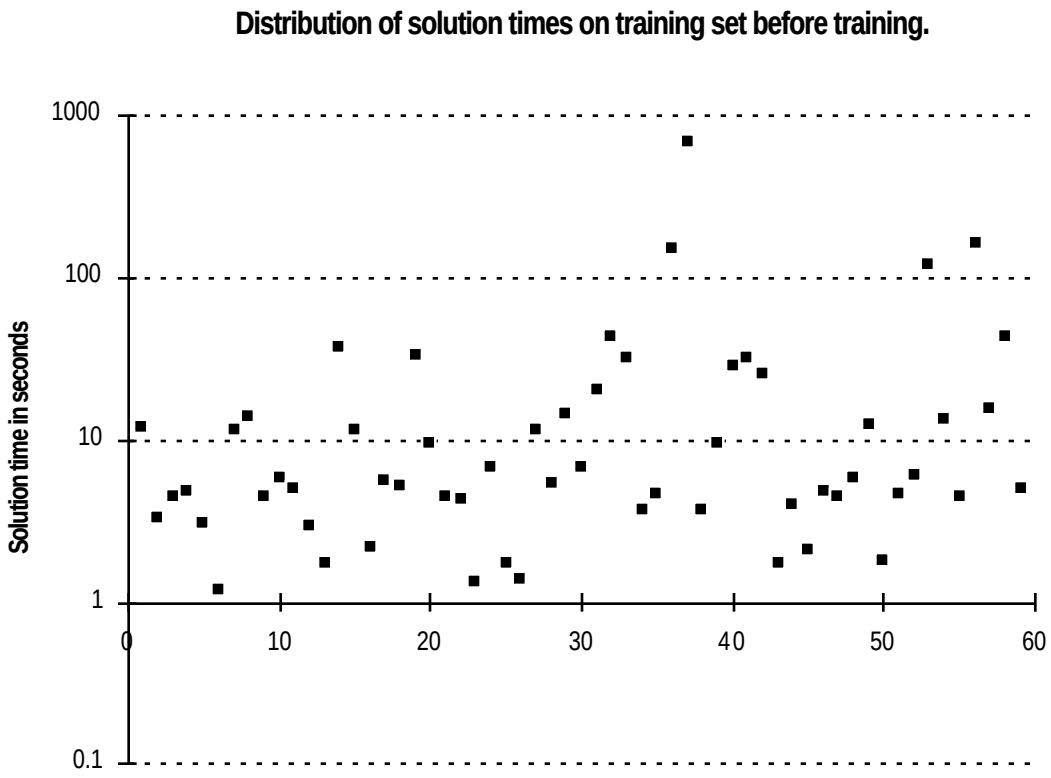


Figure 32

Figures 32 and 33 illustrate this effect of narrowing the range of solution times. Figure 32 shows the distribution of solution times for the set of training problems on an untrained version of the algorithm. The value scale is logarithmic. Figure 33 shows the distribution of solution times for the same problem set after training.

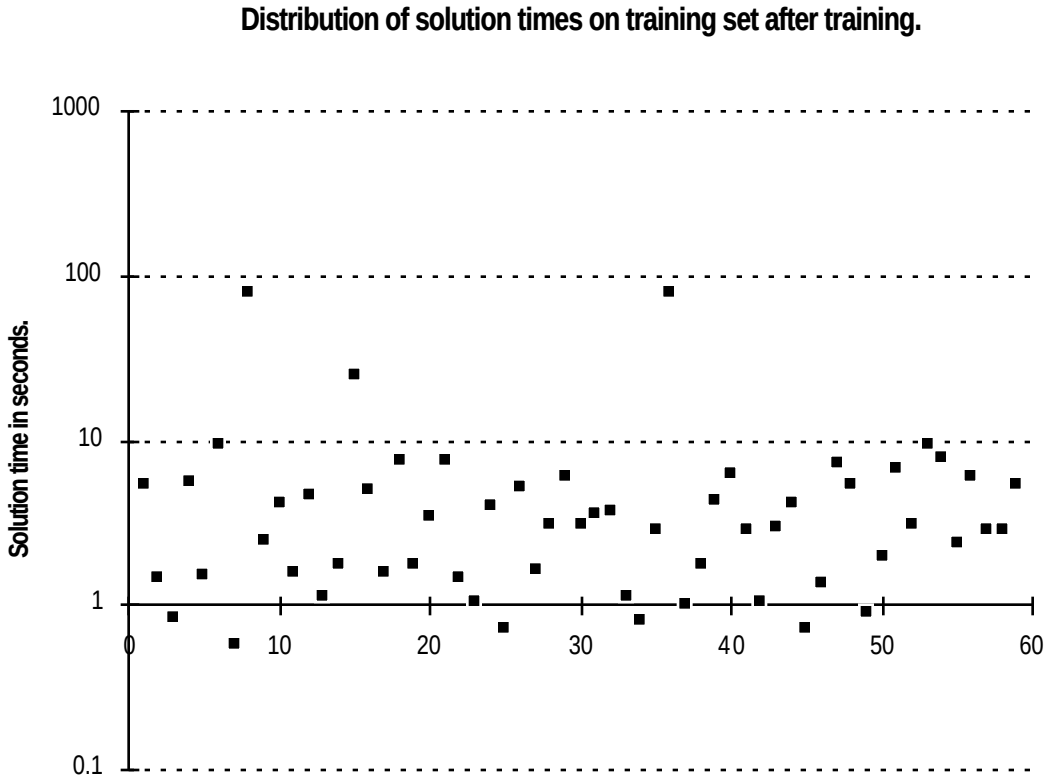
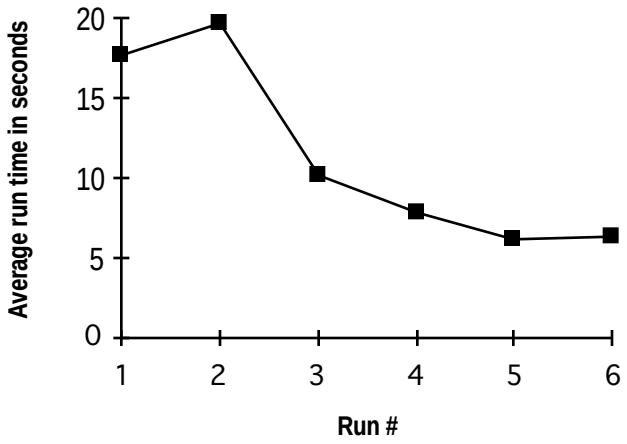


Figure 33

SCAVENGER's ability to improve performance on hard problems is explained partially by the fact that hard problems tend to have a larger number of target methods. Such problems are more likely to match nodes deep in the hierarchy, since there are more methods to choose from in matching the indices.

4.2.3 Fluctuations during learning

SCAVENGER learns incrementally, improving its performance over a succession of trials. Like many such algorithms, SCAVENGER's performance can fluctuate in the early stages of learning. Figure 34 shows a situation in which the algorithm's performance actually got worse before it improved. This case reflects data from a smaller source base and fewer target problems which exacerbated the early fluctuations.



Learning curve showing early fluctuations in solution times

Figure 34

The reason for this is that early versions of the index hierarchy may be excessively influenced by a few training problems, and cause the algorithm to be mislead. Note that SCAVENGER does recover on succeeding trials.

4.2.4 Properties of the hierarchy

SCAVENGER produces wide, shallow hierarchies. The shallowness follows from the fact that each level adds one method of a target problem to its parent; the maximum depth of the hierarchy can be no greater than the maximum number of target methods in any training problem. Examination of a typical index hierarchy reveals further details about its structure. A typical hierarchy was produced by 5 runs on a training set of 57 problems. SCAVENGER stopped learning after the fourth run. The resulting index hierarchy (which appears in Appendix 3 in its entirety) had the properties shown in table 3. Note that the average number of sources indexed at a node grows with the depth of the node in the hierarchy. This is inevitable given the combinatorics of analogical inference.

Level	# nodes	#leaves	sources/node
0	1	0	0
1	16	8	3.25
2	19	14	5.95
3	6	5	7
4	1	1	24

Distribution of sources among nodes of a typical index

Table 3

Further insights into SCAVENGER's performance can be gained by examining the number of problem solutions found at different levels of the index. In the 6 test sets discussed in this section, there were a total of 347 training problems and 325 test problems. The table 4 shows the distribution of solutions among the nodes that led to them.

Level	Final run of training set	Test set
0	32	53
1	127	109
2	144	146
3	38	17
4	3	0

Solutions found at each level of the index

Table 4

One surprising observation is that 32 of the *training* problems were not solved until SCAVENGER failed back to the root node. This is surprising since these are problems the algorithm has already seen 4 times. Because these problems did not match any other index nodes; it follows that SCAVENGER must not have created index nodes for them in the first place. This reflects the inherent conservatism of the information-theoretic learning heuristic: it does not form an index node unless it infers that this will give it some benefits. There are two reasons SCAVENGER would not form an index node from a target problem: either the problem incorporated enough constraints through numbers and types of arguments that it matched few sources (i.e., it was an easy problem), or the set of matching sources was

large but had identical descriptions so SCAVENGER was unable to make any distinctions among them. The overall strong performance of the algorithm suggests that the former explanation is the most likely.

The table also reveals much about SCAVENGER's ability to generalize learned knowledge to the test sets. 53 test problems failed to match any indices. This is to be expected, since the algorithm was not trained on these problems. For the same reason, the fact that few test problems were solved from matches at levels 3 and 4 seems reasonable. It is interesting to note that almost as many test problems were solved at levels 1 and 2 as in the training set. My explanation for this is as follows. One of the questions SCAVENGER raised was whether there would be enough recurring patterns of target problems to support the learning of generally useful categories of sources. While it seems unlikely that complex patterns involving many functions would recur across many targets, there should be smaller combinations of target methods that occur together frequently as *components* of many target problems. For example, it seems reasonable that methods that add items to a collection (push, enqueue, etc.) would frequently combine with methods that remove them (pop, dequeue, etc.). Such combinations should form a "kernel" shared by many target problems.

Inspection of the index hierarchy in appendix 3 reveals a number of such strongly coupled pairs occurring between levels 1 and 2. These include such pairings as:

Methods that get a property value and methods that modify a property value.

Methods that add to a collection and methods that remove from a collection.

Methods that add to a collection and methods that combine (sum) collections.

Methods that create a new object and those that retrieve a component of its structure.

These pairings are the "natural" sort of things that we might expect in examples intended to illustrate the behavior of a class. It is worth noting that SCAVENGER would pick these pairings out of the "noise" of the other methods appearing with them in target problems.

Another worthwhile observation concerns the number of target problems solved on matches with leaf nodes rather than internal nodes. After training, a total of 220 out of 325 training problems and 139 out of 374 test problems were solved on matches with leaf nodes. This suggests that once SCAVENGER found a set of target methods that appeared together frequently, it resisted any tendency to over-specialize these indices. This helps keep the hierarchy to a manageable size.

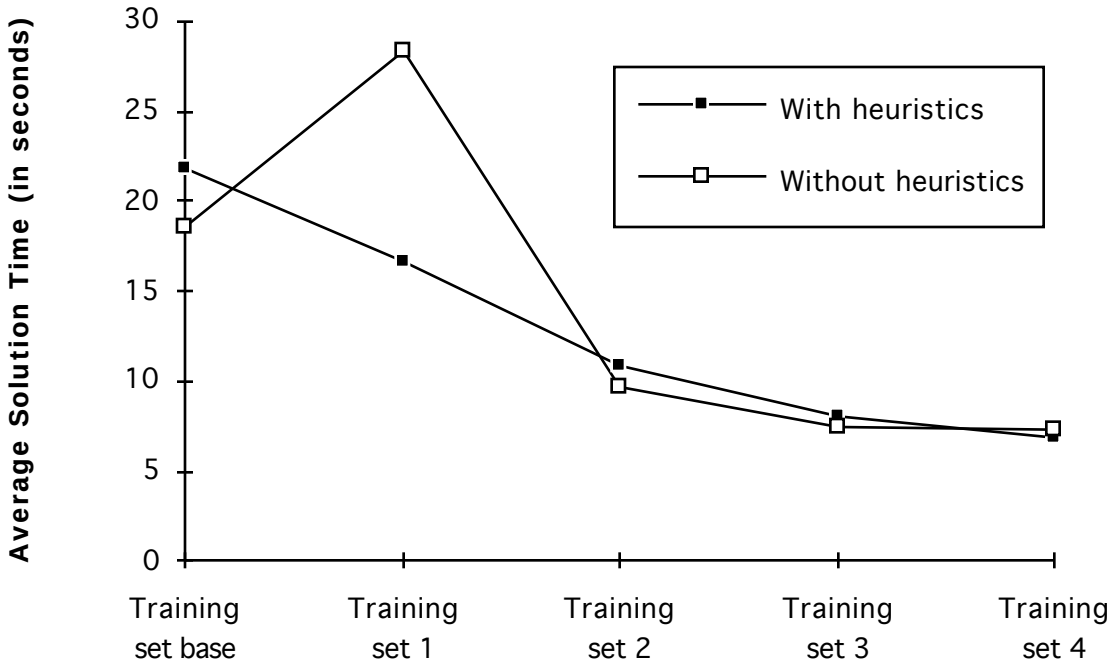
4.2.5 Effectiveness of SCAVENGER's heuristics

SCAVENGER generates all matches between target problems and indices in the hierarchy, including matches with both internal and leaf nodes. It then ranks all of these matches and tries them in order until it either finds a satisfactory interpretation of the target or runs out of matches. This is necessary, since failure of a given match to solve a problem does not mean it is not solvable, only that a particular class of analogies failed to solve it. The primary ranking heuristic evaluates matches according to their depth in the hierarchy. Where it has matched more than one node at the same level, SCAVENGER uses additional heuristics to select among these alternatives. Chapter 3 described the heuristics used in this problem domain in more detail.

Out of the total of 325 training problems in the 6 tests, the trained version of SCAVENGER solved 175 of them on analogies that resulted from the first node match tested. On the test set, 126 out of 347 problems were solved on the first match. This suggests that the combination of SCAVENGER'S learning algorithm and heuristic selection methods was effective in moving quickly to the appropriate solution.

It is interesting to ask whether this is due to the heuristics, or to SCAVENGER's learning algorithm. In order to test this, I compared the performance of the full SCAVENGER algorithm to a version that had its heuristics disabled. The non-heuristic version continued to favor analogies formed deep in the hierarchy, but did

not apply any additional heuristics. It simply tried analogies in the order they were produced. Figure 35 shows the performance of both versions of the algorithm across the same training and test sets. It is interesting to note that although the heuristic version performed better when the algorithm was partly trained, training eliminated any advantage provided by the heuristic rankings; in fact, the non-heuristic version is slightly faster, owing to the complexity of the algorithm's heuristic evaluation.



Comparison of heuristic and non-heuristic versions of SCAVENGER

Figure 35

Examination of the two version's performance on the test set gives similar results. It is interesting to note that the non-heuristic version outperformed the heuristic version on the initial test.

	With heuristics	Without heuristics
Initial test	20	17
Final test	12	11

Performance times (in seconds) on the test problems

Table 5

These findings are among the more surprising lessons of this research. One interpretation might suggest that the specific heuristics I have tested were not particularly effective. However, an alternative interpretation holds that SCAVENGER's learning algorithm is more important to the algorithm's performance than the heuristic ranking of candidate analogies. This idea, that an effective learning algorithm may have a greater influence on the performance than sophisticated search heuristics, might suggest a re-evaluation of the methodologies used in building knowledge-based systems.

4.2.6 Failure performance

If SCAVENGER fails to solve a problem with a match deep in the hierarchy, it will fail back to successively higher nodes until it either finds an answer or reaches the root. When it fails back to the root it implements an exhaustive search of the source base. In this fashion, SCAVENGER is sure to find an interpretation if one exists. This process, however, adds to the cost of search.

On evaluating each node match, SCAVENGER constructs all completions to the partial analogy produced in the match. There is a problem in that completions of the partial analogies produced by internal nodes will include analogies already generated by that node's descendents. SCAVENGER avoids testing duplicate analogies twice by keeping a list of all *interpretations*[25] of the target problem that it has tested. When completing the partial analogy produced by a node match, SCAVENGER

---

[25] As discussed in chapter 3, an interpretation is a high level description of a class of similar analogies. An interpretation is simply an assignment of argument types and semantic description to the methods in the target problems.

matches the completions against this list, and eliminates any that match. While this keeps it from testing the same analogy twice, it does add overhead to the search.

Empirical results show that SCAVENGER generates a significant number of duplicate analogies. On the tests performed in this section, the trained algorithm produced an average of 11.01 analogies per problem[26] on the training set, where 9.36 of these were unique. It generated nearly 2 duplicates per problem. On the test set, SCAVENGER produced 21.37 analogies per target problem, of which 16.58 were unique.[27]

One way to estimate the impact of this problem on SCAVENGER's performance is to give an unsolvable problem to both the trained and untrained versions of the algorithm. The trained version should take longer to exhaust the space, since it must pay the overhead of "failing back" through the hierarchy, whereas the untrained version will simply do an exhaustive search with no repetitions.

In performing this test, I created 7 problems that could not be solved with the existing source base. I did this by selecting 7 test problems, and changing the values of one or more of their results. I did not change the types of any arguments or results. The test took a baseline measure of the time it took SCAVENGER to fail on these problems by running them on an untrained version of the system. It then trained the program over 4 runs of a training set of 56 solvable problems, and repeated the failure test. The performance of the different programs on the 7 unsolvable problems is described in table 6:

	Untrained version	**Trained version**
Ave execution time	12 sec	37 sec
Total analogies generated	239	542
Unique analogies generated	239	239

The effect of learning on the SCAVENGER's failure performance

Table 6

---

[26] Only 4.47 of these were actually tested.

[27] And 10.68 were tested.

As this table indicates, the cost of failing back through the entire hierarchy is extremely expensive: more than triple the time for the untrained version. As the table also indicates, much of this comes from the problem of duplicate analogies. Unfortunately, I could find no way to fix this problem: SCAVENGER must generate the duplicates before it can detect and eliminate them.

4.3 Scaling in SCAVENGER

As the previous section demonstrates, SCAVENGER's learning algorithm has demonstrated its ability to improve performance after training on a number of target problems. An important question concerns SCAVENGER's ability to scale as the source library grows. I investigated the algorithm's scaling behavior by dividing the source library and test problems into a number of *test units*. A test unit consists of a single class, all of that class's methods and all of the target problems having methods of that class as their solution. I then measured SCAVENGER's performance on increasing numbers of test units. This evaluation followed the following procedure:

1. Randomly re-order the set of all test units.
2. For COUNT = 1 to the total number of test units, do:
    3. Load the classes and methods for the first COUNT test units into the source base.
    4. Initialize the index to a single root. Set learning to nil. Divide the problems from all test units into separate, randomly selected and re-ordered training and test sets.
    5. Run both test and training problems on the untrained algorithm to establish baseline performance.
    6 Turn on learning, and train the algorithm 4 times on the training set.
    7. Turn learning off, and re-run the training and test problem sets.

Figure 36 shows the results of this test, averaged over 6 trials to reduce the effect of ordering on the results. The top line reflects the exhaustive execution times averaged over both test and training sets; I did this to reduce variations caused by the small size of test sets in the early stages of the test.

The results of this test are encouraging, if not fully conclusive[28]. The top line shows the increases in average solution times for exhaustive search as the source base grows: it indicates the rapid growth one would expect for this type of problem.

---

[28] See chapter 5 for a more conclusive test of SCAVENGER's scaling abilities.

The bottom line shows the times for solving problems from the training set on the trained algorithm. It is remarkably flat, and also resists the fluctuations that have effected the other cases; this is encouraging. Unfortunately, due to variations in times, it is difficult to interpret the behavior of the test set on the trained algorithm (the middle line). The reason for these fluctuations is the distortion of the averages caused by hard problems in the test set.

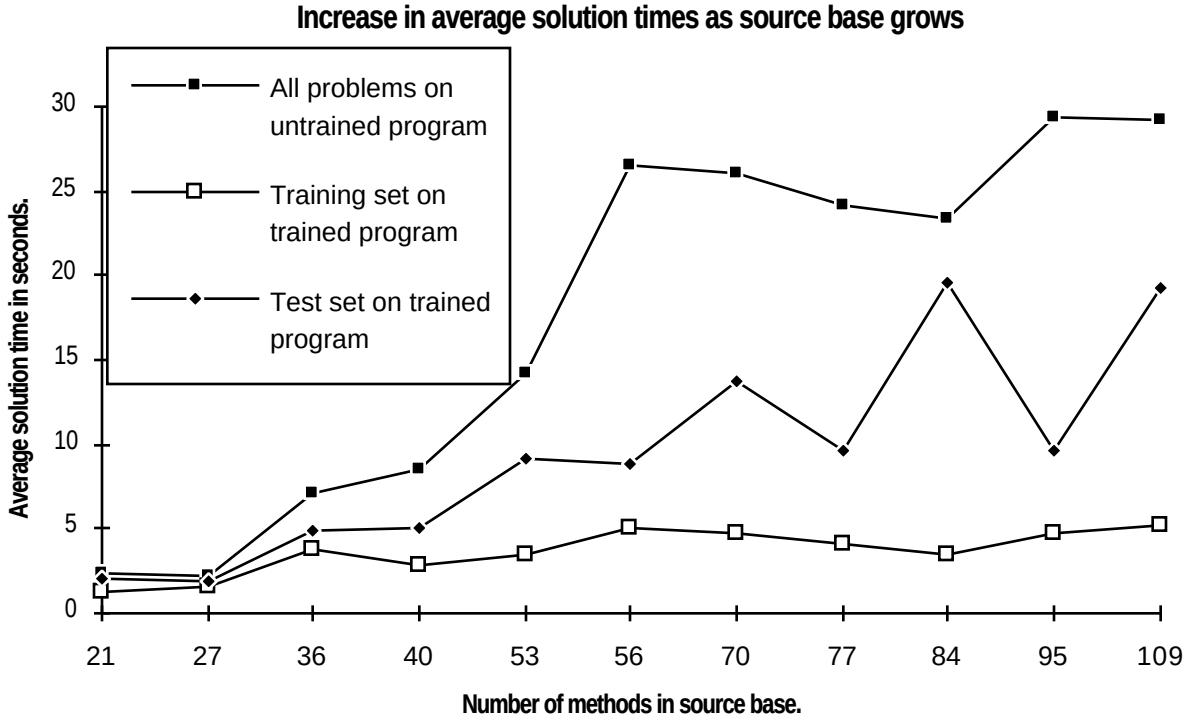


Figure 36

However, the good performance on the training problems does provide a positive answer to one question concerning the algorithm. SCAVENGER is such a complex program, that there was some concern as to whether the complexity of its execution would overwhelm any gains it made in pruning the problem space. These times clearly suggest that this is not the case, so long as SCAVENGER recognizes the target problem. If SCAVENGER recognizes the target problem, it performs very well, even as the size of the source base grows. If it fails to recognize the target problem, its performance can exceed that of exhaustive search.

4.4 Comparing SCAVENGER with source oriented clustering methods

As discussed in chapter 2, most retrieval algorithms for analogical and case-based reasoning construct indices by comparing sources to find those properties that form high-quality taxonomies. As a rule, they do not pay attention to the reasoner's experience in solving target problems. The justification for this is a desire to optimize retrieval across all targets, not just those the algorithm has seen. While this is a desirable goal, it may not be possible in realistically complex, ambiguous domains. My comparison of SCAVENGER and other approaches focuses on this issue of source centered vs. target centered indexing.

The major difficulty in performing this comparison is that most source-oriented retrieval mechanisms differ from SCAVENGER on two main points:

1. They rely on highly biased source description languages such as feature vectors (lists of values for single-valued attributes). SCAVENGER, in contrast, uses more complex relational information in the form of complete method descriptions, at each node.

2. They assume a fixed retrieval vocabulary, a list of the properties that can be used to construct indices. Learning selects the most discriminating properties from among this fixed set. In contrast, SCAVENGER adjusts its retrieval vocabulary as it encounters new target problems.

Modifying the interpretation problem to fit these requirements would simplify it excessively and reduce the validity of the comparison. Instead, I have chosen to write a clustering algorithm that captures the important features of source-oriented retrieval mechanisms, but that fits the problem representation used in SCAVENGER. This algorithm takes a top down clustering approach to building a hierarchical index. It is closest in spirit to UNIMEM (Lebowitz 1980; Lebowitz 1986; Lebowitz 1990), which was discussed in chapter 2, and has been patterned loosely after that algorithm.

4.4.1 Design of the comparison algorithm

The primary design constraint on the comparison system was to build an index hierarchy that could be searched using the current SCAVENGER algorithm. The comparison algorithm builds an index in a top-down fashion: each new child node specializes its parent by adding the information that further partitions the set of sources indexed by the parent.

A major problem in building this comparison program was to decide how to select the properties to use in constructing each child index. In the absence of a pre-defined retrieval vocabulary, the algorithm must bound the set of possibilities itself. SCAVENGER choose node specializing patterns from the source methods used in solving a target problem: it used the target problem to bound the set of possibilities it must consider. In contrast, the comparison algorithm takes a *class-oriented* approach, selecting node specializers from the methods of a given class. The algorithm builds the hierarchy incrementally, adding one class at a time according to the following algorithm:

1. Repeat for each class to be added to the hierarchy:

    2. Let CLASS = the current source class.
    3. Search the hierarchy, finding all index nodes that match some subset of the methods of CLASS.
    4. For each matching node, add the matching methods of CLASS to the indexed sources. Also, add CLASS to the list of classes stored under that node.
    5. From among the matched nodes, select those that were deepest in the hierarchy. Consider these nodes for specialization.
    6. For each node to be specialized, do:

        7. Retrieve all classes that have source methods stored under the parent index. Call these CLASS-LIST.
        8. Consider each method of CLASS in turn. Let the current method = METHOD. For each value of METHOD, partition CLASS-LIST into those that

have methods with signatures and definitions that match METHOD under some analogical mapping, and those that do not.
9. Evaluate each such partition using the information-theoretic evaluation function described in chapter 3.
10. Determine the information gain of the method giving the best partition. For each method of CLASS having the same rating, create a new child index representing its pattern. Store it and all other matching sources under the new child index.

Note that in specializing an index, step 10 will create a separate child for each favorably ranked method of the current class. I chose to do this instead of placing all these methods together in a single child index to avoid over-specialization.

4.4.2 Evaluating the comparison algorithm

The hierarchies constructed by the comparison algorithm vary greatly depending upon the order with which classes are inserted into the hierarchy. In order to reduce such effects, I ran the comparison algorithm 6 times, with a different order of insertion each time. Because target problems played no role in forming the indices, I tested each version against all 112 target problems. I compared this with a trained version of SCAVENGER's performance on the previously unseen test problems. I also included data on an algorithm that solved all 112 problems exhaustively, i.e., using no index hierarchy. Table 7 summarizes these results.

	Exhaustive Search	**Comparison Algorithm**	**Trained SCAVENGER**
Time per problem (seconds)	21	21	11
Tests per problem	24	15	11

Solution times for SCAVENGER and the comparison algorithm

Table 7

These results were surprising, since I had expected that the comparison algorithm would run at least as well as SCAVENGER. UNIMEM has been widely tested, and is recognized to be an effective algorithm; the comparison algorithm is close enough to it that it should have performed effectively. It is also interesting to note

that the comparison algorithm appeared much better when its performance is measured in terms of analogies tested per problem. These results make more sense if we compare the graphs produced by SCAVENGER and the comparison algorithm.

The comparison algorithm produced a larger hierarchy than SCAVENGER. The table 8 shows the number of nodes at each level, averaged across all trials for both algorithms.

	Comparison Algorithm	Trained SCAVENGER
Nodes at level 0	1	1
Nodes at level 1	18.6	15.2
Nodes at level 2	43.6	19.6
Nodes at level 3	48.6	7
Nodes at level 4	21.6	0.6

Profile of the index hierarchies for SCAVENGER and the comparison algorithm

Table 8

The impact of these different graph sizes is evident in a comparison of the number of analogies generated by both algorithms (table 9).

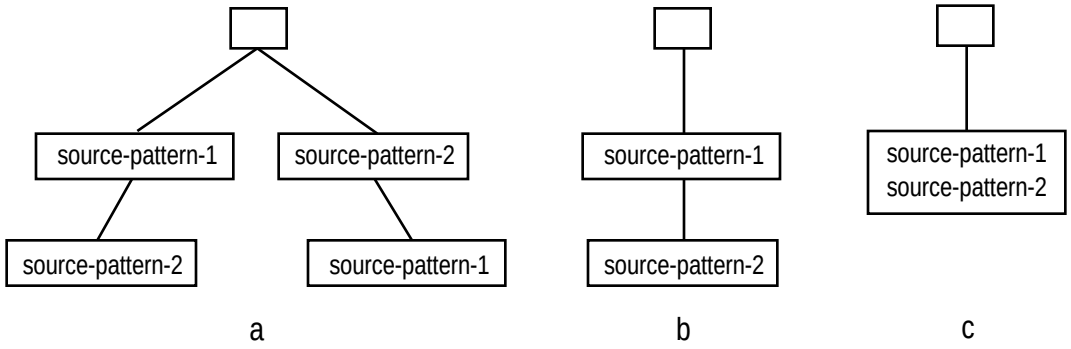
	Exhaustive Search	Comparison Algorithm	Trained SCAVENGER
Analogies per problem	47	48	11
Distinct analogies	47	27	9

Distinct analogies generated by SCAVENGER and the comparison algorithm

Table 9

The problem with the comparison version is that its graph is so large that it essentially guides the reasoner to consider all syntactically allowable analogies for each target. SCAVENGER's smaller hierarchy prunes away many syntactically allowable analogies on the basis of their failure to appear in previously seen targets. SCAVENGER builds its hierarchy in an effort to generalize from the target problems it has seen, while the comparison version, since it cannot access this information, must attempt to optimize retrieval over the entire space of possible targets.

Examination of the comparison algorithm's hierarchies casts additional light on this distinction. Figure 37.a shows a pattern that frequently occurs in indices produced by the comparison algorithm. This pattern would occur if step 10 of the algorithm found that source-pattern-1 and source-pattern-2 gave equally desirable partitions to the source base and made them sister nodes. Subsequent class insertions could further specialize these nodes to produce the pattern of 37.a. The problem with this formation is that matches with either leaf would match the other, leading to duplicate analogies. Although SCAVENGER eliminates duplicate analogies before testing, this adds to the time overhead, and accounts for the fact that the comparison algorithm did better on tests performed then on execution times.



Alternative index hierarchies

Figure 37

Eliminating one of the branches eliminates the duplicate analogy problem, but retrieves a fundamentally different set of analogies. The pattern in 37.b prevents targets that do not match source-pattern-1 from finding a match with source-pattern-2. Essentially, it says that source-pattern-2 should only be considered in conjunction with source-pattern-1. Item c of the figure is another solution. Changing the retrieval algorithm so that nodes can store multiple patterns and allowing the algorithm to match target methods with the largest subset(s) of them would reduce the duplicate analogy problem. However, comparing b and c illustrates another difference between SCAVENGER and the comparison algorithm. SCAVENGER will frequently produce index structures like 37.b. This enables it to represent the constraint that one source method only be considered in conjunction with another. This is a much stronger restriction on search, and I believe it accounts for much of the difference between SCAVENGER and the comparison algorithm. Target problems are a logical

source of such information, since they describe patterns of method interaction in actual use. I cannot think of a way of inferring this type of constraint from an examination of source classes and methods only.

4.4.3 Target oriented indexing and the interpretation problem

A deeper understanding of the differences between the two approaches requires further examination of the interpretation problem itself. In a sense, I stacked the deck against the comparison version by prohibiting it from using a pre-defined retrieval vocabulary. If I had specified a short list of features that could have been applied to all source methods, and also required that each target method be described in the same terms, a retrieval method like UNIMEM or COBWEB would probably work better than SCAVENGER.

The problem SCAVENGER solves is harder than those solved by many clustering algorithms or source retrieval mechanisms. Many aspects of the problem, such as the allowance of analogies that re-order arguments, contribute greatly to its complexity. The target transcripts provide very little information, aside from the types of a few arguments, that can be used to select appropriate sources. By explicitly disallowing a priori, static retrieval vocabularies, I further handicapped the source-centered approach. An important question is whether these restrictions are realistic.

I believe they are. Of the many assumptions made in traditional source retrieval systems, the assumption of a fixed, initially known retrieval vocabulary seems to be the most unrealistic. Intelligent programs should be able to cope with unspecified, unbounded, changing retrieval vocabularies. They should be able to handle under-specified problems in a reasonable way. SCAVENGER attempts to address these issues.

However, without some sort of bias, such problems would clearly be intractable. Instead of relying upon a fixed retrieval vocabulary, SCAVENGER uses the assumption that target problems will include enough recurring patterns to justify the target centered approach. It assumes that the space of target problems it will actually encounter is vastly smaller and contains more uniformities than the space of syntactically allowable target problems. In many domains, this is a reasonable assumption.

I believe that many realistic problems support this assumption. The problem of interpreting tutorial examples is clearly such a domain. It may be that one of the criteria humans use to create good examples is to combine elements that somehow "belong" together. Similarly, many diagnostic domains are characterized by patterns of co-occurring symptoms. Perhaps the existence of such uniformities in scientific problems is what Einstein was referring to when he commented that the surprising thing about the universe is that it *is* comprehensible.

It is also possible that a learner may actually project such patterns upon problems in the absence of sufficient indications to the contrary. One of the surprising things about SCAVENGER was its ability to find analogies that ran a source transcript correctly, but differed from my intended solution. Because target problems were so under-constrained, SCAVENGER had a great latitude in interpreting them. Where there is little order evident in a problem domain, an intelligent agent must create its own order. This ability is one of SCAVENGER's more unique properties, and is wholly consistent with the interaction theory of metaphor.

4.5 Conclusion

These evaluations have provided a number of insights into the the interactionist approach to metaphor and the three major conjectures derived from it (see section 1.3). This most strongly supported of the conjectures is the utility of empirical memory management. In order to be effective, empirical memory management must be able to find a sufficient number of recurring patterns in target problems, and represent them at a high enough level of generality.

These tests have shown that SCAVENGER not only learns well on repeated trials with the same problems, but acquires knowledge that generalizes well to new instances. The use of ID3's information-theoretic evaluation function to select index specializers does a good job of selecting patterns that are effective in discriminating sources and prevents the hierarchy from growing excessively. The algorithm scales well, with retrieval times remaining relatively flat as the size of the source base grows.

In comparing SCAVENGER's empirical memory management strategy to a more traditional, source-oriented approach, SCAVENGER significantly outperformed the comparison version. One reason SCAVENGER outperformed the traditional approach is that this problem provides little information for matching targets with sources. Since most traditional retrieval mechanisms rely upon an adequate retrieval vocabulary, this handicapped the comparison version. SCAVENGER does not require such a vocabulary; consequently, it may be useful in problem domains where little information is available about targets.

Another reason SCAVENGER outperformed the comparison version is that it is actually solving a different problem: traditional approaches try to optimize retrieval across all possible target patterns; SCAVENGER only tries to improve performance on patterns that are similar to those it has seen. This difference is a major source of SCAVENGER's strength. In many applications, such as the interpretation of tutorial examples, it is not necessary to cover all possible problems: most syntactically legal problems never occur in practice. In domains where the set of actually occurring problems is much smaller than the set of all possible problem instances, SCAVENGER should outperform techniques that strive to optimize performance across all possibilities. However, the penalty SCAVENGER must pay for this gain is a very poor performance on novel problem instances: if a problem does not match an index in SCAVENGER's hierarchy, the cost of solving the problem can be worse than a simple exhaustive search.

These tests also tended to support the effectiveness of assumption-based retrieval, although not as strongly. Inspection of the index hierarchy showed that many paths through the hierarchy could be discriminated by types and numbers of arguments alone, reducing the need for assumption based retrieval. The problem domain discussed in the next chapter provides a much more demanding test of this conjecture.

The third major conjecture of this dissertation argued that heuristics for measuring similarity should evaluate analogies in the context of target problems, often relying on more complex evaluations of the analogical inference's effects on the overall structure of the target problem. This conjecture was not strongly supported by the data. Although SCAVENGER's heuristics played an important role in

improving the performance of the untrained algorithm, they actually slowed the performance of the fully trained system slightly. This suggests that the power of SCAVENGER's learning algorithm is much more important than the sophistication of its heuristic evaluations.

5
Evaluating SCAVENGER: Diagnosis of Procedural Bugs

The imagination loses vitality as it ceases to adhere to what is real.

Wallace Stevens
The Noble Rider and the Sound of Words

SCAVENGER is a general analogical reasoner: it can be applied to any problem that takes the form of interpreting observations by finding similarities with existing knowledge. Diagnosis is particularly amenable to SCAVENGER's approach: we can view it as a process of interpreting symptoms to arrive at their underlying cause.

The domain I have chosen for this series of tests is that of diagnosing bugs in children's subtraction skills. This problem was first examined by (Brown and Burton 1978; Brown and VanLehn 1980; Burton 1982; VanLehn 1990). Briefly, their work explores the idea that most errors children make in arithmetic result from defective but consistently applied procedures. Teachers can identify these faulty procedures and use these diagnoses to correct failures. For example, assume a student performs the erroneous subtraction:

```
  634
- 468
  276
```

This error could be produced by a bug of borrowing when necessary but forgetting to decrement the digit borrowed from. Brown and VanLehn (1980) have described a large number of such procedural bugs, and demonstrated their ability to predict children's performance in simple arithmetic.

This problem domain has a particularly good fit to SCAVENGER's representations. The skill being debugged is procedural in nature: since LISP transcripts are sequences of interacting procedures, they are an ideal representation for procedural knowledge.

The tests[29] discussed in this chapter address three primary goals: The first is to further corroborate the results of chapter 4. The second goal is to provide a more demanding test of assumption-based retrieval. As will become evident, this domain allows SCAVENGER little opportunity to discriminate sources based on the numbers and types of arguments. Effective source selection must rely upon the ability of assumption-based retrieval to project relevant interpretations on target problems. Finally, this domain tests SCAVENGER's performance on realistic problems. Most of the tests described in this chapter were performed on data taken from tests performed on human subjects by VanLehn (1990). This provides valuable insights into an essential question raised by this research: if SCAVENGER is to perform effectively, problem instances must exhibit frequently recurring patterns of analogy; do such patterns occur with sufficient regularity in real domains?

5.1 Representing Target Problems and Analogical Sources

5.1.1 Representing diagnostic problems

The most straightforward way to describe SCAVENGER's representation and diagnosis of buggy subtraction problems is through an example of its diagnosis of the subtraction bug that appeared on the previous page. I have written a function, find-bug, that translates subtraction problems into a format suitable to SCAVENGER. Evaluating the LISP form

```
(find-bug 634 468 276)
```

breaks the subtraction down into a series of unknown operations on successive pairs of digits. The target transcript for this problem is:

```
(setq w (make-working-memory)) ->  (instance working-memory 1)
(#:G873 4 8 w) ->  ?
(#:G874 3 6 w) ->  ?
(#:G875 6 4 w) ->  ?
(show-result w) -> 276
```

In this transcript, w is bound to an instance of the class working-memory. Working-memory contains two slots: a borrow slot that contains the value (0 or 1 in correct

[29]The tests described in this chapter were performed on a Macintosh 6100 Power PC.

subtraction) that is to be decremented from the next column. The result slot accumulates the column results as the transcripts proceed; show-result returns the result value accumulated in working memory (providing the borrow slot = 0). The target method names, #G873, #G874 and #G875, are dummy names[30] for each operation. In interpreting the target, SCAVENGER maps these onto the appropriate sources. Those mappings form its diagnosis.

The source library consists of a number of methods for subtracting digits. Each of these takes two digits and an instance of working-memory, and returns an integer. For example, normal-subtract performs correct column subtraction, and has the definition:

1. If the borrow slot of working memory = 1, then decrement the top digit, and set borrow to 0.

2. If the top digit is less than the bottom digit, then add 10 to the top digit and set the borrow slot of working-memory to 1.

3. Subtract the digits, and concatenate their difference to the result stored in working memory.

The source base includes such error methods as borrow-no-decrement, smallest-from-largest, always-borrow, etc. For example, borrow-no-decrement has the definition

1. If the top digit is less than the bottom digit, then add 10 to the top digit (ignore the borrow slot of working memory).

2. Subtract the digits, and add the difference to the result in working memory.

The heuristics for ranking competing analogies minimize the total number of sources used and maximize the number of targets mapped to normal-subtract. These heuristics are simpler than those used in the domain of chapter 4. Because of that chapter's results on the relative unimportance of heuristic ranking of analogies in SCAVENGER, I chose to use simple heuristics and to pay more attention to the behavior of SCAVENGER's learning algorithm in these tests.

---

[30] Produced by the LISP function gensym.

Solving this transcript produces the mapping:

```
Method map.
  #:G874 -----> NORMAL-SUBTRACT
      arg-0 --> arg-0; arg-1 --> arg-1; arg-2 --> arg-2
  #:G873 -----> BORROW-NO-DECREMENT
      arg-0 --> arg-0; arg-1 --> arg-1; arg-2 --> arg-2
  #:G875 -----> BORROW-NO-DECREMENT
      arg-0 --> arg-0; arg-1 --> arg-1; arg-2 --> arg-2
```

This reproduces the behavior seen in the target problem:

```
(SETQ W (MAKE-WORKING-MEMORY)) -> #<WORKING-MEMORY #x2E8079>
(NORMAL-SUBTRACT 4 8 W) ->  6
(BORROW-NO-DECREMENT 3 6 W) ->  7
(BORROW-NO-DECREMENT 6 4 W) -> 2
(GET-RESULT W) ->  276
```

This solution diagnosed the problem as being produced by a combination of normal-subtract and borrow-no-decrement. The remainder of this section discusses the representation of sources and problems in more detail.

5.1.2 Selecting and representing buggy subtraction operators

Brown and VanLehn (1980) list 97 bugs that they used to diagnose problems in children's arithmetic. I have implemented 66 of these procedures as source operations (67 including the normal subtraction operator). Combinations of these 67 procedures capture an additional 23 of Brown and VanLehn's bugs. The reason I did not implement all 97 bugs concerns contextual issues in the application of operators. In implementing buggy source operators, I encountered three types of bugs:

1. Bugs that ignore context. These do not pay attention to their position in the problem or interactions with other operations (except those that could be expressed using working-memory's borrow variable). Borrow-no-decrement is an example of this type of bug. These fit directly into the representational scheme I have chosen, and I have implemented all 66 of them. Adding normal-subtract to these gives a total of 67 source operators.

2. Bugs that are combinations of bugs of type 1 with their usage constrained by context. For example, (Brown and VanLehn 1980) define borrow-only-once as "When there are several borrowers, the student decrements only the first borrower." (page 423) Borrow-only-once may also be thought of as an appropriate combination of normal-subtract and borrrow-no-decrement operations. I have not implemented these bugs directly, since SCAVENGER discovers them as combinations of type 1 bugs. An additional 23 of Brown and VanLehn's bugs were of this type, and SCAVENGER will find them in this indirect fashion.

3. Bugs that were inherently dependent on context in ways that could not easily be expressed as combinations of type 1 operators. These are bugs that made extensive use of contextual information, and could not be described under the representation I have chosen. An example of such a bug is borrow-decrementing-to-by-extras, which Brown and VanLehn define: "When there is a borrow across 0's, the student does not add 10 to the column he is doing but instead adds 10 minus the number of 0's borrowed across." (page 420) Rather than changing the representation in drastic, ad hoc ways (such as adding specialized counters or flags to working memory), I have chosen to ignore these bugs. There were only 8 of them, and they occur infrequently.

Appendix 5 provides a complete list of the operators used in these tests.

In the tests of this chapter, I have not allowed SCAVENGER to re-order arguments in forming analogies. Allowing the re-ordering of operators in this domain is not necessary, and in many cases, such as subtract-smaller-from-larger would be confusing.

Extensions to LISP's type hierarchy

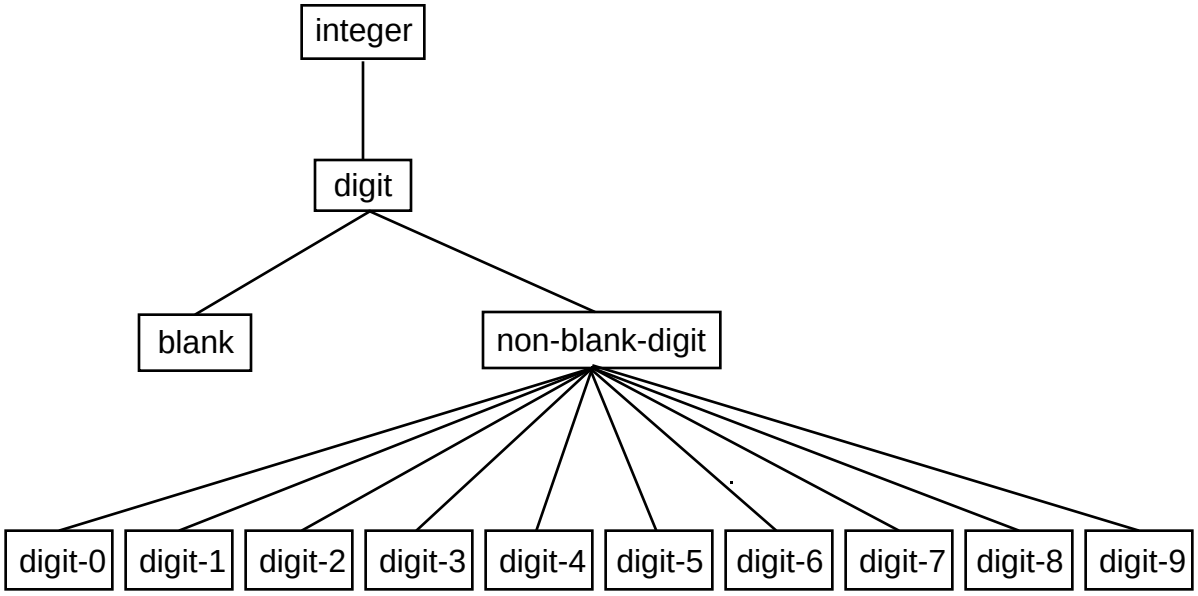
In representing source signatures, I extended the Common LISP type system to include the types shown in Figure 38. This extension adds a type, digit, which includes blank and the numbers 0 through 9. Digit has specializations of blank and non-blank-digit, and non-blank-digit is further specialized with a type for each digit. This is a reasonable extension to the LISP type system that gives SCAVENGER the ability to

make some initial distinctions between target problems. For example, the operator diff-0-n=0 describes the bug of assuming that 0 minus any other digit is always 0. Specifying its signature as:

diff-0-n=0: digit-0 x digit x working-memory -> digit-0

prevents SCAVENGER from trying to apply this operator in situations where it obviously does not fit.

It is worth noting that the the ability of argument types to distinguish source operators is fairly limited, even with these extensions to the type system. All sources have the same number of arguments, and the majority of them take any two digits as arguments.



Adding digit types to the LISP type hierarchy

Figure 38

Describing operators

The language SCAVENGER uses to describe source operators exerts a strong influence on its behavior. Assumption-based retrieval and the related learning algorithm use both argument types and high-level descriptions of function behavior to categorize similar sources. Because the description of sources is such a potentially strong

source of bias, I have made an effort to describe sources in terms that might reasonably be thought of as natural. The tests of chapter 4, relied on the sorts of descriptions people commonly give LISP functions. This problem domain supports a more systematic approach based on a cognitive model of the subtraction process.

Brown and Burton (1978) model children's knowledge of arithmetic as a procedural network. This is a network of interacting procedures which, when performing properly, performs correct arithmetic. Bugs originate out of failures in one or more components of this network. I have based my bug descriptions on a simplified version of this network.

When viewed at a high level, the procedural network for subtraction consists of the steps:

1. Select a column of digits.

2. If the top digit is smaller than the bottom digit, borrow from the column to the left and add 10 to the top digit.

3. Decrement the top digit of the column to the left by 1.

4. Perform the subtraction.

Although Brown et al further decompose these steps, this higher level of description allowed SCAVENGER to represent general categories of bugs. Based on this model, I have defined the following source descriptors:

1. normal-op. This is used only once, to describe normal subtraction.

2. transpose-error. An error in step 1, in which the student re-orders the operands to subtract the top digit from the bottom digit.

3. borrow-error. This describes any failure in step 2, such as failing to borrow, or always borrowing.

4. decrement-error. This describes any failure in step 3, such as forgetting to decrement.

5. subtract-error. A failure of step 4 such as assuming that n-0 = 0.

6. add-error. A failure of step 4 in which the student adds instead of subtracting.

I have described source methods using these predicates. Although most sources have been described using a single descriptor, some require combinations of them, giving a total of 11 different bug descriptions. Appendix 5 lists the operators used in this work and their descriptions.

5.1.3 Interesting properties of this problem and its representation

The most important aspect of this problem is the fact that the problem instances used to test SCAVENGER occurred naturally: they are the results produced by human children on subtraction tests and reflect no biases on my part.

Second, the problem domain is extremely complex. Given a three digit subtraction problem and 67 sources, there are a maximum of 67^3 or 300,763 cases that must be considered. While the number of cases is not unusually large for realistic problems, the overhead involved in evaluating the original transcript under each hypothesis makes their evaluation slow.

It is important to note that the problem representation differs from the approach taken in Brown et. al.'s original work. I have done this in an effort specifically to challenge SCAVENGER's ability to find meaningful patterns of source interactions. As an example, consider the definition of the bug smaller-from-larger, a bug that always subtracts the smaller number from the larger instead of borrowing. My definition of the buggy operator is:

1. If the top digit is smaller than the bottom digit, then subtract the top digit from the bottom.

2. Otherwise, the operator cannot be applied.

Because the operator cannot be applied if the top digit is larger than the bottom, most problems that involve a smaller-from-larger bug require that it be used in conjunction with the normal-subtract operator. SCAVENGER must find this combination of bugs through search. An alternative implementation of smaller-from-larger that performed normal subtraction if the top digit was larger than the bottom,

would have eliminated the need for such combinations of sources. If I had represented all bugs in this fashion, SCAVENGER's task would have been much easier: Instead of trying all combinations of buggy operators, it simply could have tried each operator in turn. However, I specifically wanted to test SCAVENGER's ability to find meaningful combinations of operations.

5.1.4 An overview of tests performed

The first of the tests compares SCAVENGER'S results to human diagnoses. This considers such questions as how often SCAVENGER finds the same results as humans, the reasons for its differences and the effect of learning on SCAVENGER's interpretations.

The next set of tests are similar to those performed in sections 4.2 and 4.3. They test SCAVENGER's learning ability, and the scalability of the algorithm as problem sizes grow. I have not repeated the comparison with the source centered retrieval mechanism described in section 4.4. The source-centered comparison algorithm inserts items into its index on a class by class basis. Since the sources in this test only have one class, working-memory, the comparison algorithm could not be applied to this domain.

5.2 Identifying Bugs

5.2.1 Identifying exemplars of different bugs

An interesting aspect of this domain is its inherent ambiguity. For example, the buggy result:

```
  62
- 37
  35
```

could be explained either by the bug smaller-from-larger, or by borrow-no-decrement. How did SCAVENGER resolve this ambiguity? The first test examines SCAVENGER's performance in diagnosing problems that were specifically intended to exemplify specific bugs. Brown and VanLehn (1980) list all the bugs included in their theory;

each bug description includes an example. Since these examples were intended to illustrate specific bugs, I assume that the authors made an effort to make them as unambiguous as possible.

This test used 46 examples of buggy operators taken from Brown and VanLehn (1980). Each example had been used to illustrate a different bug. An untrained version of SCAVENGER placed these 46 bugs in only 24 different diagnostic categories, agreeing with the human description 22 times. Note that given the available information, SCAVENGER's diagnoses are just as "correct" as the human's: they successfully reproduced the example behavior. This result indicates the extent of the domain's ambiguity: even problems intended to exemplify specific bugs allowed multiple interpretations.

The reason SCAVENGER found fewer interpretations than the humans is straightforward. The untrained version of the algorithm tried all the sources in the same order each time, quitting when it found a solution. This imposed a strong, if arbitrary, bias on the interpretation SCAVENGER gave to ambiguous bugs.[31]

The effect of learning on SCAVENGER's results is more interesting. After being trained on three repetitions of the problem instances, SCAVENGER placed them in only 20 different diagnoses. It found fewer different bugs after learning than before. The explanation is straightforward: after learning, the index hierarchy reordered the search space, producing a different set of interpretations.

These results are particularly interesting when we consider them in light of the interaction theory of metaphor. The interaction theory holds that the interpretation of both targets and sources evolves through their mutual participation in a series of metaphors (or analogies). SCAVENGER exhibits this type of behavior. After appearing in a number of comparisons (the training runs), the interpretations of many of the targets changed; after these same runs, SCAVENGER's interpretation of the relative importance of sources also changed. Although I did not program this

---

[31] If the algorithm had been allowed to continue search, it would have eventually produced a set of all interpretations that fit: this would have included the human interpretation.

behavior into the algorithm in any deliberate sense, its emergence further indicates the extent to which SCAVENGER is an instantiation of the interaction theory.

5.2.2 Diagnosing bugs in individual students

Burton (1982) describes the approach taken to handling this ambiguity in the original DEBUGGY program. Given an individual child's results on a set of subtraction problems, DEBUGGY constructs its diagnoses by:

1. Finding all bugs that correctly reproduce at least one of the student's wrong answers.

2. Eliminating bugs that are subsumed by other bugs. Bug X is subsumed by bug Y if Y will reproduce all problems diagnosed by X, and also diagnoses additional problems.

3. Produce candidate composite diagnoses. Since an individual may have several bugs in combination, DEBUGGY must potentially consider all subsets of the original set of bugs. Since this is frequently intractable, DEBUGGY only considers composite diagnoses containing no more than 4 individual bugs.

4. Select the diagnosis that best predicts the student's performance. DEBUGGY preferred diagnoses that avoided predicting wrong answers for problems the student answered correctly.

SCAVENGER implements a much simpler approach to interpretation than described above, treating each problem independently. The only record of multiple diagnoses it retains is implicit in the structure of its index.[32] It is, however, worth considering the possibility that the interactions of individual problems retained in the index structure might lead to a tendency to converge on diagnoses that match those produced by the DEBUGGY algorithm. Recall, for example, the tendency of

---

[32] It would be possible to apply SCAVENGER to step 1 of the above algorithm, simply by allowing it to find all interpretations of a each problem, although this simply uses SCAVENGER as an exhaustive search engine.

SCAVENGER's learning algorithm to narrow its selections as it learns (discussed in section 5.2.1).

I performed a simple test of this question, running SCAVENGER on the test results of 16 different students. SCAVENGER's examination of each student's results began with an untrained version of the algorithm. SCAVENGER made two passes through the student's results, using its learning algorithm to construct an index that summarized each student's bugs.

However, SCAVENGER showed no real tendency to converge to diagnoses that matched DEBUGGY's. On the 16 students, SCAVENGER came to agree with DEBUGGY in only 4 cases. Inspection of these cases suggests that the agreement was largely coincidental: all cases of agreement involved the bug, smaller-from-larger. This probably resulted from a combination of this bug's position in the source base and its frequent occurrence in student's performance.

This result is hardly surprising, given the sophistication of DEBUGGY's heuristic approach and the fact that SCAVENGER's source selection and learning algorithms make no explicit effort to find diagnoses that cover sets of target problems.

5.3 Learning and scaling

The results discussed in the preceding section underscore the ambiguity inherent in the problem domain, and the simplicity of SCAVENGER's inference techniques. This section focuses on the ability of SCAVENGER's learning and retrieval algorithms to improve its performance over larger problem sets. Given a population of problems taken from many different children, can SCAVENGER find enough regularities to significantly improve its performance?

One of the limitations of the problem domain discussed in chapter 4 was the fact that I had constructed the test problems myself. Although I attempted to avoid biases that might unfairly favor SCAVENGER, it is probably impossible to guarantee the success of such an effort. One of the attractive aspects of this problem domain is the availability of test data that was generated independently of this research. This data has provided a challenging test of SCAVENGER's ability to find regularities in large collections of data.

5.3.1 Selecting a set of test problems

The problems used in these tests were taken from VanLehn's (1990) "Southbay" study of children's subtraction skills. Prof. VanLehn generously provided me with the "raw" data from this study: this consisted of a group of children's results on 6 different subtraction tests, and included the test results for each child, a description of that child's diagnosed bugs, and other information on each child's age and class. In producing test problems for the evaluations performed in this chapter, I processed this data in the following fashion:

1. Extract the erroneous results from all test results, along with the diagnoses given them in VanLehn's original work. This produced approximately 7500 erroneous subtraction problems.

2. Eliminate problems involving 4 digit numbers from the list of problems. As will become evident, the combinatorics involved in trying combinations of operators led to very long solution times for the untrained algorithm; I have eliminated them in an effort to keep test times manageable.

3. Randomly select 500 problems from this final set. I then added the correct solutions to all 76 problems to this set.

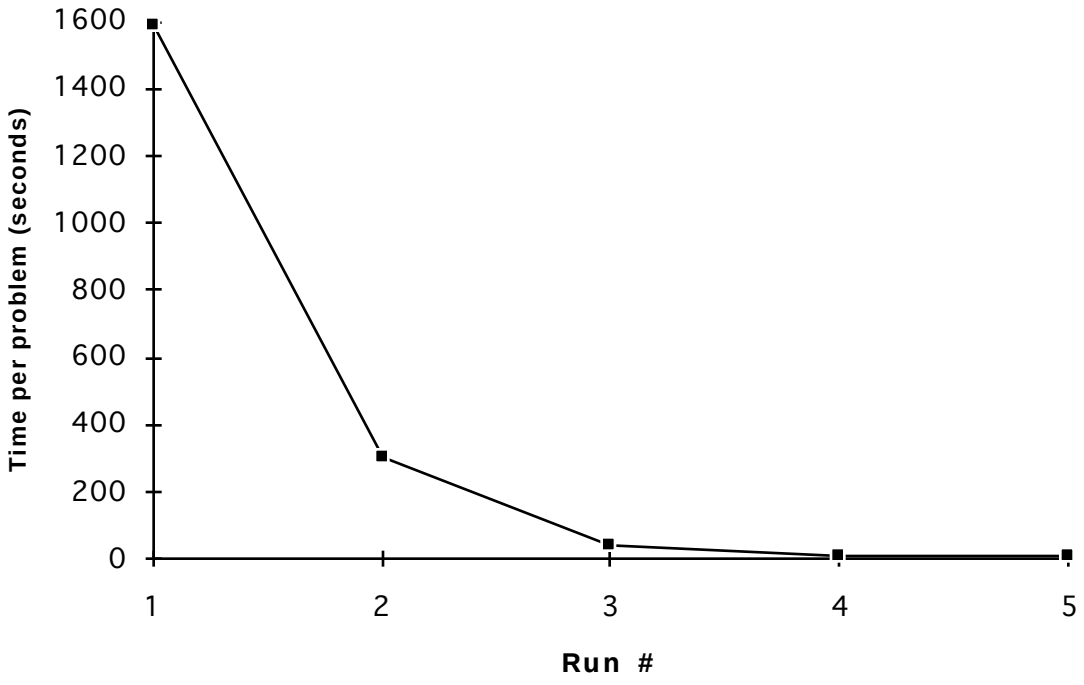
This produced a set of 576 test problems. I chose my test problems at random from this set. With the exception of eliminating problems involving four digit numbers, all acts of selecting and ordering test problems were random.

5.3.2 Learning

The learning test randomly chose 161 problems from the problem set described in section 5.3.1, and divided them into separate test and training sets. The training set contained 75 problems, and the test set 86 problems. Figure 39 shows SCAVENGER's improvements across repeated trials of the training data. Although it is difficult to read exact times from the figure, the trained version took an average of 11 seconds per problem. The longest solution time for a training problem was 30.5 seconds, the

fastest time was just over 2 seconds. Here, the improvements are even greater than those of chapter 4.

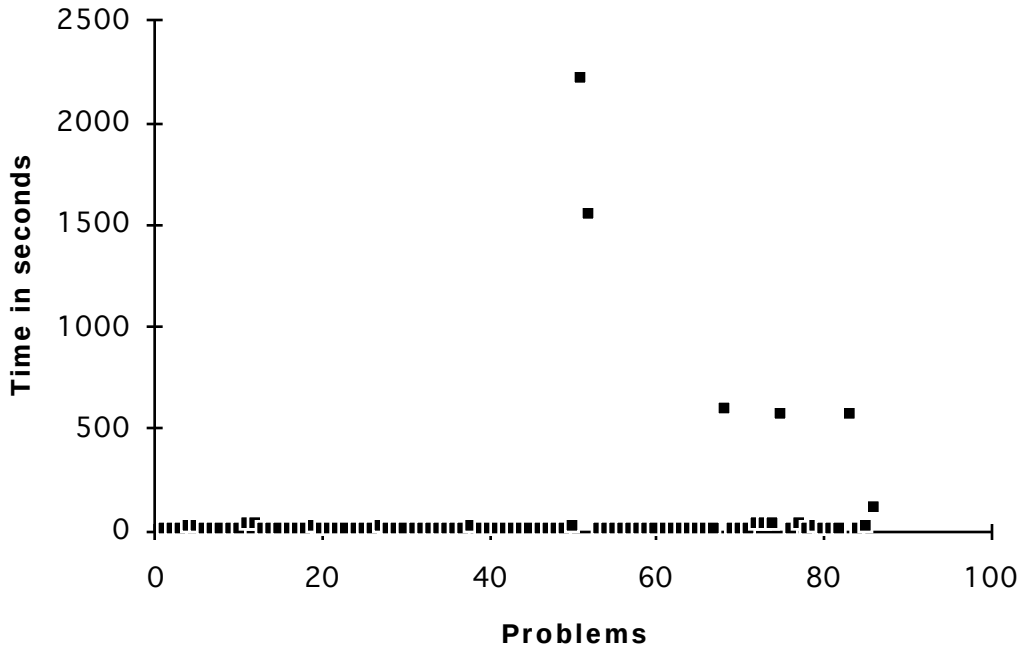
This is due to the extreme difficulty of the problem for the untrained algorithm. SCAVENGER could eliminate few buggy operators using such information as numbers and types of arguments. All operators took two digits and an instance of working memory as arguments. Although some operators could only apply if one of the digits equaled a specific value (usually 0 or 1), SCAVENGER still had to consider a large number of buggy operators. The untrained version of the algorithm wound up generating an average of 3096 combinations of operators per problem; after training, SCAVENGER generated only an average of 78 hypotheses to solve a problem.



Improvement in average problem solution times across repeated trials on training set
Figure 39

In applying SCAVENGER to the test set (problems that the learning algorithm had not seen), the untrained version took about 1500 seconds per problem; this

improved to to an average of 76 seconds per problem after training on the separate training set. Although this is a strong result, it is worth noting that it was skewed by the presence of a small number of problems that SCAVENGER did not recognize, and, consequently, took a very long time (2220, 1556, 593, 569, 568 and 116 seconds). If these were eliminated, the average time on the test problems drops to under 11 seconds. Figure 40 illustrates the variation in execution times of the trained algorithm on the test problems.



Distribution of solution times on test problem set

Figure 40

These results further support the effectiveness of SCAVENGER's learning and retrieval algorithms. Its performance also indicates the effectiveness of the generalizations provided by the description language used to index functions and the existence of a sufficient number of recurring problems in the population of children.

The extent to which recurring problems influenced the outcome is characterized in table 10. This summarizes the interpretations SCAVENGER found for the

combined training and test sets. Although SCAVENGER was certainly helped by recurring bugs, there were enough infrequently occurring bugs to indicate that the algorithm's abilities to generalize also contributed to its performance.

Bug	Occurrences
1. SMALLER-FROM-LARGER	69
2. NORMAL-SUBTRACT	15
3. BORROW-NO-DECREMENT	12
4. N-N=1-AFTER-BORROW	6
5. N-N=9-PLUS-DECREMENT	4
6. DECREMENT-1-TO-11	4
7. ZERO-AFTER-BORROW	4
8. ADD-BORROW-CARRY-SUB	3
9. BORROW-ACCUMULATE-DECREMENT	3
10. DECREMENT-ON-BORROW	3
11. BORROW-INTO-ONE=10	3
12. BORROW-FROM-ZERO	3
13. DIFF-N-0=0	3
14. ZERO-INSTEAD-OF-BORROW	3
15. BORROW-INTO-0-IS-9	3
16. DONT-DECREMENT-ZERO	2
17. SMALLER-FROM-LARGER & SUB-ONE-OVER-BLANK	2
18. ADD-INSTEAD-OF-SUB	2
19. ADD-NO-CARRY-INSTEAD-OF-SUB	2
20. BORROW-ACROSS-ZERO-TOUCHED-0-N=N	2
21. ALWAYS-BORROW & N-N=9-PLUS-DECREMENT	2
22. DIFF-N-N=N	2
23. BORROW-FROM-BOTTOM	1
24. BORROW-FROM-BOTTOM & BORROW-ACCUMULATE-DEC	1
25. BORROW-FROM-BOTTOM & ZERO-AFTER-BORROW	1
26. ADD-BORROW-CARRY-SUB & BORROW-ACCUMULATE-DEC	1
27. BORROW-UNIT-DIFF & SUBTRACT-ACCUMULATED-DECS	1

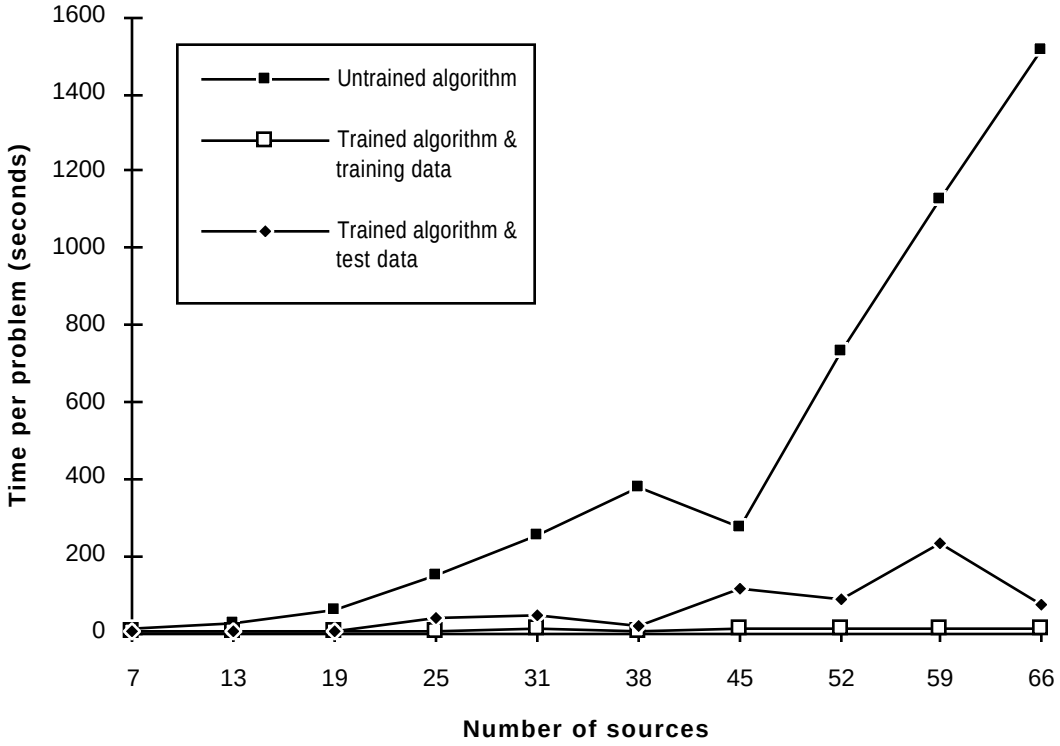
28. BORROW-UNIT-DIFF	1
29. SMALLER-FROM-LARGER & ALWAYS-BORROW	1
30. BORROW-TREAT-ONE-AS-0	1
31. TREAT-TOP-ZERO-AS-10	1

Summary of SCAVENGER's Diagnoses

Table 10

5.3.3 Scalability

The algorithm also continues to perform well on scaling tests. To examine its scaling behavior, I divided the source base into 10 sets of 6 or 7 buggy operators each; each of these "test units" included approximately 16 problems that could be solved with those operators. Repeated trials "grew" the source base by adding test sets. Each test randomly divided the problems into separate test and training sets and repeated the learning test described in 5.3.1. Figure 41 shows SCAVENGER's run times on this test.



Scaling test results

Figure 41

The performance of the untrained algorithm shows the exponential rise in complexity we would expect of exhaustive search. However, the performance of the trained algorithm remains nearly flat when applied to problems it has trained on. The trained algorithm also remains efficient on the test problems, although the results do show more fluctuation. As in chapter 4, the reason for this greater variation is the existence of small numbers of novel problems that caused the algorithm to perform badly.

5.4 Conclusion

This chapter provides further insights into the performance of SCAVENGER. SCAVENGER's application to a diagnostic domain supports the general applicability of the algorithm.

The most important result of this evaluation was a further corroboration of the effectiveness of SCAVENGER's learning algorithm. Because all sources had the same number of arguments, and because type information failed to distinguish sources adequately, the untrained algorithm had to manage large numbers of sources. SCAVENGER improved its performance by two orders of magnitude over the training runs.

Another goal of these tests was to further explore the effectiveness of assumption-based retrieval. Because information about types and numbers of arguments did little to distinguish sources, most of the improvements in performance came through assumption-based retrieval's process of projecting familiar patterns of analogy on source data and confirming them through experiment. The success of this approach on this domain results from two factors: the first is the structure of the domain itself. Without sufficient recurring patterns of analogy in the problem domain, assumption-based retrieval could do little to improve performance. One of the important results of this series of tests was an illustration of the extent to which such recurring patterns can be found in realistic problem domains. The second source of the effectiveness of assumption-based retrieval is the ability of the learning algorithm to construct effective hierarchies. Although SCAVENGER uses a variation of a well tested learning algorithm, ID3, I did modify it in several critical ways. Perhaps the most significant of these was in the use of local information obtained from a single problem to generate specializations to the index hierarchy. This contrasts with ID3's use of a global analysis of entire sets of test problems to construct its decision trees. It is not obvious that the learning algorithm would continue to function well if restricted to such limited training information. The results strongly suggest that it does.

It is interesting to contrast this approach with the more analytic approach taken by diagnostic expert systems. SCAVENGER does not reason about problems; it simply looks for viable patterns of failure, generalizing and remembering these patterns. This "analogize test and remember" approach could be useful in domains where models of the failure behavior of individual system components are known, but rules for reasoning about the interactions of these component failures may be unknown. For instance if we are debugging devices where failure of one component

could cause or interact with failures of another component, SCAVENGER could be a powerful tool for discovering and recording recurring patterns of interacting failures.

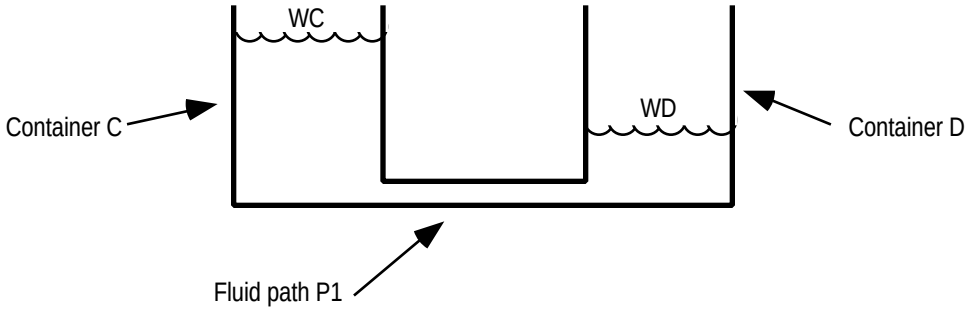
These tests corroborated the effectiveness of the interactionist retrieval mechanism on a difficult, "real-world" problem. Applying SCAVENGER to data produced by human children, demonstrates both its generality and its effectiveness on a difficult, naturally occurring problem.

6
Evaluating SCAVENGER: Reasoning About Simulations

All our reasonings concerning matter of fact are founded on a species of analogy which leads us to expect from any cause the same events which we have observed to result from similar causes.

David Hume
An Inquiry Concerning Human Understanding

There is a close relationship between the interpretation problem as formulated in SCAVENGER and simulation based reasoning. In testing hypothesized analogies, SCAVENGER executes sequences of CLOS methods. Object oriented languages, like CLOS, are a powerful tool for implementing simulations. SCAVENGER makes inferences and tests them by constructing and running CLOS programs; simulation-based problem solvers implement a similar process. This chapter briefly describes the way in which SCAVENGER can be applied to reasoning about qualitative simulations. Unlike the preceding problem domains, this work does not attempt to evaluate SCAVENGER's performance with large source bases and problem sets, but rather serves as a simple proof of concept and a further demonstration of the algorithm's flexibility.



A simple flow system
Figure 42

Qualitative Process (QP) theory is a tool for describing and reasoning about the qualitative behavior of physical systems (Forbus 1984). It allows us to represent relationships between quantities, the way in which quantities change in a system, and the interactions of physical processes. An example, taken from (Forbus 1984) illustrates its capabilities. Figure 42 shows a system of two water tanks (C and D)

connected at their bases by a pipe (P1). WC and WD represent the water levels in each tank at a given time. Given this situation, humans will recognize that water will flow from left to right through the pipe until the levels in the containers are equal. We infer this by reasoning qualitatively about the system, not by applying quantitative laws. QP theory is a formal model of this type of qualitative reasoning.

QP theory represents qualitative relationships between quantities. A quantity consists of two parts: an amount and a derivative; each of these, in turn, has an amount and a sign. Rather than doing arithmetic with amounts and derivatives, QP theory reasons qualitatively about their relationships. In figure 42:

amount(WC) > amount(WD)

When water flows through P1, the derivative of the quantity WC takes on a value of -1, indicating that the level is falling. Similarly, the derivative of WD is +1. Derivatives can take on values of +1, 0 or -1, indicating the direction of a change, but not its amount. Reasoning about the flow process leads to the conclusion that eventually,

amount(WC) = amount(WD)

and all the derivatives become 0.

Quantities are related by functional relationships that show how changes in one quantity influence the amount of another. In the example of figure 42, a change of water level causes a corresponding change in the amount of water in a container. If w is the water in a container, this relationship is written:

level(w) $_{Q+}$ amount-of(w)

if the relationship is inverse, use the operator: $_{Q-}$.

Process is a central notion in QP theory. A process changes properties of objects over time. In figure 42, the flow process changes the water levels in the tanks. A process consists of:

1. The individual objects it applies to.

2. Preconditions and quantity conditions that must be met if it is to apply.

3. Functional relations between properties of objects imposed by the process.

4. Influences the process exerts on object properties.

An influence can be either positive (I+), in which increases in one quantity cause another to increase, or negative (I-), in which one quantity causes another to decrease.

In the current example, the flow process is described (Forbus 1984):

process fluid-flow
Individuals:
src a contained liquid
dst a contained liquid
path a fluid path, fluid-connected(src, dst, path)
Preconditions:
Aligned(path)
Quantity conditions:
amount(pressure(src)) > amount(pressure(dst))
Relation:
flow-rate $_{Q+}$ (amount(pressure(src)) - amount(pressure(dst))
Influences:
I+ (amount-of(dst), amount(flow-rate)
I- (amount-of(src), amount(flow-rate)

The influences state that the amount of the flow rate causes the amount of water in dst to increase, and causes the amount of water in src to decrease.

This has been a brief overview of an extremely general theory, but it is enough to illustrate SCAVENGER's ability to reason with QP simulations. There are a number of

reasoning problems that can be solved using QP theory. One of these is determining the activity of processes in a situation. (Forbus 1984) describes this problem as:

> *A process instance has a status of* Active *or* Inactive *according to whether or not the particular process it represents is acting between its individuals. By determining whether or not the preconditions and quantity conditions are true, a status can be assigned to each process instance for a situation (page 111).*

Forbus describes how this problem can be solved by reasoning about preconditions of target processes. SCAVENGER offers another approach to this problem: that of forming and testing analogies, and using learning to improve performance across trials. The remainder of this chapter illustrates it on the example of figure 42.

In the SCAVENGER solution, objects are represented as instances of CLOS objects. A quantity is an object having slots for amount and derivative. A tank has the definition:

```
tank:
    height
    level
    open-p
    inlet
    outlet
```

The height slot indicates the height of the tank, and the level slot the water level. These slots are bound to instances of quantity. Open-p is set to t or nil, telling whether the tank is open or not. Inlet and outlet are pointers to fluid-paths that allow water to flow in and out of the tank.

A fluid-path has the slots:

```
fluid-path
    source
    destination
    flow-rate
```

source and destination are pointers to instances of tank, and flow-rate is set to 0 if there is no flow, 1 if water flows from source to destination, and -1 if it flows in the other direction. In addition to these objects, I have defined methods for accessing their components, and for connecting them into a structure such as described in the figure.

The flow process is a method that takes two instances of tank and an instance of fluid-path. It checks the preconditions and quantity conditions specified in the flow specification; if they are met, it applies the associated influences to its arguments. These set the flow-rate to 1, the derivative of the level of the source to -1 and the derivative of the destination level to +1. In addition, I have defined a no-flow process, which does nothing. To make the example more interesting, I have also defined evaporation and no-evaporation processes.

In addition, I have used the QP-descriptions to define the method descriptions and construct index patterns. For example, the definition of the flow method is:

```
signature: tank tank fluid-path -> nil
result: result
definition: ((I+ level arg-0) (I- level arg-1))
```

Note the use of influences in the method definition.

Assume that we are given the derivatives of the water levels in the two containers of figure 42 (WC is going down, and WD is going up), and want to determine the processes that could account for that observation. We can give SCAVENGER the following description of the system in question:

```
(setq t1 (make-tank 10 8)) -> ?
(setq t2 (make-tank 12 6)) -> ?
(setq p1 (make-fluid-path)) -> ?

(connect t1 p1) -> ?
(connect p1 t2) -> ?

(?process1 t1 t2 p1) -> ?
(?process2 t1) -> ?
(?process3 t2) -> ?

(get-derivative (get-level t1)) -> -1
(get-derivative (get-level t2)) -> 1
```

In this problem, all the method names match those in the source base, and will be recognized except ?process1, ?process2 and ?process3. Running the algorithm finds an explanation of the observed changes in terms of active and inactive processes:

```
(SETQ T1 (MAKE-TANK 10 8)) -> #<TANK #x3E1B09>
```

```
(SETQ T2 (MAKE-TANK 12 6)) ->  #<TANK #x3E1B69>
(SETQ P1 (MAKE-FLUID-PATH)) -> #<FLUID-PATH #x3E1B89>
(CONNECT T1 P1) -> NIL
(CONNECT P1 T2) -> NIL
(FLOW T1 T2 P1) -> NIL
(NO-EVAPORATION T1) -> NIL
(NO-EVAPORATION T2) -> NIL
(GET-DERIVATIVE (GET-LEVEL T1)) ->  -1
(GET-DERIVATIVE (GET-LEVEL T2)) -> 1
```

In this example, SCAVENGER has determined that the flow process is active, and the evaporation process is inactive. This approach could be expanded to include more objects and processes.

SCAVENGER and reasoning about QP descriptions

This example suggests a fruitful interaction between SCAVENGER and QP theory. The ease of implementing the problem attests to SCAVENGER's generality. The ease of adapting QP theory's description language as a language for describing SCAVENGER sources further supports the conjecture that "natural" description languages can be an effective basis for SCAVENGER indices.

In turn, SCAVENGER has the promise of making contributions to QP theory. Although QP theory's ontology is simple and elegant, the reasoning processes involved in solving problems is quite complex. This conflicts with the theory's goal of representing

> *the commonsense knowledge people have about the physical world (Forbus 1984) (page 87).*

For example, although I could easily imagine children having intuitions about such concepts as qualitative derivatives (water levels go up and down) or relations (as the water level goes down, there is less water in the glass), I find it hard to imagine how they can reason about preconditions in the fashion specified in the theory. Yet children and other relative novices do understand the qualitative behaviors of physical systems. How can they do this without performing complex inferences?

SCAVENGER performs a very primitive form of inference. At first, it simply tries random alternatives until something works. As it gains experience, it remembers

things that have worked and comes to prefer these explanations if they can be made to fit the target. As it learns, it acquires a set of explanation templates that it uses with a superficial understanding of their meaning. Like a novice, it will even try to apply its favorite explanations in areas where an expert would realize they clearly do not fit. Also, like a novice, SCAVENGER becomes very good at solving familiar problems, but often lacks an expert's ability to generalize its knowledge to novel situations. Coupling SCAVENGER with QP theory offers a tool for studying the role of "reasoning impoverished" but "learning intensive" strategies in problem solving.

7
Summary of Results and Future Research

The pattern which connects is a metapattern. It is a pattern of patterns.

Gregory Bateson

Mind and Nature

This dissertation was motivated by a belief that metaphor and analogy are essential processes underlying intelligent behavior. Important aspects of these processes include the ability to reason about similarity, the transfer of semantic knowledge between analogical sources and targets, and the role of systematic patterns of relationships in selecting and developing analogies. The interaction theory of metaphor provided a framework for my computational investigation of these phenomena. This dissertation explored the ramifications of the interaction theory for selecting an analogical source that would support viable, productive analogies.

This conclusion summarizes the results of this research. It begins by summarizing the empirical findings, and re-examines the dissertation's original conjectures. Finally, it suggests areas of continued research.

7.1 Empirical evaluation of SCAVENGER

I evaluated SCAVENGER's performance across three domains. The first two, the interpretation of tutorial examples of LISP function behavior and the diagnosis of bugs in children's subtraction skills, were the basis of the quantitative evaluations of the algorithm's behavior. The third domain, reasoning about physical simulations, was included as a demonstration of the algorithm's flexibility. SCAVENGER exhibited a number of interesting quantitative properties:

1. It learns well. On the test of chapter 4, the algorithm not only demonstrated a 70% speedup on training problems, but also transferred enough of its learned knowledge to a distinct test set to effect a 50%

speedup. On the problem of diagnosing bugs in children's subtraction, the algorithm demonstrated a two order of magnitude speedup on both test and training sets.

2. It does not overspecialize its index hierarchy. By using a variation of the information-theoretic evaluation function from ID3 to determine when to specialize nodes, SCAVENGER only adds a new node if this will give it some benefit. This prevents the algorithm from acquiring overly specialized indices.

3. It scales well. Although the final word on SCAVENGER's scaling ability will require much larger source bases, my results suggest that it scales reasonably well. In particular, its execution times seem to be relatively insensitive to the size of the source base, so long as it is solving a problem that is similar to one it has already seen. However, if it encounters a novel problem, i.e. a problem that has no syntactic similarity to those it has seen before, it can perform badly.

4. It is insensitive to the language used to represent index nodes. This is suggested by the fact that it achieved good learning effects with a very simple node description language. During its development, I experimented with different description languages; so long as the language seemed reasonable and gave some support to the formation of general categories of methods, the algorithm performed well. This contrasts with the reliance most case-based retrieval mechanisms place on strongly biased retrieval vocabularies.

5. It compares favorably with more traditional retrieval methods. In comparing it to traditional, clustering based retrieval methods, SCAVENGER out-performed the source centered approach on the problem of interpreting tutorial examples of LISP function behavior.

7.2 SCAVENGER and the interaction theory

Chapter 1 presented three conjectures about source retrieval for analogical reasoning. These conjectures are reflected in the major aspects of SCAVENGER's

architecture. The insights the SCAVENGER experiments afforded into these conjectures are:

7.2.1 Assumption-based retrieval.

Assumption-based retrieval was an effort to integrate analogical inference with the process of selecting an appropriate source. Given a problem that provides insufficient knowledge to select an appropriate source, assumption-based retrieval allows an algorithm to transfer properties of candidate sources to the target on a non-monotonic basis, evaluate those inferences heuristically, and use these results to select among candidate sources.

My results have shown that while this approach requires more processing than traditional methods, it can, when coupled with the use of an appropriate learning algorithm to maintain a hierarchical index, result in significant improvements over either exhaustive analogize-and-test approaches or traditional retrieval mechanisms. After training, SCAVENGER exhibited significant speedups on both training and test sets. The algorithm seems to scale well, so long as target problems bear some similarity to problems it has already seen, although trained versions can perform poorly on completely novel problems.

Chapter 4 demonstrated that SCAVENGER outperformed a more traditional memory organization strategy, although the under-constrained nature of the test problem handicapped the comparison algorithm. In both test domains, little information is initially given about target problems: Target transcripts only indicate the number of arguments in target methods and the types of some of their arguments. As is often the case with under-constrained problems, a non-monotonic, "assume and evaluate" strategy may be the only reasonable approach; SCAVENGER has demonstrated that this can work effectively for analog source retrieval.

Although the limitations SCAVENGER's problem formulation placed on retrieval differed from the assumptions usually made by analogical or case-based reasoners, They are both realistic and necessary if we are trying to understand general mechanisms of analogical inference. Many potential applications of analogical and

case-based reasoning involve such under-constrained problems. These include work in theory formation and discovery, diagnosis and debugging in areas where available knowledge is limited or hard to elicit, and efforts to use analogy to model aspects of human learning. Assumption-based retrieval extends the basic mechanisms of analogical reasoning to include such poorly defined problems.

7.2.2 The use of systematic structure to evaluate similarity

The ability to convey entire systems of relationships in a single comparison, is an important property of analogy. Although systematicity has been explored in the context of analogical inference (Falkenhainer et al 1989; Gentner 1983), it has not been applied to the problem of selecting a relevant source.

The interpretation of LISP function behavior is an ideal test problem for examining the role of systematic structure in source retrieval. As I have formulated this problem, target problems do not contain enough information about their *individual* components to support retrieval of relevant sources. The only clues to the interpretation of target components is the pattern of their relationships to other aspects of the problem. Retrieval must exploit this structure.

SCAVENGER did this in three ways:

1. Indices represent more complex patterns than is usually the case in indexing systems. Each index described an entire source method[s].

2. Search through the hierarchy implicitly reconstructed recurring patterns of function interaction. Matches with indices deeper in the tree reflected a combination of methods in the target.

3. SCAVENGER's heuristics for ranking candidate analogies analyzed systematic properties of the different interpretations of the target that resulted from competing sources.

It is interesting to note that as SCAVENGER learned, the heuristic ranking of analogies proved less important to the algorithm's performance than the learning algorithm. This idea that "smart learning" is more important than "smart search"

is an interesting challenge to much of the conventional wisdom in AI problem solving.

7.2.3 Empirical memory management

Most retrieval systems for analogical and case-based reasoning build indices by selecting properties from a pre-defined retrieval vocabulary. As defined in this work, the problem of interpreting observations prevented the definition of such vocabularies. A central conjecture of this dissertation asserts that information that proved useful in solving target problems is an effective source of index patterns.

Although the interaction theory of metaphor is often cited in the literature on analogical reasoning, one of its central assertions, that metaphors transfer knowledge from the target to the source, as well as from the source to the target, is often ignored in models of analogical reasoning. In using target problems to restrict the set of properties that can be used in constructing indices, SCAVENGER implements a form of knowledge transfer back to the source. Specifically, it transfers knowledge of the relevance of different source properties from the target context back to the source.

The success of this approach depends upon the likelihood that such patterns of relevance will repeat across the space of target problems. Although there is no logical reason that such recurring patterns should exist, there are a number of domains where this assumption may be justified. The problem of interpreting tutorial examples, is such a domain. Diagnosis is another area where this assumption should prove valid. Many symptoms frequently co-occur with other symptoms; SCAVENGER can find and exploit such patterns. The failure of certain components may induce failures in other components of a system; SCAVENGER can discover these interactions.

My evaluations supported this conjecture. It is also interesting to note that while SCAVENGER worked well on problems that were similar to those it had seen, it could perform worse than exhaustive search on completely novel problems. Statistics on the number of problem solutions that resulted from nodes deeper in the index

hierarchy further supported the existence of a sufficient number of recurring patterns of target problems to counteract the added overhead of failure.

In adapting ID3's information-theoretic evaluation strategies to the problem of specializing SCAVENGER's indices, I changed several of the contextual assumptions made in the original ID3 research. In basing its analysis on the solution of a single target problem, SCAVENGER applied its analysis to a critically limited set of data, and also risked the possibility that contextual constraints found in one target might not generalize to other target problems. Empirical results suggest that information-theoretic evaluation strategies are robust enough to overcome both of these limitations.

7.2.4 Conclusion

The SCAVENGER experiments have shown that for appropriate problems, and given an appropriate representation language, the primary conjectures of this dissertation hold. Work in applying SCAVENGER to different domains suggest that the algorithm's underlying assumptions will generalize to a variety of applications. However, the algorithm and its applications deserve and require further study; the next section discusses areas for future research.

7.3 Future Directions

An obvious direction for future work must involve applying SCAVENGER to a variety of new problem domains. The evaluation of chapters 4 and 5 covered two domains in depth: the interpretation of tutorial examples, and the diagnosis of procedural failures. Chapter 6 provided a simple proof of concept showing how SCAVENGER could be applied to simulation based reasoning. Further application of SCAVENGER to realistic problems is an obvious extension of this research. Such problem areas include:

7.3.1 Case-based reasoning

Case-based reasoning is the area in which SCAVENGER can have the most direct impact. Most case-based reasoners rely on such simplifying assumptions as the existence of a fixed, unchanging retrieval vocabulary. They assume that target

problems will include sufficient information selected from this vocabulary to select appropriate sources, and that sources can be adequately discriminated under terms taken from this vocabulary. However, this assumption leaves most case-based reasoners excessively reliant on biases in the selection and representation of retrieval vocabularies. SCAVENGER has shown that it is possible to construct retrieval mechanisms that work well in the absence of such biases.

Another area in which SCAVENGER shows promise for contributing to CBR is the problem of store-compute trade-offs. Managing these trade-offs is an important problem for case-based reasoning. SCAVENGER manages these trade-offs and gains flexibility by performing a type of reconstructive transfer. SCAVENGER does not store entire instances of successful problem solutions; instead, it stores generalized descriptions of key patterns of those solutions in its index structure. It then reconstructs source solutions for each target using these general patterns as a guide. This strategy reduces the storage needed for maintaining large collections of source solutions, and speeds up retrieval through a hierarchical index. The price paid for this gain is a greater time spent in retrieving and reconstructing source solutions.

7.3.2 SCAVENGER and the application of analogies to scientific discovery

Scientific discovery is another promising application area. Hesse (1966) has shown the importance of analogical reasoning in human science. Scientific and commonsense theories of the world may be seen as conforming to a relatively small number of abstract patterns of theoretical explanation. For example, physical theories often take the form of equilibrium laws, mechanical explanations, field effects or flow theories. In social interactions, a general pattern of "exchanging things of equal value" governs relationships as diverse as buying and selling, employment, contract law and politics. This is not surprising if we assume that new theories are formed through analogies with existing knowledge. Further work may use SCAVENGER to test this conjecture.

7.3.3 Cognitive models of analogy

Perhaps the most exciting area of continued research would be to evaluate SCAVENGER as a model of analogical reasoning in humans. Because the interaction

view of metaphor is a cognitive/philosophical theory, SCAVENGER already has roots in this area. Additional cognitive science influences on its design include Gick and Holyoak's work on abstraction and analogy. In addition, much of its performance during evaluation was suggestive of human behavior. For example, SCAVENGER's tendency to "lock onto" favorite solutions and possibly mis-apply them, particularly during the early stages of learning, suggests the problem solving behavior of novices. The fact that SCAVENGER's performance can get worse before it gets better has a similar human quality. These observations suggest that it is worthy of consideration as a serious model of analogy formation in humans. I would enjoy the opportunity to test this hypothesis, first by searching the literature on human analogy for patterns of behavior reflected in the algorithm, and later by designing and performing experiments with human subjects.

7.4 Conclusion

One of the primary goals of this research was to explore the mechanisms by which metaphor supported the construction of interpretations of empirical data. In doing this research, I was guided by the interaction theory of metaphor, and much of this work should be regarded as an exploration of the empirical application of that theory.

The choice and representation of test problem reflects this concern. Most work in source retrieval for case-based and analogical reasoning, assumes a fixed retrieval vocabulary in which the values of certain properties are known for all objects. However, the interaction theory of metaphor questions these assumptions, focusing on the view that similarity arises out of complex systems of relations, and interactions between individual objects. The analogical interpretation problem addressed in the SCAVENGER experiments assume that the names and interpretation of target terms are unknown, and must be inferred through an such a process of interpretation. This contrasts with more traditional reasoning systems that begin with a collection of names for things and rules that reason with those names.

Analysis of SCAVENGER's performance suggest insights into the relationship between perception, analogy and interpretation. It recognizes that perception is not a simple, deterministic, data-driven process, but a complex *construction* of an

interpretation, where perceptions are as shaped by an agent's prior experience, and current goals as by so-called "objective" reality. Its interpretations are constrained empirically (the requirement that it run the transcript correctly) but also depend upon such non-empirical criteria as a preference for interpretations that closely match those that have succeeded in the past. SCAVENGER's interpretation of target transcripts can vary according to its past experience. The SCAVENGER experiments show how analogies can effect this construction of interpretation and provide an explicit computational model of that process.

An interesting aspect of SCAVENGER is the primitive nature of its inference strategies. It does not perform deductive reasoning, or traditional statistical analysis. It begins with a trial and error process of discovery and remembers generalized descriptions of things that work. Recent years have seen a rise in work on inference strategies that, while lacking such properties as soundness and completeness, seem better suited to managing an agent's behavior in complex, changing, open worlds. Research in artificial life, neural networks, and analogical reasoning reflect this change of focus, and reveal the richness of such primitive inference strategies. There is a greater recognition that deductive reasoning is not the basis of intelligence, but is only one of its products.

Finally, it is worth re-stating that SCAVENGER uses analogies to project viable interpretations onto otherwise under-determined sets of observations. Philosophers, psychologists and cognitive scientists have long recognized that intelligent agents actively construct interpretations, rather than merely manipulate symbolically described truths about an objective world. The interaction theory of metaphor provides a basis for a computational theory of the processes underlying that construction. SCAVENGER is an instantiation of that theory.

Appendix 1
The Class and Method Library Used in the Tests of Chapter 4

The experiments of chapter 4 used a source library consisting of the classes and methods described in this appendix. The classes cover a number of different domains, including mathematics, accounting, data structures and simulation. These areas, along with the number of classes and methods in each, are:

Domain	**# classes**	**# methods**
Accounting	2	13
Complex numbers	1	9
Data structures	8	55
Date arithmetic	1	13
Investments	3	9
Location using x-y co-ordinates	1	6
Object-oriented data base	1	7
Rational numbers	1	10
Strings	1	15
Thermostat-room-heater simulation	3	15
Totals	**22**	**152**

Before listing the definitions of these classes and methods, it is useful to describe the language and conventions used in their description.

SCAVENGER's retrieval system exploits three types of knowledge about source methods: their argument signatures, their side-effects and a high level description of their semantics. The signature specifies the types of arguments and results. These types are either classes from the source library, or built-in LISP types.

The second component of a source description is a specification of the function's result and side-effects: this specifies which arguments are changed by the function

action. I have divided this into separate specifications of the result returned by the function and "hidden" side-effects. This specification helps distinguish methods, and is also needed to construct the graph of function behavior that SCAVENGER uses to rank candidate analogies.

The third component of a source description is a high level description of function semantics; this is a collection of predicates on function arguments and results.

For example, the upcase method for the string class takes a string as its argument and raises all of its characters to upper case. The display method shows the value of a string. If s is bound to an instance of string, and has the value: "hello," these functions have the behavior:

```
? (display s)
"hello"
? (display (upcase s))
"HELLO"
```

Upcase is specified as:

```
signature: string -> string
result: arg-0
side-effects: ()
description: ((modify-property arg-0))
```

This states that upcase takes a single string as an argument and returns a string as result. It returns its first argument as a result, and has the behavior of modifying a property of that argument.

Similarly, the get-substring method, returns the portion of a string specified by its starting and ending positions. It has the behavior (using the value of s from the preceding example):

```
? (display (get-substring 2 4 s))
"LL"
```

Get-substring has the specification:

signature: fixnum x fixnum x string -> string
result: result
side-effects: ()
description ((list-of-elements arg-2 result))

This states that the get-substring method takes three arguments, two fixnums (the LISP name for a short integer), and a string. It returns a different string as a result, and has no side effects. The result is a list of elements in argument 2. When a function such as get-substring, returns a new object as a result, I have adopted the convention of labeling the returned value result.

The set-amount function for the entry class of the accounting domain illustrates the representation of side-effects. Set-amount takes as arguments a new amount for an entry and an entry. It returns the value of the new amount as a result, and changes the entry as a side-effect. For example, if e is bound to an instance of entry, then set-amount would have the effect:

```
? (set-amount 100.00 e)
100.00
? (get-amount e)
100.00
```

Set-amount has the description:

signature: float x entry -> float
result: arg-0
side-effects: (arg-1)
description: ((set-property-value arg-0 arg-1))

This states that the method returns its first argument as a result, and has the side-effect of changing its second argument. It has the behavior of setting one of the property values of argument 1 to argument 0.

A method-description language

The language used to specify method descriptions is important to SCAVENGER's behavior. It should be general enough to cover sets of similar methods, supporting abstraction in the hierarchy, but it should be specific enough to make useful distinctions between categories of methods. However, optimizing such a language

expressly to improve SCAVENGER's performance would be difficult if not impossible. I have used the heuristic of describing methods at a general level, using terms that would seem natural to a computer scientist.

The language I have used to describe the sources used in the tests of chapter 4 is:

(add-to-collection object-i object-j)

Adds object-i to the collection specified in object-j.

(apply-function object-i object-j)

Object-i is a function applied to object-j.

(combine object-i . . .)

Combines all its arguments into a single structure.

(copy-of object-i)

Specifies a copy of object-i.

(diff object-i object-j)

The result is the difference between objects i and j. In addition to numeric subtraction, I use this to describe more general forms of difference, such as set difference.

(div object-i object-j)

The quotient of objects i and j.

(equality-test object-i object-j)

Returns t if object-i equals object-j, nil otherwise.

(find-on-key object-i object-j)

Retrieves elements of object-j using object-i as a key.

(is-in-collection object-i object-j)

Object-i is a member of the collection, object-j.

(list-of-elements object-i object-j)

Object-j is a list of elements from the collection, object-i.

(modify-property object-i)

Changes a property value in object-i. The new value is not passed in as an argument, but is computed according to some fixed algorithm, such as incrementing a value by 1.

(new-object object-i)

Object-i is a new object.

(ordinal-comparison object-i object-j)

The method performs an ordinal comparison of objects i and j.

(property-test object-i)

The method is a predicate testing a property of object-i.

(property-value object-i object-j)

Object-j is a property value for object-i.

(remove-from-collection object-i object-j)

Removes object-i from the collection, object-j.

(root object-i)

The result is a root of object-i.

(set-property-value object-i object-j)

Set the value of a property of object-i to object-j.

(select-argument object-i . . .)

Selects one of a group of objects, based on some criterion such as finding the maximum.

(structure-component object-i object-j)

Object-j is a component of the structure, object-i.

(sum object-i object-j)

The sum of objects i and j. In addition to numeric addition, I use this to describe more general forms of summation, such as set union and string concatenation.

(times object-i object-j)

The product of objects i and j. In addition to numeric multiplication, I use this to describe additional operations such as set intersection.

The source classes used to test SCAVENGER

The classes and methods used to test SCAVENGER, along with their descriptions, are:

1. ACCOUNTING DOMAIN

CLASSES

account

An account allows the posting of entries (debits and credits), the retrieval of entries by date and the computation of an account balance.

entry

An entry is a credit or debit to an account. It includes an amount and a date.

Both account and entry use the date class defined below.

METHODS

make-entry

signature: float x date -> entry
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of the entry class.

get-amount

signature: entry -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

Returns the amount of an entry.

set-amount
signature: float x entry -> float
result: arg-0
side-effects: (arg-1)
description: ((set-property-value arg-0 arg-1))

Sets the amount of an entry.

get-date

signature: entry -> date
result: result
side-effects: ()
description: ((property-value arg-0 result))

Gets the date an entry was posted.

set-date

signature: date x entry -> entry
result: arg-0
side-effects: (arg-1)
description: ((set-property-value arg-0 arg-1))

Sets the posting date for an entry.

make-account

signature: fixnum x date -> account
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of an account, given an account number and date the account was opened.

get-entries

signature: account -> list
result: result
side-effects: ()
description: ((list-of-elements arg-0 result))

Lists all the entries in an account.

get-balance

signature: account x -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

Gets the balance of an account.

get-date-opened

signature: account -> date
result: result
side-effects: ()
description: ((property-value arg-0 result))

Gets the date an account was opened.

get-number

signature: account -> fixnum
result: result
side-effects: ()
description: ((property-value arg-0 result))

Returns an account number.

post-entry

signature: account x float x date
result: arg-0
side-effects: ()
description: ((add-to-collection (combine arg-1 arg-2) arg-0))

Adds an entry to an account

list-entries

signature: account x date
result: result
side-effects: ()
description: ((list-of-elements arg-0 result))

Lists all entries to an account on a certain date.

purge-entries

signature: account -> account
result: arg-0
side-effects: ()
description: ((modify-property arg-0))

Purges the entries in an account.

---

2. COMPLEX NUMBER DOMAIN

CLASSES

complex-number

METHODS

make-complex

signature: number x number -> complex-number
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of complex number, given its real and imaginary values.

get-real-part

signature: complex-number -> number
result: result
side-effects: ()
description: ((structure-component arg-0 result)))

get-imaginary-part

signature: complex-number -> number
result: result
side-effects: ()
description: ((structure-component arg-0 result)))

complex-sqrt

signature: complex-number -> complex-number
result: result
side-effects: ()
description: ((root arg-0))

complex-+

signature: complex-number x complex-number -> complex-number
result: result
side-effects: ()
description: ((sum arg-0 arg-1))

complex--

signature: complex-number x complex-number -> complex-number
result: result
side-effects: ()
description: ((diff arg-0 arg-1))

complex-*

signature: complex-number x complex-number -> complex-number
result: result
side-effects: ()
description: ((times arg-0 arg-1))

complex-/

signature: complex-number x complex-number -> complex-number
result: result
side-effects: ()
description: ((div arg-0 arg-1))

complex-=

signature: complex-number x complex-number -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

---

3. DATA STRUCTURE DOMAIN

CLASSES

data-structure

An abstract class that defines the basic structure of its derived classes.

stack

bag

set

sorted-queue

queue

deque

A deque is a double ended queue, allowing inserts and removals from both ends.

alist

An alist is a lisp style association list.

METHODS

empty

signature: data-structure -> t
result: result
side-effects: ()
description: ((property-test arg-0))

Determines if a data structure is empty.

equal

signature: data-structure x data-structure -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

empty

signature: stack -> t
result: result
side-effects: ()
description: ((property-test arg-0))

equal

signature: stack x stack -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

make-stack

signature: -> stack
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of stack.

push

signature: t x stack -> stack
result: arg-1
side-effects: ()
description: ((add-to-collection arg-0 arg-1))

pop

signature: stack -> t
result: result
side-effects: (arg-0)
description: ((remove-from-collection result arg-0))

empty

signature: bag -> t
result: result
side-effects: ()
description: ((property-test arg-0))

equal

signature: bag x bag -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

make-bag

signature: -> bag
result: result
side-effects: ()
description: ((new-object result))

add

signature: t bag -> bag
result: arg-1
side-effects: ()
description: ((add-to-collection arg-0 arg-1))

Adds an element to a bag.

add-copy

signature: t bag -> bag
result: result
side-effects: ()
description: ((add-to-collection arg-0 (copy-of arg-1)))

Makes a copy of a bag, and adds an element to it.

member

signature: t bag -> t
result: result
side-effects: ()
description: ((is-in-collection arg-0 arg-1))

delete

signature: t bag -> bag
result: arg-1
side-effects: ()
description: ((remove-from-collection arg-0 arg-1))

delete-copy

signature: t bag -> bag
result: result
side-effects: ()
description: ((remove-from-collection arg-0 (copy-of arg-1)))

Makes a copy of a bag, and removes an element from the copy.

count

signature: bag -> fixnum
result: result
side-effects: ()
description: ((property-value result))

Counts the number of items in a bag.

union

signature: bag x bag -> bag
result: result
side-effects: ()
description: ((sum arg-0 arg-1))

empty

signature: set -> t
result: result
side-effects: ()
description: ((property-test arg-0))

equal

signature: set x set -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

make-set

signature: -> set
result: result
side-effects: ()
description: ((new-object result))

add

signature: t set -> set
result: arg-1
side-effects: ()
description: ((add-to-collection arg-0 arg-1))

Adds an element to a set.

add-copy

signature: t set -> set
result: result
side-effects: ()
description: ((add-to-collection arg-0 (copy-of arg-1)))

Makes a copy of a set, and adds an element to it.

member

signature: t set -> t
result: result
side-effects: ()
description: ((is-in-collection arg-0 arg-1))

delete

signature: t set -> set
result: arg-1
side-effects: ()
description: ((remove-from-collection arg-0 arg-1))

delete-copy

signature: t set -> set
result: result
side-effects: ()
description: ((remove-from-collection arg-0 (copy-of arg-1)))

Makes a copy of a set, and removes an element from the copy.

count

signature: set -> fixnum
result: result
side-effects: ()
description: ((property-value result))

Counts the number of items in a set.

union

signature: set x set -> set
result: result
side-effects: ()
description: ((sum arg-0 arg-1))

intersection

signature: set x set -> set
result: result
side-effects: ()
description: ((times arg-0 arg-1))

difference

signature: set x set -> set
result: result
side-effects: ()
description: ((diff arg-0 arg-1))

empty

signature: sorted-queue -> t
result: result
side-effects: ()
description: ((property-test arg-0))

equal

signature: sorted-queue x sorted-queue -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

make-sorted-queue

signature: -> sorted-queue
result: result
side-effects: ()
description: ((new-object result))

insert

signature: t x sorted-queue -> sorted-queue
result: arg-1
side-effects: ()
description: ((add-to-collection arg-0 arg-1))

first

signature: sorted-queue -> t
result: result
side-effects: (arg-0)
description: ((remove-from-collection result arg-0))

Removes the first element of a sorted queue and returns it.

merge

signature: sorted-queue x sorted-queue -> sorted-queue
result: result
side-effects: ()
description: ((sum arg-0 arg-1))

Merges two sorted queues, forming a third.

empty

signature: queue -> t
result: result
side-effects: ()
description: ((property-test arg-0))

equal

signature: queue x queue -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

make-queue

signature: -> queue
result: result
side-effects: ()
description: ((new-object result))

enqueue

signature: t x queue -> queue
result: arg-1
side-effects: ()
description: ((add-to-collection arg-0 arg-1))

dequeue

signature: queue -> t
result: result
side-effects: (arg-0)
description: ((remove-from-collection result arg-0))

conc

signature: queue x queue -> queue
result: result
side-effects: ()
description: ((sum arg-0 arg-1))

Concatenates two queues into a third.

empty

signature: deque -> t
result: result
side-effects: ()
description: ((property-test arg-0))

equal

signature: deque x deque -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

make-deque

signature: -> deque
result: result
side-effects: ()
description: ((new-object result))

enqueue

signature: t x deque -> deque
result: arg-0
side-effects: ()
description: ((add-to-collection arg-0 arg-1))

Inserts at the rear of a deque.

dequeue

signature: deque -> t
result: result
side-effects: (arg-0)
description: ((remove-from-collection result arg-0))

Removes from the front of a deque.

enqueue-front

signature: t x deque -> deque
result: arg-0
side-effects: ()
description: ((add-to-collection arg-0 arg-1))

Inserts at the front of a deque.

dequeue-rear

signature: deque -> t
result: result
side-effects: (arg-0)
description: ((remove-from-collection result arg-0))

Inserts at the front of a deque.

conc

signature: deque x deque -> queue
result: result
side-effects: ()
description: ((sum arg-0 arg-1))

Concatenates two deques into a third.

empty

signature: alist -> t
result: result
side-effects: ()
description: ((property-test arg-0))

equal

signature: alist x alist -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

make-alist

signature: -> alist
result: result
side-effects: ()
description: ((new-object result))

acons

signature: t t alist -> alist
result: arg-2
side-effects: ()
description: ((add-to-collection (combine arg-0 arg-1) arg-2)

Creates a (key . datum) record from its first two arguments, and adds it to an alist.

assoc

signature: t alist -> cons
result: result
side-effects: ()
description: ((find-on-key arg-0 arg-1))

Looks up a (key . datum) pair on the key.

del

signature: t alist -> cons
result: arg-1
side-effects: ()
description: ((remove-from-collection arg-0 arg-1))

Removes a (key . datum) pair, matching on the key.

---

4. DATE DOMAIN

CLASSES

date

METHODS

make-date

signature: integer x integer x integer -> date
result: result
side-effects: ()
description: ((new-object result))

Creates a new date, given a valid month, day and year. Returns nil if date not valid.

get-month

signature: date -> fixnum
result: result
side-effects: ()
description: ((structure-component arg-0))

get-day

signature: date -> fixnum
result: result
side-effects: ()
description: ((structure-component arg-0))

get-year

signature: date -> fixnum
result: result
side-effects: ()
description: ((structure-component arg-0))

leap-year-p

signature: date -> t
result: result
side-effects: ()
description: ((property-test arg-0))

Determines if a date is a leap year.

days-in-month

signature: date -> integer
result: result
side-effects: ()
description: ((property-value arg-0))

Returns the number of days in a month.

valid-date-p

signature: date -> t
result: result
side-effects: ()
description: ((property-test arg-0))

Determines if a date is valid.

date-=

signature: date x date -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

date-<=

signature: date x date -> t
result: result
side-effects: ()
description: ((ordinal-comparison arg-0 arg-1))

inc-date

signature: date -> date
result: arg-0
side-effects: ()
description: ((modify-property arg-0))

Increments a date by one day.

dec-date

signature: date -> date
result: arg-0
side-effects: ()
description: ((modify-property arg-0))

Decrements a date by one day.

add-days

signature: integer x date -> date
result: arg-1
side-effects: ()
description: ((sum arg-0 arg-1))

Adds a number of days to a date.

date-diff

signature: date x date -> integer
result: result
side-effects: ()
description: ((diff arg-0 arg-1))

Computes the number of days between two dates.

---

5. INVESTMENTS DOMAIN

CLASSES

investment

An abstract class defining the basic structure of all investments.

variable-rate-investment

An investment, such as a stock, in which the value, and hence the rate of return, can fluctuate.

fixed-rate-investment

An investment, such as a certificate of deposit, in which the value changes according to a pre-defined rate of return.

METHODS

make-variable-rate-investment

signature: string x float x date -> variable-rate-investment
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of variable-rate-investment given its name, initial value, and the date it was made. Initializes current value to the purchase price.

get-current-value

signature: variable-rate-investment -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

set-value

signature: variable-rate-investment float date -> variable-rate-investment
result: arg-0
side-effects: ()
description: ((set-property-value (combine arg-1 arg-2) arg-0))

Sets the value of a variable-rate-investment on a given date.

get-gain

signature: variable-rate-investment -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

Gets the profit (to date) on an investment.

get-percentage-gain

signature: variable-rate-investment -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

Gets the total percentage profit (to date) on an investment.

get-yearly-return

signature: variable-rate-investment -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

Gets the yearly rate of return on an investment.

make-fixed-rate-investment

signature: string x float x date x float -> fixed-rate-investment
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of fixed-rate-investment given its name, initial value, the date it was made, and the rate of return.

get-interest-rate

signature: fixed-rate-investment -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

get-value

signature: fixed-rate-investment x date -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

Gets the value on a given date of a fixed-rate-investment.

6. LOCATION DOMAIN

CLASSES

location

A location in a simple x-y co-ordinate system.

METHODS

make-location

signature: number x number -> location
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of a location given its x and y co-ordinates.

get-x

signature: location -> number
result: result
side-effects: ()
description: ((structure-component arg-0))

Gets the x co-ordinate value of a location.

get-y

signature: location -> number
result: result
side-effects: ()
description: ((structure-component arg-0))

Gets the y co-ordinate value of a location.

loc-=

signature: location x location -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

distance

signature: location x location -> number
result: result
side-effects: ()
description: ((diff arg-0 arg-1))

7. OBJECT ORIENTED DATABASE

object-db

A simple object-oriented "data base," allowing storage and retrieval of objects based on values of fields. Does not archive objects to disk, but keeps them in memory. I used this extensively in the SCAVENGER implementation.

METHODS

make-oo-db

signature: t list -> object-db
result: result
side-effects: ()
description: ((new-object result))

Creates an object oriented database given a name of the type of its entries and a list of keys.

insert
signature: t object-db -> object-db
result: arg-1
side-effects: ()
description: ((add-to-collection arg-0 arg-1))

Inserts a class instance into an object data base.

retrieve

signature: t t object-db -> t
result: result
side-effects: ()
description: ((find-on-key arg-0 arg-1 arg-2))

Retrieves the first match on a given field name and value.

retrieve-all

signature: t t object-db -> t
result: result
side-effects: ()
description: ((list-of-elements result arg-2))

Retrieves all matches on a given field name and value.

del

signature: t x object-db -> t
result: arg-1
side-effects: ()
description: ((remove-from-collection arg-0 arg-1))

Removes an object instance from a data base.

filter-db

signature: function object-db -> list
result: result
side-effects: ()
description: ((list-of-elements result arg-2))

Retrieves all elements of a data base that satisfy a given predicate.

map-db

signature: function x object-db -> list
result: result
side-effects: ()
description: ((apply-function arg-0 arg-1))

Applies a function to every element of an object data base, and returns the results.

---

8. RATIONAL NUMBER DOMAIN

CLASSES

rational

A lisp-style rational number, consisting of a numerator and a denominator.

METHODS

make-rational

signature: integer x integer -> rational
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of rational number, given its numerator and denominator.

get-numerator

signature: rational -> integer
result: result
side-effects: ()
description: ((structure-component arg-0 result)))

get-denominator

signature: rational -> integer
result: result
side-effects: ()
description: ((structure-component arg-0 result)))

rational-+

signature: rational x rational -> rational
result: result
side-effects: ()
description: ((sum arg-0 arg-1))

rational--

signature: rational x rational -> rational
result: result
side-effects: ()
description: ((diff arg-0 arg-1))

rational-*

signature: rational x rational -> rational
result: result
side-effects: ()
description: ((times arg-0 arg-1))

rational-/

signature: rational x rational -> rational
result: result
side-effects: ()
description: ((div arg-0 arg-1))

rational-=

signature: rational x rational -> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

rational-<=

signature: rational x rational -> t
result: result
side-effects: ()
description: ((ordinal-comparison arg-0 arg-1))

rational-max

signature: rational x rational -> rational
result: result
side-effects: ()
description: ((select-argument arg-0 arg-1))

---

9. STRING DOMAIN

CLASSES

string

METHODS

make-string

signature: character* -> string
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of a string class given a lisp-style string. E.g.: "abcd".

display

signature: string -> character*
result: result
side-effects: ()
description: ((property-value arg-0 result))

Displays the value of a string instance in a more conventional LISP format.

string-=

signature: string x string-> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

string-/=

signature: string x string-> t
result: result
side-effects: ()
description: ((equality-test arg-0 arg-1))

string-<=

signature: string x string -> t
result: result
side-effects: ()
description: ((ordinal-comparison arg-0 arg-1))

string->=

signature: string x string -> t
result: result
side-effects: ()
description: ((ordinal-comparison arg-0 arg-1))

string-length

signature: string -> fixnum
result: result
side-effects: ()
description: ((property-value arg-0 result))

concatenate

signature: string x string -> string
result: result
side-effects: ()
description: ((sum arg-0 arg-1))

reverse

signature: string -> string
result: arg-0
side-effects: ()
description: ((modify-property arg-0))

get-substring

signature: fixnum x fixnum x string -> string
result: result
side-effects: ()
description: ((list-of-elements arg-2 result))

Finds a substring, given the position numbers of its first and last elements.

substring-p

signature: string string -> t
result: result
side-effects: ()
description: ((ordinal-comparison arg-0 arg-1))

Determines if one string is a substring of another.

copy-string

signature: string -> string
result: result
side-effects: ()
description: ((copy-of arg-0))

upcase

signature: string -> string
result: arg-0
side-effects: ()
description: ((modify-property arg-0))

Makes a string all upper case.

downcase

signature: string -> string
result: arg-0
side-effects: ()
description: ((modify-property arg-0))

Makes a string all lower case.

null-string

signature: string -> t
result: result
side-effects: ()
description: ((property-test arg-0))

---

10. THERMOSTAT DOMAIN

A simple simulation of room-heater-thermostat systems.

CLASSES

thermostat

room

heater

METHODS

get-room

signature: heater -> room
result: result
side-effects: ()
description: ((structure-component arg-0 result))

Returns the instance of room that a heater heats.

get-room

signature: thermostat -> room
result: result
side-effects: ()
description: ((structure-component arg-0 result))

Returns the instance of room that a thermostat controls.

get-heater

signature: thermostat -> heater
result: result
side-effects: ()
description: ((structure-component arg-0 result))

Returns the instance of heater that a thermostat controls.

get-thermostat

signature: room -> thermostat
result: result
side-effects: ()
description: ((structure-component arg-0 result))

Returns the instance of thermostat that regulates a room.

get-status

signature: heater -> symbol
result: result
side-effects: ()
description: ((property-value arg-0 result))

Returns "on" or "off" indicating the status of a heater.

get-setting

signature: thermostat -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

get-temperature

signature: room -> float
result: result
side-effects: ()
description: ((property-value arg-0 result))

make-thermostat

signature: float -> thermostat
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of thermostat given its initial setting.

make-room

signature: float -> room
result: result
side-effects: ()
description: ((new-object result))

Creates an instance of room given its initial temperature.

make-heater

signature: -> heater
result: result
side-effects: ()
description: ((new-object result))

connect

signature: thermostat x heater x room -> thermostat
result: arg-0
side-effects: (arg-1 arg-2)
description: ((combine arg-0 arg-1 arg-2))

Connects a room, thermostat and heater into a simulation.

reset

signature: thermostat x float -> float
result: arg-1
side-effects: (arg-0)
description: ((set-property-value arg-0 arg-1))

Changes a thermostat setting. The thermostat then checks the room temperature, and either turns on the heater or does nothing.

reset

signature: room x float -> float
result: arg-1
side-effects: (arg-0)
description: ((set-property-value arg-0 arg-1))

Changes a thermostat setting given the room it is in. The thermostat then checks the room temperature, and either turns on the heater or does nothing.

heat

signature: heater -> float
result: result
side-effects: (arg-0)
description: ((modify-property arg-0))

If the heater is on, raises room temperature to equal thermostat setting.

change-temp

signature: room float -> float
result: arg-1
side-effects: (arg-0)
description: ((set-property-value arg-0 arg-1))

Changes a room's temperature. The thermostat then compares it to its setting, and turns on the heater if the new temperature is less.

Appendix 2
The Problem Set Used in the Tests of Chapter 4

The problem set used in evaluating SCAVENGER in chapter 4 consisted of 112 different problems from the various domains. The problems are all solvable using the source base. Since SCAVENGER does not abandon an unsolvable problem until performing an exhaustive search, I felt that including unsolvable problems would only distort my efforts to evaluate program performance.

The problems were distributed over the various domains as follows:

Domain	**# problems**
data structure	32
rational numbers	9
dates	13
accounting	9
complex numbers	9
locations	4
strings	20
thermostat simulation	8
investments	8
Total	**112**

A SCAVENGER problem is a transcript of a series of LISP executions and their results. The operator -> represents a single LISP evaluation; it takes two arguments: the evaluated form and the result. Expressions of the form: (instance class-name #) indicate an instance of a class. In order to best illustrate the nature of the target problems, I present the complete set of target problems for the data structure classes, followed by smaller samples of problems from each of the other domains. The comment at the beginning of each problem illustrates the correct analogical mapping.

A sampling of test problems

```
;;;;;;;;;;;;;; The complete set of data structure problems ;;;;;;;;;;;;;;;;;

;; target = stack
;; ?target-function-1 = push
;; ?target-function-2 = pop
;; ?target-function-3 = make-stack

(scavenger-find

 `(,(-> (setq x (?target-function-3)) (instance target 1))
   ,(-> (?target-function-1 'a x) (instance target 1))
   ,(-> (?target-function-1 'b x) (instance target 1))
   ,(-> (?target-function-2 x) b)
   ,(-> (?target-function-2 x) a)))

;; ?target-function-2 = push
;; ?target-function-3 = pop
;; ?target-function-1 = make-stack
;; ?target-function-4 = empty-c

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (?target-function-4 x) t)
   ,(-> (?target-function-2 1 x) (instance target 1))
   ,(-> (?target-function-4 x) nil)
   ,(-> (?target-function-2 2 x) (instance target 1))
   ,(-> (?target-function-3 x) 2)))

;; target = stack
;; ?target-function-2 = push
```

```
;; ?target-function-3 = equal-c
;; ?target-function-1 = make-stack

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (setq y (?target-function-1)) (instance target 2))
   ,(-> (?target-function-2 1 x) (instance target 1))
   ,(-> (?target-function-2 2 x) (instance target 1))
   ,(-> (?target-function-2 1 y) (instance target 2))
   ,(-> (?target-function-2 2 y) (instance target 2))
   ,(-> (?target-function-3 x y) t)
   ,(-> (?target-function-2 3 y) (instance target 2))
   ,(-> (?target-function-3 x y) nil)))
```

```
;; target = stack
;; ?target-function-1 = insert
;; ?target-function-2 = empty-p
;; ?target-function-3 = make

(scavenger-find

 `(,(-> (setq x (?target-function-3)) (instance target 1))
   ,(-> (?target-function-2 x) t)
   ,(-> (?target-function-1 'a x) (instance target 1))
   ,(-> (?target-function-2 x) nil)))

;; target = stack
;; ?target-function-1 = push
;; ?target-function-2 = equal
;; ?target-function-3 = make-stack

(scavenger-find

 `(,(-> (setq x (?target-function-3)) (instance target 1))
   ,(-> (setq y (?target-function-3)) (instance target 2))
   ,(-> (?target-function-1 'a x) (instance target 1))
   ,(-> (?target-function-1 'a y) (instance target 2))
   ,(-> (?target-function-2 x y) t)))

;; target = bag
;; ?target-function-2 = add-c
;; ?target-function-3 = count-c
;; ?target-function-1 = make-bag

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 t x) (instance target 1))
```

```
   ,(-> (?target-function-2 t x) (instance target 1))
   ,(-> (?target-function-2 t x) (instance target 1))
   ,(-> (?target-function-3 x) 3)))

;; target = bag
;; ?target-function-2 = add-c
;; ?target-function-3 = count-c
;; ?target-function-4 = member-c
;; ?target-function-1 = make-bag

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 nil x) (instance target 1))
   ,(-> (?target-function-2 nil x) (instance target 1))
   ,(-> (?target-function-3 x) 2)
   ,(-> (?target-function-4 nil x) t)))
```

```
;; target = bag
;; ?target-function-2 = add-c
;; ?target-function-3 = count-c
;; ?target-function-4 = delete-c
;; ?target-function-1 = make-bag

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 t x) (instance target 1))
   ,(-> (?target-function-2 t x) (instance target 1))
   ,(-> (?target-function-3 x) 2)
   ,(-> (?target-function-4 t x) (instance target 1))
   ,(-> (?target-function-3 x) 1)))

;; target = bag
;; ?target-function-1 = add
;; ?target-function-2 = count-members
;; ?target-function-3 = make-bag

(scavenger-find

 `(,(-> (setq x (?target-function-3)) (instance target 1))
   ,(-> (?target-function-1 'a x) (instance target 1))
   ,(-> (?target-function-2 x) 1)
   ,(-> (?target-function-1 'b x) (instance target 1))
   ,(-> (?target-function-2 x) 2)))

;; target = bag
;; ?target-function-1 = add-to-collection
;; ?target-function-2 = union
;; ?target-function-3 = member
;; ?target-function-4 = make-collection
```

```
(scavenger-find

 `(,(-> (setq x (?target-function-4)) (instance target 1))
   ,(-> (setq y (?target-function-4)) (instance target 2))
   ,(-> (?target-function-1 'a x) (instance target 1))
   ,(-> (?target-function-1 'b y) (instance target 2))
   ,(-> (setq z (?target-function-2 x y)) (instance target 3))
   ,(-> (?target-function-3 'a z) t)
   ,(-> (?target-function-3 'b z) t)))

;; target = bag
;; ?target-function-2 = add-c
;; ?target-function-3 = count-c
;; ?target-function-4 = union-c
;; ?target-function-1 = make-bag

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (setq y (?target-function-1)) (instance target 2))
   ,(-> (?target-function-2 t x) (instance target 1))
   ,(-> (?target-function-2 t x) (instance target 1))
   ,(-> (?target-function-2 t x) (instance target 1))
   ,(-> (?target-function-3 x) 3)
   ,(-> (?target-function-2 t y) (instance target 2))
   ,(-> (?target-function-2 t y) (instance target 2))
   ,(-> (?target-function-3 (target-function-4 x y)) 5)))

;; target = bag
;; ?target-function-2 = add-c
;; ?target-function-3 = equal-c
;; ?target-function-1 = make-set
```

```
(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (setq y (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'b y) (instance target 2))
   ,(-> (?target-function-2 'b y) (instance target 2))
   ,(-> (?target-function-2 'b y) (instance target 2))
   ,(-> (?target-function-3 y x) t)))



;; target = bag
;; ?target-function-2 = add-c
;; ?target-function-3 = equal-c
;; ?target-function-1 = make-set

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (setq y (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 'a x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'a y) (instance target 2))
   ,(-> (?target-function-2 'b y) (instance target 2))
   ,(-> (?target-function-3 y x) nil)))
```

```
;; target = set
;; ?target-function-2 = add-c
;; ?target-function-3 = count-c
;; ?target-function-4 = union-c
;; ?target-function-1 = make-set

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (setq y (?target-function-1)) (instance target 2))
   ,(-> (?target-function-2 'a x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'a y) (instance target 2))
   ,(-> (?target-function-2 'c y) (instance target 2))
   ,(-> (?target-function-3 (?target-function-4 x y)) 3)))



;; target = set
;; ?target-function-2 = add-c
;; ?target-function-3 = count-c
;; ?target-function-4 = intersection-c
;; ?target-function-1 = make-set

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (setq y (?target-function-1)) (instance target 2))
   ,(-> (?target-function-2 'a x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'a y) (instance target 2))
   ,(-> (?target-function-2 'c y) (instance target 2))
   ,(-> (?target-function-3 (?target-function-4 x y)) 1)))
```

```
;; target = set
;; ?target-function-2 = add-c
;; ?target-function-3 = count-c
;; ?target-function-1 = make-set

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 'a x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'a x) (instance target 1))
   ,(-> (?target-function-3 x) 2)))

;; target = set
;; ?target-function-2 = add-c
;; ?target-function-3 = equal-c
;; ?target-function-1 = make-set

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (setq y (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 'a x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'c x) (instance target 1))
   ,(-> (?target-function-2 'b y) (instance target 2))
   ,(-> (?target-function-2 'c y) (instance target 2))
   ,(-> (?target-function-2 'a y) (instance target 2))
   ,(-> (?target-function-3 y x) t)))

;; target = set
;; ?target-function-2 = add-c
;; ?target-function-3 = equal-c
;; ?target-function-1 = make-set
```

```
(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (setq y (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 'a x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-2 'c x) (instance target 1))
   ,(-> (?target-function-2 'a y) (instance target 2))
   ,(-> (?target-function-2 'b y) (instance target 2))
   ,(-> (?target-function-2 'b y) (instance target 2))
   ,(-> (?target-function-3 y x) nil)))

;; target = set
;; ?target-function-2 = add-c
;; ?target-function-3 = del-c
;; ?target-function-4 = member-c
;; ?target-function-1 = make-set

(scavenger-find

 `(,(-> (setq x (?target-function-1)) (instance target 1))
   ,(-> (?target-function-2 'a x) (instance target 1))
   ,(-> (?target-function-2 'b x) (instance target 1))
   ,(-> (?target-function-4 'a x) t)
   ,(-> (?target-function-3 'a x) (instance target 1))
   ,(-> (?target-function-4 'a x) nil)))

;; target = set
;; ?target-function-1 = make-set
;; ?target-function-2 = add-to-set
;; ?target-function-3 = member
;; ?target-function-4 = intersection
```

```
(scavenger-find

   `(,(-> (setq x (?target-function-1)) (instance target 1))
     ,(-> (setq y (?target-function-1)) (instance target 2))
     ,(-> (?target-function-2 'a x) (instance target 1))
     ,(-> (?target-function-2 'b x) (instance target 1))
     ,(-> (?target-function-2 'a y) (instance target 2))
     ,(-> (?target-function-2 'c y) (instance target 2))
     ,(-> (setq z (?target-function-4 x y)) (instance target 3))
     ,(-> (?target-function-3 'a z) t)
     ,(-> (?target-function-3 'b z) nil)
     ,(-> (?target-function-3 'c z) nil)))

;; target = sorted-queue
;; ?target-function-1 = make-queue
;; ?target-function-2 = insert-c
;; ?target-function-3 = first

(scavenger-find

   `(,(-> (setq x (?target-function-1)) (instance target 1))
     ,(-> (?target-function-2 1 x) (instance target 1))
     ,(-> (?target-function-2 5 x) (instance target 1))
     ,(-> (?target-function-2 3 x) (instance target 1))
     ,(-> (?target-function-3 x) 1)
     ,(-> (?target-function-3 x) 3)))

;; target = sorted-queue
;; ?target-function-1 = make-queue
;; ?target-function-2 = insert-c
;; ?target-function-3 = equal-c
```

```
(scavenger-find

  `(,(-> (setq x (?target-function-1)) (instance target 1))
    ,(-> (setq y (?target-function-1)) (instance target 2))
    ,(-> (?target-function-2 1 x) (instance target 1))
    ,(-> (?target-function-2 5 x) (instance target 1))
    ,(-> (?target-function-2 3 x) (instance target 1))
    ,(-> (?target-function-2 3 y) (instance target 2))
    ,(-> (?target-function-2 5 y) (instance target 2))
    ,(-> (?target-function-2 1 y) (instance target 2))
    ,(-> (?target-function-3 x y) t)
    ,(-> (?target-function-4 x) 1)
    ,(-> (?target-function-4 x) 3)))

;; target = sorted-queue
;; ?target-function-1 = make-queue
;; ?target-function-2 = insert-c
;; ?target-function-3 = first
;; ?target-function-4 = merge

(scavenger-find

  `(,(-> (setq x (?target-function-1)) (instance target 1))
    ,(-> (setq y (?target-function-1)) (instance target 2))
    ,(-> (?target-function-2 1 x) (instance target 1))
    ,(-> (?target-function-2 5 x) (instance target 1))
    ,(-> (?target-function-2 3 y) (instance target 2))
    ,(-> (?target-function-2 2 y) (instance target 2))
    ,(-> (setq z (?target-function-4 x y)) (instance target 3))
    ,(-> (?target-function-3 z) 1)
    ,(-> (?target-function-3 z) 2)
    ,(-> (?target-function-3 z) 3)))
```

```
;; target = queue
;; ?target-function-1 = make-queue
;; ?target-function-2 = enqueue
;; ?target-function-3 = dequeue

(scavenger-find

  `(,(-> (setq x (?target-function-1)) (instance target 1))
    ,(-> (?target-function-2 'a x) (instance target 1))
    ,(-> (?target-function-2 'b x) (instance target 1))
    ,(-> (?target-function-3 x) a)
    ,(-> (?target-function-3 x) b)))

;; target = queue
;; ?target-function-1 = make-queue
;; ?target-function-2 = enqueue
;; ?target-function-3 = dequeue
;; ?target-function-4 = empty

(scavenger-find

  `(,(-> (setq x (?target-function-1)) (instance target 1))
    ,(-> (?target-function-4 x) t)
    ,(-> (?target-function-2 1 x) (instance target 1))
    ,(-> (?target-function-2 2 x) (instance target 1))
    ,(-> (?target-function-4 x) nil)
    ,(-> (?target-function-3 x) 1)
    ,(-> (?target-function-3 x) 2)))

;; target = queue
;; ?target-function-1 = make-queue
;; ?target-function-2 = enqueue
;; ?target-function-3 = dequeue
;; ?target-function-4 = concatenate
```

```
(scavenger-find

  `(,(-> (setq x (?target-function-1)) (instance target 1))
    ,(-> (?target-function-2 'a x) (instance target 1))
    ,(-> (?target-function-2 'b x) (instance target 1))
    ,(-> (setq y (?target-function-1)) (instance target 2))
    ,(-> (?target-function-2 'c x) (instance target 2))
    ,(-> (?target-function-2 'd x) (instance target 2))
    ,(-> (setq z (?target-function-4 x y)) (instance target 3))
    ,(-> (?target-function-3 z) a)
    ,(-> (?target-function-3 z) b)
    ,(-> (?target-function-3 z) c)
    ,(-> (?target-function-3 z) d)))

;; target = deque
;; ?target-function-1 = make-deque
;; ?target-function-2 = enqueue
;; ?target-function-3 = enqueue-front
;; ?target-function-4 = dequeue
;; ?target-function-5 = dequeue-rear

(scavenger-find

  `(,(-> (setq x (?target-function-1)) (instance target 1))
    ,(-> (?target-function-2 'a x) (instance target 1))
    ,(-> (?target-function-2 'b x) (instance target 1))
    ,(-> (?target-function-3 'c x) (instance target 1))
    ,(-> (?target-function-3 'd x) (instance target 1))
    ,(-> (?target-function-4 x) d)
    ,(-> (?target-function-5 x) b)))
```

```
;; target = alist
;; ?target-function-1 = make-alist
;; ?target-function-2 = acons
;; ?target-function-3 = assoc

(scavenger-find

   `(,(-> (setq x (?target-function-1)) (instance target 1))
     ,(-> (?target-function-2 1 'a x) (instance target 1))
     ,(-> (?target-function-2 2 'b x) (instance target 1))
     ,(-> (cdr (?target-function-3 2 x)) b)))

;; target = alist
;; ?target-function-1 = make-alist
;; ?target-function-2 = acons
;; ?target-function-3 = assoc

(scavenger-find

   `(,(-> (setq x (?target-function-1)) (instance target 1))
     ,(-> (?target-function-2 1 'a x) (instance target 1))
     ,(-> (?target-function-2 2 'b x) (instance target 1))
     ,(-> (?target-function-2 3 'c x) (instance target 1))
     ,(-> (car (?target-function-3 2 x)) 2)
     ,(-> (cdr (?target-function-3 2 x)) b)))

;;;;;;;;;;;;;;;;;;;A sample of rational number problems;;;;;;;;;;;;;;;;

;; target = rational
;; ?target-function-1 = make-rational
;; ?target-function-2 = get-numerator
;; ?target-function-3 = get-denominator
;; ?target-function-4 = rational-+
```

```
(scavenger-find

 `(,(-> (setq x (?target-function-1 1 3)) (instance target 1))
   ,(-> (setq y (?target-function-1 1 4)) (instance target 2))
   ,(-> (setq z (?target-function-4 x y)) (instance target 3))
   ,(-> (?target-function-2 z) 7)
   ,(-> (?target-function-3 z) 12)))

;; target = rational
;; ?target-function-1 = make-rational
;; ?target-function-2 = get-numerator
;; ?target-function-3 = get-denominator
;; ?target-function-4 = rational-/

(scavenger-find

 `(,(-> (setq x (?target-function-1 2 3)) (instance target 1))
   ,(-> (setq y (?target-function-1 1 1)) (instance target 2))
   ,(-> (?target-function-2 x) 2)
   ,(-> (?target-function-3 x) 3)
   ,(-> (setq y (?target-function-4 y x)) (instance target 3))
   ,(-> (?target-function-2 y) 3)
   ,(-> (?target-function-3 y) 2)))

;
```

```
;;;;;;;;;;;;;;;;;;A sample of date problems ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; target = date
;; ?target-function-1 = make-date
;; ?target-function-2 = get-month
;; ?target-function-3 = get-day
;; ?target-function-4 = get-year
(scavenger-find

 `(,(-> (setq x (?target-function-1 7 4 1994)) (instance target 1))
   ,(-> (?target-function-2 x) 7)
   ,(-> (?target-function-3 x) 4)
   ,(-> (?target-function-4 x) 1994)))

;; target = date
;; ?target-function-1 = make-date
;; ?target-function-2 = date-=
;; ?target-function-3 = date-increment
;; ?target-function-4 = date-decrement

(scavenger-find
 `(,(-> (setq x (?target-function-1 1 1 1994)) (instance target 1))
   ,(-> (setq y (?target-function-1 1 1 1994)) (instance target 2))
   ,(-> (?target-function-4 y) (instance target 2))
   ,(-> (?target-function-2 x (?target-function-3 y)) t)))

;; target = date
;; ?target-function-1 = make-date
;; ?target-function-2 = date-<
;; ?target-function-3 = date-=
;; ?target-function-4 = date-inc
;; ?target-function-5 = date-dec
```

```
(scavenger-find
 `(,(-> (setq x (?target-function-1 1 1 1994)) (instance target 1))
   ,(-> (setq y (?target-function-1 1 1 1994)) (instance target 2))
   ,(-> (?target-function-4 x) (instance target 1))
   ,(-> (?target-function-2 y x) t)
   ,(-> (?target-function-3 y (?target-function-5 x)) t)))

;;;;;;;;;;;;;;;;;;;A sample of account problems;;;;;;;;;;;;;;;;;;;;;;

;; target-1 = account
;; target-2 = date
;; ?target-function-1 = make-account
;; ?target-function-2 = make-date
;; ?target-function-3 = post-transaction
;; ?target-function-4 = get-balance

(scavenger-find
 `(,(-> (setq x (?target-function-1 1234 (?target-function-2 1 1 1994)))
        (instance target-1 1))
   ,(-> (?target-function-4 x) 0.0)
   ,(-> (?target-function-3 x 100.00 (?target-function-2 1 20 1994))
        (instance target-1 1))
   ,(-> (?target-function-3 x 150.00 (?target-function-2 1 30 1994))
        (instance target-1 1))
   ,(-> (?target-function-3 x -200.00 (?target-function-2 2 21 1994))
        (instance target-1 1))
   ,(-> (?target-function-4 x) 50.00)))
```

```
;; target-1 = account
;; target-2 = date
;; ?target-function-1 = make-account
;; ?target-function-2 = make-date
;; ?target-function-3 = post-transaction
;; ?target-function-4 = list-transactions
;; ?target-function-5 = find- transaction

(scavenger-find
 `(,(-> (setq x (?target-function-1 1234 (?target-function-2 1 1 1994)))
        (instance target-1 1))
   ,(-> (?target-function-3 x 10.00 (?target-function-2 1 20 1994))
        (instance target-1 1))
   ,(-> (?target-function-3 x 15.00 (?target-function-2 1 20 1994))
        (instance target-1 1))
   ,(-> (?target-function-3 x -20.00 (?target-function-2 1 20 1994))
        (instance target-1 1))
   ,(-> (length (?target-function-4 x)) 3)
   ,(-> (length (?target-function-5 x (?target-function-2 1 20 1994))) 3)))

;;;;;;;;;;;;;;;;;;;;A sample of complex number problems;;;;;;;;;;;;;;;

;; target = complex
;; ?target-function-1 = make-complex
;; ?target-function-2 = complex-sqrt
;; ?target-function-3 = get-real-part
;; ?target-function-4 = get-imaginary-part
(scavenger-find
```

```
; example
 `(,(-> (setq x (?target-function-1 -1 0)) (instance target 1))
   ,(-> (setq y (?target-function-2 x)) (instance target 2))
   ,(-> (?target-function-3 y) 0)
   ,(-> (?target-function-4 y) 1)))

;; target = complex
;; ?target-function-1 = make-complex
;; ?target-function-2 = complex-sqrt
;; ?target-function-3 = complex-*
;; ?target-function-4 = complex-=

(scavenger-find

 `(,(-> (setq x (?target-function-1 -1 0)) (instance target 1))
   ,(-> (setq y (?target-function-2 x)) (instance target 2))
   ,(-> (?target-function-4
       x
        (?target-function-3 y y)) t)))

;;;;;;;;;;;;;;;;;;;;A sample of location problems;;;;;;;;;;;;;;;;;;;;;;;;

;; target-class = location
;; ?target-method-1 = make-location
;; ?target-method-2 = distance

(scavenger-find

 `(,(-> (setq x (?target-method-1 0 0)) (instance target 1))
   ,(-> (setq y (?target-method-1 3 4)) (instance target 2))
   ,(-> (?target-method-2 x y)
       5)))

;
```

```
;;;;;;;;;;;;;;;;;;A sample of string problems;;;;;;;;;;;;;;;;;;;;;;;;

;; target-class = string
;; ?target-method-1 = make-string
;; ?target-method-2 = concatenate
;; ?target-method-3 = display
;; ?target-method-4 = null-string

(scavenger-find

 `(,(-> (setq x (?target-method-1 "")) (instance target 1))
   ,(-> (?target-method-4 x) t)
   ,(-> (setq y (?target-method-1 "hello")) (instance target 2))
   ,(-> (?target-method-4 y) nil)
   ,(-> (setq z (target-method-2 x y)) (instance target 3))
   ,(-> (?target-method-3 z) "hello")))

;; target-class = string
;; ?target-method-1 = make-string
;; ?target-method-2 = display
;; ?target-method-3 = upcase
;; ?target-method-4 = downcase

(scavenger-find

 `(,(-> (setq x (?target-method-1 "hello")) (instance target 1))
   ,(-> (?target-method-3 x) (instance target 1))
   ,(-> (?target-method-2 x) "HELLO")
   ,(-> (?target-method-4 x) (instance target 1))
   ,(-> (?target-method-2 x) "hello")))

;; target-class = string
;; ?target-method-1 = make-string
;; ?target-method-2 = substring-p
```

```
(scavenger-find

 `(,(-> (setq x (?target-method-1 "Good morning")) (instance target 1))
   ,(-> (setq y (?target-method-1 "morning")) (instance target 2))
   ,(-> (setq z (?target-method-1 "evening")) (instance target 3))
   ,(-> (?target-method-2 y x) 5)
   ,(-> (?target-method-2 z x) nil)))

;;;;;;;;;;;;;;;;;;;;A sample of thermostat problems;;;;;;;;;;;;;;;;;;;;;;;;;

;; target-class-1 = thermostat
;; target-class-2 = room
;; target-class-3 = heater
;; ?target-method-1 = make-thermostat
;; ?target-method-2 = make-room
;; ?target-method-3 = make-heater
;; ?target-method-4 = connect
;; ?target-method-6 = get-temperature
;; ?target-method-7 = heat

(scavenger-find

 `(,(-> (setq x (?target-method-1 70.0)) (instance target-class-1 1))
   ,(-> (setq r (?target-method-2 65.0)) (instance target-class-2 1))
   ,(-> (setq h (?target-method-3)) (instance target-class-3 1))
   ,(-> (?target-method-4 x h r) (instance target-class-1 1))
   ,(-> (?target-method-6 r) 65.0)
   ,(-> (?target-method-7 h) ?)
   ,(-> (?target-method-6 r) 70.0)))
```

```
;;;;;;;;;;;;;;;;;;;;A sample of Investment problems ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; target-1 = fixed-rate-investment
;; ?target-method-1 = make-fixed-rate-investment
;; ?target-method-2 = make-date
;; ?target-method-3 = get-value

(scavenger-find

 `(,(-> (setq x (?target-method-1 "foo" 100.00 (?target-method-2 1 1 1994) 0.07))
        (instance target-1 1))
   ,(-> (?target-method-3 x (?target-method-2 1 1 1994)) 100.00)
   ,(-> (?target-method-3 x (?target-method-2 1 1 1995)) 107.00)))
```

Appendix 3
A SCAVENGER Hierarchy

The following hierarchy resulted from one of the runs of chapter 4. It was produced by 5 iterations on a training set of 57 problems. The training set was re-ordered before each run. I have used both indentation and explicit labels to show the level of each node in the hierarchy. Because this is a direct copy of output produced by my program, some of the types and methods appear in LISP syntax. For example, the expression, #<INDEX-CLASS #x484381>, indicates an instance of index-class with id number 484381. An index class is a "dummy" class used by an index to match target classes and reference source classes. Similarly, #:|index-method82654| indicates a "dummy" index method used to match targets and reference sources.

```
#<INDEX #x2C0D49> is at level 0
|---------------------------------------
| 0 sources indexed.
|---------------------------------------

        #<INDEX #x484329> is at level 1
        |---------------------------------------
        | 9 sources indexed.
        |--------------
        |#:|index-method82654|
        |(#<INDEX-CLASS #x484381> #<INDEX-CLASS #x484381>) -> #<INDEX-CLASS #x484381>
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((SUM ARG-0 ARG-1))
        |---------------------------------------
```

```
        #<INDEX #x48D2F9> is at level 2
        |----------------------------------------
        | 2 sources indexed.
        |--------------
        |#:|index-method84543|
        |(INTEGER INTEGER) -> #<INDEX-CLASS #x484381>
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((NEW-OBJECT RESULT))
        |----------------------------------------

#<INDEX #x4831F1> is at level 1
|----------------------------------------
| 1 sources indexed.
|--------------
|#:|index-method82307|
|(#<INDEX-CLASS #x483249>) -> #<INDEX-CLASS #x483249>
|Result = RESULT
|Side-effects = NIL
|Description = ((COPY-OF ARG-0))
|----------------------------------------

        #<INDEX #x48D0F9> is at level 2
        |----------------------------------------
        | 2 sources indexed.
        |--------------
        |#:|index-method84434|
        |(#<INDEX-CLASS #x483249> #<INDEX-CLASS #x483249>) -> T
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((EQUALITY-TEST ARG-0 ARG-1))
        |----------------------------------------
```

```
#<INDEX #x478111> is at level 1
|----------------------------------------
| 1 sources indexed.
|--------------
|#:|index-method80968|
|(#<INDEX-CLASS #x478169>) -> STRING
|Result = RESULT
|Side-effects = NIL
|Description = ((PROPERTY-VALUE ARG-0 RESULT))
|----------------------------------------

        #<INDEX #x4790A1> is at level 2
        |----------------------------------------
        | 2 sources indexed.
        |--------------
        |#:|index-method81504|
        |(#<INDEX-CLASS #x478169> #<INDEX-CLASS #x478169>) -> #<INDEX-CLASS
        #x478169>
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((SUM ARG-0 ARG-1))
        |----------------------------------------

        #<INDEX #x4788E1> is at level 2
        |----------------------------------------
        | 3 sources indexed.
        |--------------
        |#:|index-method81119|
        |(#<INDEX-CLASS #x478169>) -> #<INDEX-CLASS #x478169>
        |Result = ARG-0
        |Side-effects = NIL
        |Description = ((MODIFY-PROPERTY ARG-0))
        |----------------------------------------
```

#<INDEX #x472689> is at level 1

|----------

| 1 sources indexed.

|----------

|#:|index-method80867|

|(#<INDEX-CLASS #x4726E1> #<INDEX-CLASS #x4726E1>) -> #<INDEX-CLASS #x4726E1>

|Result = RESULT

|Side-effects = NIL

|Description = ((SELECT-ARGUMENT ARG-0 ARG-1))

|----------

#<INDEX #x472521> is at level 1

|----------

| 1 sources indexed.

|----------

|#:|index-method80739|

|(#<INDEX-CLASS #x472579>) -> FLOAT

|Result = RESULT

|Side-effects = (ARG-0)

|Description = ((MODIFY-PROPERTY ARG-0))

|----------

#<INDEX #x46ADD9> is at level 1

|----------

| 4 sources indexed.

|----------

|#:|index-method79481|

|(NUMBER NUMBER) -> #<INDEX-CLASS #x46AE31>

|Result = RESULT

|Side-effects = NIL

|Description = ((NEW-OBJECT RESULT))

|----------

```
        #<INDEX #x479681> is at level 2
        |----------------------------------------
        | 2 sources indexed.
        |--------------
        |#:|index-method82238|
        |(#<INDEX-CLASS #x46AE31> #<INDEX-CLASS #x46AE31>) ->
        |                        #<INDEX-CLASS #x46AE31>
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((TIMES ARG-0 ARG-1))
        |----------------------------------------

#<INDEX #x46AC81> is at level 1
|---------------------------------------
| 1 sources indexed.
|--------------
|#:|index-method79401|
|(#<INDEX-CLASS #x46ACD9> #<INDEX-CLASS #x46ACD9>) -> NUMBER
|Result = RESULT
|Side-effects = NIL
|Description = ((DIFF ARG-0 ARG-1))
|---------------------------------------

#<INDEX #x46A8C1> is at level 1
|---------------------------------------
| 7 sources indexed.
|--------------
|#:|index-method79371|
|(T #<INDEX-CLASS #x46A919>) -> #<INDEX-CLASS #x46A919>
|Result = ARG-1
|Side-effects = NIL
|Description = ((ADD-TO-COLLECTION ARG-0 ARG-1))
|---------------------------------------
```

```
#<INDEX #x478B39> is at level 2
|----------------------------------------
| 2 sources indexed.
|--------------
|#:|index-method81244|
|(#<INDEX-CLASS #x46A919>) -> FIXNUM
|Result = RESULT
|Side-effects = NIL
|Description = ((PROPERTY-VALUE ARG-0 RESULT))
|----------------------------------------

#<INDEX #x477D31> is at level 2
|----------------------------------------
| 5 sources indexed.
|--------------
|#:|index-method80911|
|(#<INDEX-CLASS #x46A919>) -> T
|Result = RESULT
|Side-effects = (ARG-0)
|Description = ((REMOVE-FROM-COLLECTION RESULT ARG-0))
|----------------------------------------

        #<INDEX #x484A01> is at level 3
        |----------------------------------------
        | 5 sources indexed.
        |--------------
        |#:|index-method82918|
        |(#<INDEX-CLASS #x46A919>) -> T
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((PROPERTY-TEST ARG-0))
        |----------------------------------------
```

#<INDEX #x46B7E9> is at level 2

|----------

| 7 sources indexed.

|----------

|#:|index-method79918|

|(#<INDEX-CLASS #x46A919> #<INDEX-CLASS #x46A919>) -> #<INDEX-CLASS #x46A919>

|Result = RESULT

|Side-effects = NIL

|Description = ((SUM ARG-0 ARG-1))

|----------

#<INDEX #x46B501> is at level 2

|----------

| 3 sources indexed.

|----------

|#:|index-method79805|

|(T #<INDEX-CLASS #x46A919>) -> #<INDEX-CLASS #x46A919>

|Result = ARG-1

|Side-effects = NIL

|Description = ((REMOVE-FROM-COLLECTION ARG-0 ARG-1))

|----------

```
                #<INDEX #x485869> is at level 3
                |----------------------------------------
                | 2 sources indexed.
                |--------------
                |#:|index-method84070|
                |(#<INDEX-CLASS #x46A919>) -> FIXNUM
                |Result = RESULT
                |Side-effects = NIL
                |Description = ((PROPERTY-VALUE ARG-0 RESULT))
                |----------------------------------------

#<INDEX #x46A679> is at level 1
|---------------------------------------
| 4 sources indexed.
|--------------
|#:|index-method79263|
|(#<INDEX-CLASS #x46A6D1>) -> T
|Result = RESULT
|Side-effects = (ARG-0)
|Description = ((REMOVE-FROM-COLLECTION RESULT ARG-0))
|---------------------------------------

#<INDEX #x46A521> is at level 1
|---------------------------------------
| 1 sources indexed.
|--------------
|#:|index-method79153|
|(T #<INDEX-CLASS #x46A579>) -> CONS
|Result = RESULT
|Side-effects = NIL
|Description = ((FIND-ON-KEY ARG-0 ARG-1))
|---------------------------------------
```

```
#<INDEX #x46A029> is at level 1
|----------------------------------------
| 10 sources indexed.
|--------------
|#:|index-method79071|
|(#<INDEX-CLASS #x46A081> #<INDEX-CLASS #x46A081>) -> T
|Result = RESULT
|Side-effects = NIL
|Description = ((ORDINAL-COMPARISON ARG-0 ARG-1))
|----------------------------------------

        #<INDEX #x46F101> is at level 2
        |----------------------------------------
        | 44 sources indexed.
        |--------------
        |#:|index-method80164|
        |(#<INDEX-CLASS #x46A081> #<INDEX-CLASS #x46A081>) -> T
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((ORDINAL-COMPARISON ARG-0 ARG-1))
        |----------------------------------------
```

#<INDEX #x469ED9> is at level 1

|----------

| 1 sources indexed.

|----------

|#:|index-method79039|

|(T #<INDEX-CLASS #x469F31>) -> #<INDEX-CLASS #x469F31>

|Result = RESULT

|Side-effects = NIL

|Description = ((REMOVE-FROM-COLLECTION ARG-0 (COPY-OF ARG-1)))

|----------

#<INDEX #x469BD9> is at level 1

|----------

| 5 sources indexed.

|----------

|#:|index-method79002|

|(#<INDEX-CLASS #x469C31> #<INDEX-CLASS #x469C31>) -> #<INDEX-CLASS #x469C31>

|Result = RESULT

|Side-effects = NIL

|Description = ((DIFF ARG-0 ARG-1))

|----------

#<INDEX #x484EE1> is at level 2

|----------

| 4 sources indexed.

|----------

|#:|index-method83167|

|(NUMBER NUMBER) -> #<INDEX-CLASS #x469C31>

|Result = RESULT

|Side-effects = NIL

|Description = ((NEW-OBJECT RESULT))

|----------

```
#<INDEX #x478D69> is at level 2
|----------------------------------------
| 2 sources indexed.
|--------------
|#:|index-method81401|
|(INTEGER INTEGER) -> #<INDEX-CLASS #x469C31>
|Result = RESULT
|Side-effects = NIL
|Description = ((NEW-OBJECT RESULT))
|----------------------------------------

#<INDEX #x471F61> is at level 2
|----------------------------------------
| 2 sources indexed.
|--------------
|#:|index-method80575|
|(T #<INDEX-CLASS #x469C31>) -> #<INDEX-CLASS #x469C31>
|Result = ARG-1
|Side-effects = NIL
|Description = ((ADD-TO-COLLECTION ARG-0 ARG-1))
|----------------------------------------

        #<INDEX #x48D509> is at level 3
        |----------------------------------------
        | 2 sources indexed.
        |--------------
        |#:|index-method84628|
        |(#<INDEX-CLASS #x469C31>) -> FIXNUM
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((PROPERTY-VALUE ARG-0 RESULT))
        |----------------------------------------
```

```
#<INDEX #x469149> is at level 1
|----------------------------------------
| 3 sources indexed.
|--------------
|#:|index-method78896|
|(INTEGER INTEGER INTEGER) -> #<INDEX-CLASS #x4691A1>
|Result = RESULT
|Side-effects = NIL
|Description = ((NEW-OBJECT RESULT))
|----------------------------------------

        #<INDEX #x485241> is at level 2
        |----------------------------------------
        | 6 sources indexed.
        |--------------
        |#:|index-method83213|
        |(#<INDEX-CLASS #x4691A1> #<INDEX-CLASS #x4691A1>) -> T
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((ORDINAL-COMPARISON ARG-0 ARG-1))
        |----------------------------------------

        #<INDEX #x478249> is at level 2
        |----------------------------------------
        | 9 sources indexed.
        |--------------
        |#:|index-method81030|
        |(#<INDEX-CLASS #x4691A1>) -> FIXNUM
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((STRUCTURE-COMPONENT ARG-0))
        |----------------------------------------
```

#<INDEX #x472169> is at level 2

|----------

| 3 sources indexed.

|----------

|#:|index-method80653|

|(INTEGER #<INDEX-CLASS #x4691A1>) -> #<INDEX-CLASS #x4721C1>

|Result = RESULT

|Side-effects = NIL

|Description = ((NEW-OBJECT RESULT))

|----------

#<INDEX #x4792A1> is at level 3

|----------

| 3 sources indexed.

|----------

|#:|index-method82189|

|(#<INDEX-CLASS #x4721C1>) -> LIST

|Result = RESULT

|Side-effects = NIL

|Description = ((LIST-OF-ELEMENTS ARG-0 RESULT))

|----------

#<INDEX #x471A51> is at level 2

|----------

| 6 sources indexed.

|----------

|#:|index-method80346|

|(#<INDEX-CLASS #x4691A1> #<INDEX-CLASS #x4691A1>) -> INTEGER

|Result = RESULT

|Side-effects = NIL

|Description = ((DIFF ARG-0 ARG-1))

|----------

#<INDEX #x46ED21> is at level 2

|----------

| 3 sources indexed.

|----------

|#:|index-method80120|

|(STRING FLOAT #<INDEX-CLASS #x4691A1> FLOAT) -> #<INDEX-CLASS #x46ED79>

|Result = RESULT

|Side-effects = NIL

|Description = ((NEW-OBJECT RESULT))

|----------

#<INDEX #x46B059> is at level 2

|----------

| 6 sources indexed.

|----------

|#:|index-method79615|

|(#<INDEX-CLASS #x4691A1>) -> #<INDEX-CLASS #x4691A1>

|Result = ARG-0

|Side-effects = NIL

|Description = ((MODIFY-PROPERTY ARG-0))

|----------

#<INDEX #x483331> is at level 3

|----------

| 18 sources indexed.

|----------

|#:|index-method82396|

|(#<INDEX-CLASS #x4691A1>) -> FIXNUM

|Result = RESULT

|Side-effects = NIL

|Description = ((STRUCTURE-COMPONENT ARG-0))

|----------

```
#<INDEX #x470EB1> is at level 3
|----------------------------------------
| 12 sources indexed.
|--------------
|#:|index-method80273|
|(#<INDEX-CLASS #x4691A1> #<INDEX-CLASS #x4691A1>) -> T
|Result = RESULT
|Side-effects = NIL
|Description = ((ORDINAL-COMPARISON ARG-0 ARG-1))
|----------------------------------------

        #<INDEX #x48B471> is at level 4
        |----------------------------------------
        | 24 sources indexed.
        |--------------
        |#:|index-method84305|
        |(#<INDEX-CLASS #x4691A1> #<INDEX-CLASS #x4691A1>) -> T
        |Result = RESULT
        |Side-effects = NIL
        |Description = ((ORDINAL-COMPARISON ARG-0 ARG-1))
        |----------------------------------------

#<INDEX #x468F81> is at level 1
|----------------------------------------
| 2 sources indexed.
|--------------
|#:|index-method78849|
|(#<INDEX-CLASS #x468FD9> #<INDEX-CLASS #x468FD9>) -> INTEGER
|Result = RESULT
|Side-effects = NIL
|Description = ((DIFF ARG-0 ARG-1))
|----------------------------------------
```

#<INDEX #x468DE9> is at level 1

|----------

| 1 sources indexed.

|----------

|#:|index-method78776|

|(INTEGER #<INDEX-CLASS #x468E69>) -> #<INDEX-CLASS #x468E41>

|Result = RESULT

|Side-effects = NIL

|Description = ((NEW-OBJECT RESULT))

|----------

Appendix 4
Sample SCAVENGER Runs

This appendix presents several transcripts of SCAVENGER runs produced by the tests of chapter 4. I have annotated them with explanatory comments; these appear in *italics*.

An example of a bag's behavior

In this example, SCAVENGER interprets an example of the behavior of the class, bag. The initial call is:

```
(scavenger-find
    `(,(-> (setq x (?target-function-1)) (instance target 1))
    ,(-> (?target-function-2 t x) (instance target 1))
    ,(-> (?target-function-2 t x) (instance target 1))
    ,(-> (?target-function-3 x) 2)
    ,(-> (?target-function-4 t x) (instance target 1))
    ,(-> (?target-function-3 x) 1)))
```

The intended interpretation is[33]:

```
target = bag
?target-function-2 = add-c
?target-function-3 = count-c
?target-function-4 = delete-c
?target-function-1 = make-bag
```

SCAVENGER prints the target transcript in a more readable form:

```
Interpreting target problem:
    ? (SETQ X (?TARGET-FUNCTION-1))
   (INSTANCE TARGET 1)
    ? (?TARGET-FUNCTION-2 T X)
   (INSTANCE TARGET 1)
    ? (?TARGET-FUNCTION-2 T X)
   (INSTANCE TARGET 1)
    ? (?TARGET-FUNCTION-3 X)
   2
    ? (?TARGET-FUNCTION-4 T X)
```

---

[33] I added the "-c" suffix to many source methods in order to avoid name collisions with built in LISP functions such as delete.

```
(INSTANCE TARGET 1)
 ? (?TARGET-FUNCTION-3 X)
1
```

This run was set to stop after finding the first successful hypothesis:

```
Found 1 successful hypotheses:
Hypotheses #<HYPOTHESIS #x2F9399> was composed of the following mappings:
  Class map.
   TARGET -----> BAG-C
  Method map.
   ?TARGET-FUNCTION-2 -----> ADD-C
       arg-0 --> arg-0; arg-1 --> arg-1
   ?TARGET-FUNCTION-1 -----> MAKE-BAG-C
   ?TARGET-FUNCTION-3 -----> COUNT-C
       arg-0 --> arg-0
   ?TARGET-FUNCTION-4 -----> DELETE-C
       arg-0 --> arg-0; arg-1 --> arg-1
```

Quality of fit is the percentage of the source class's methods mapped in the problem solution. SCAVENGER uses this to select among competing hypotheses, preferring those with a better fit.

```
The quality of fit to the source class was 0.364
```

SCAVENGER transfers the source definitions to the target methods:

```
It inferred the following causal structure:
?TARGET-FUNCTION-1 takes arguments: NIL
       returns result: RESULT
       Function has description: ((NEW-OBJECT RESULT))
?TARGET-FUNCTION-2 takes arguments: (ARG-0 ARG-1)
       returns result: ARG-1
       Function has description: ((ADD-TO-COLLECTION ARG-0 ARG-1))
?TARGET-FUNCTION-4 takes arguments: (ARG-0 ARG-1)
       returns result: ARG-1
       Function has description: ((REMOVE-FROM-COLLECTION ARG-0 ARG-1))
?TARGET-FUNCTION-3 takes arguments: (ARG-0)
       returns result: RESULT
       Function has description: ((PROPERTY-VALUE ARG-0 RESULT))
```

It then evaluates the target under the source mapping:

```
Produced the evaluation:
    ? (SETQ X (MAKE-BAG-C))
    #<BAG-C #x301F69>
    ? (ADD-C T X)
    #<BAG-C #x301F69>
    ? (ADD-C T X)
    #<BAG-C #x301F69>
    ? (COUNT-C X)
```

```
   2
    ? (DELETE-C T X)
    #<BAG-C #x301F69>
    ? (COUNT-C X)
   1
Completed search
```

An example of the complex number class

This run is the canonical example of the behavior of complex numbers: computing the square root of -1. The problem statement is:

```
(scavenger-find
     `(,(-> (setq x (?target-function-1 -1 0)) (instance target 1))
      ,(-> (setq y (?target-function-2 x)) (instance target 2))
      ,(-> (?target-function-3 y) 0)
      ,(-> (?target-function-4 y) 1)))
```

The intended interpretation is:

```
target = complex
?target-function-1 = make-complex
?target-function-2 = complex-sqrt
?target-function-3 = get-real-part
?target-function-4 = get-imaginary-part
```

The transcript is:

```
Interpreting target problem:
     ? (SETQ X (?TARGET-FUNCTION-1 -1 0))
    (INSTANCE TARGET 1)
     ? (SETQ Y (?TARGET-FUNCTION-2 X))
    (INSTANCE TARGET 2)
     ? (?TARGET-FUNCTION-3 Y)
    0
     ? (?TARGET-FUNCTION-4 Y)
    1

Found 1 successful hypotheses:
Hypotheses #<HYPOTHESIS #x307321> was composed of the following mappings:
  Class map.
    TARGET -----> COMPLEX-NUMBER
  Method map.
    ?TARGET-FUNCTION-4 -----> GET-IMAGINARY-PART
       arg-0 --> arg-0
    ?TARGET-FUNCTION-2 -----> COMPLEX-SQRT
       arg-0 --> arg-0
    ?TARGET-FUNCTION-3 -----> GET-REAL-PART
       arg-0 --> arg-0
    ?TARGET-FUNCTION-1 -----> MAKE-COMPLEX
```

```
        arg-0 --> arg-0; arg-1 --> arg-1
    The quality of fit to the source class was 0.444
It inferred the following causal structure:
?TARGET-FUNCTION-1 takes arguments: (ARG-0 ARG-1)
        returns result: RESULT
        Function has description: ((NEW-OBJECT RESULT))
?TARGET-FUNCTION-2 takes arguments: (ARG-0)
        returns result: RESULT
        Function has description: ((ROOT ARG-0))
?TARGET-FUNCTION-3 takes arguments: (ARG-0)
        returns result: RESULT
        Function has description: ((STRUCTURE-COMPONENT ARG-0 RESULT))
?TARGET-FUNCTION-4 takes arguments: (ARG-0)
        returns result: RESULT
        Function has description: ((STRUCTURE-COMPONENT ARG-0 RESULT))
Produced the evaluation:
    ? (SETQ X (MAKE-COMPLEX -1 0))
    #<COMPLEX-NUMBER #x31A2F9>
    ? (SETQ Y (COMPLEX-SQRT X))
    #<COMPLEX-NUMBER #x31A321>
    ? (GET-REAL-PART Y)
   0
    ? (GET-IMAGINARY-PART Y)
   1
Completed search
```

An example from the thermostat simulation domain

This example illustrates the thermostat simulation domain, one of the more complex problems in the source base. A simulation consists of instances of heater, thermostat and room classes. These are connected by the method connect. Through these connections, changes to one can effect others. In this example, I make instances of room and thermostat, with temperatures and settings of 65 and 70 degrees respectively. The heat method runs the simulation, raising the room temperature to 70. The problem statement is:

```
(scavenger-find
 `(,(-> (setq x (?target-method-1 70.0)) (instance target-class-1 1))
   ,(-> (setq r (?target-method-2 65.0)) (instance target-class-2 1))
   ,(-> (setq h (?target-method-3)) (instance target-class-3 1))
   ,(-> (?target-method-4 x h r) (instance target-class-1 1))
   ,(-> (?target-method-5 r) 65.0)
   ,(-> (?target-method-6 h) ?)
   ,(-> (?target-method-5 r) 70.0)))
```

The intended interpretation is:

target-class-1 = thermostat
target-class-2 = room
target-class-3 = heater
?target-method-1 = make-thermostat
?target-method-2 = make-room
?target-method-3 = make-heater
?target-method-4 = connect
?target-method-5 = get-temperature
?target-method-6 = heat

The transcript of SCAVENGER's execution is:

Interpreting target problem:

```
 ? (SETQ X (?TARGET-METHOD-1 70.0))
(INSTANCE TARGET-CLASS-1 1)
 ? (SETQ R (?TARGET-METHOD-2 65.0))
(INSTANCE TARGET-CLASS-2 1)
 ? (SETQ H (?TARGET-METHOD-3))
(INSTANCE TARGET-CLASS-3 1)
 ? (?TARGET-METHOD-4 X H R)
(INSTANCE TARGET-CLASS-1 1)
 ? (?TARGET-METHOD-5 R)
65.0
 ? (?TARGET-METHOD-6 H)
?
 ? (?TARGET-METHOD-5 R)
70.0

Found 1 successful hypotheses:
Hypotheses #<HYPOTHESIS #x315091> was composed of the following mappings:
  Class map.
    TARGET-CLASS-2 -----> ROOM
    TARGET-CLASS-1 -----> THERMOSTAT
    TARGET-CLASS-3 -----> HEATER
  Method map.
    ?TARGET-METHOD-2 -----> MAKE-ROOM
       arg-0 --> arg-0
    ?TARGET-METHOD-4 -----> CONNECT
       arg-0 --> arg-0; arg-1 --> arg-1; arg-2 --> arg-2
    ?TARGET-METHOD-3 -----> MAKE-HEATER
    ?TARGET-METHOD-6 -----> HEAT
       arg-0 --> arg-0
    ?TARGET-METHOD-1 -----> MAKE-THERMOSTAT
       arg-0 --> arg-0
    ?TARGET-METHOD-5 -----> GET-TEMPERATURE
       arg-0 --> arg-0
    The quality of fit to the source class was 0.400
It inferred the following causal structure:
```

```
?TARGET-METHOD-1 takes arguments: (ARG-0)
        returns result: RESULT
        Function has description: ((NEW-OBJECT RESULT))
?TARGET-METHOD-2 takes arguments: (ARG-0)
        returns result: RESULT
        Function has description: ((NEW-OBJECT RESULT))
?TARGET-METHOD-3 takes arguments: NIL
        returns result: RESULT
        Function has description: ((NEW-OBJECT RESULT))
?TARGET-METHOD-4 takes arguments: (ARG-0 ARG-1 ARG-2)
        returns result: ARG-0
        and produces side effects on: (ARG-1 ARG-2)
        Function has description: ((COMBINE ARG-0 ARG-1 ARG-2))
?TARGET-METHOD-6 takes arguments: (ARG-0)
        returns result: RESULT
        and produces side effects on: (ARG-0)
        Function has description: ((MODIFY-PROPERTY ARG-0))
?TARGET-METHOD-5 takes arguments: (ARG-0)
        returns result: RESULT
        Function has description: ((PROPERTY-VALUE ARG-0 RESULT))

Produced the evaluation:
     ? (SETQ X (MAKE-THERMOSTAT 70.0))
    #<THERMOSTAT #x3166B9>
     ? (SETQ R (MAKE-ROOM 65.0))
    #<ROOM #x316711>
     ? (SETQ H (MAKE-HEATER))
    #<HEATER #x316759>
     ? (CONNECT X H R)
    #<THERMOSTAT #x3166B9>
     ? (GET-TEMPERATURE R)
    65.0
     ? (HEAT H)
    70.0
     ? (GET-TEMPERATURE R)
    70.0
Completed search
```

Appendix 5
Buggy Operators used in the Tests of Chapter 5

This appendix lists the source method and class definitions used in the tests of chapter 5. These tests used the buggy operators described in (Brown and VanLehn 1980).

The sole class definition specified a working memory used to manage borrows across columns and accumulate results. This had the definition:

```
(defclass working-memory ()
   ((result :initarg :result
            :initform ()
            :reader get-result
            :writer set-result)
    (borrow :initarg :borrow
            :initform 0
            :reader get-borrow
            :writer set-borrow))))
```

The operations were taken from the list of bugs in appendix 2 of (Brown and VanLehn 1980). I have tried to implement them as faithfully as possible, although certain ambiguities in their descriptions in the article made this occasionally difficult. The following lists enumerates the bugs used in the tests, along with their signatures and the descriptions SCAVENGER used to construct its index hierarchy.

1. normal-subtract: digit x digit x working-memory -> digit
   definition: (normal-op)

2. 0-N=0/after/borrow: digit-1 x digit x working-memory -> digit-0
   definition: (subtract-error borrow-error)

3. 0-N=N/after/borrow: digit-1 x digit x working-memory -> digit
   definition: (subtract-error borrow-error)

4. 1-1=0/after/borrow: digit-1 x digit-1 x working-memory -> digit-0
   definition: (borrow-error subtract-error)

5. 1-1=1/after/borrow: digit-1 x digit-1 x working-memory -> digit-1
   definition: (borrow-error subtract-error)

6. add-borrow-carry-sub: digit x digit x working-memory -> digit
   definition: (add-error)

7. add-borrow-decrement: digit x digit x working-memory -> digit
   definition: (decrement-error)

8. add-borrow-decrement-without-carry: digit x digit x working-memory -> digit
   definition: (decrement-error)

9. add-instead-of-sub: digit x digit x working-memory -> digit
   definition: (add-error)

10. add-no-carry-instead-of-sub: digit x digit x working-memory -> digit
    definition: (add-error)

11. always-borrow: digit x digit x working-memory -> digit
    definition (borrow-error)

12. borrow-accumulate-decrement: digit x digit x working-memory -> digit
    definition: (decrement-error)

    NOTE: This method, when combined with subtract-big-decrement gives always-borrow-left

13. subtract-accumulated-decrements: digit x digit x working-memory -> digit
    definition: (decrement-error)

14. blank-instead-of-borrow: digit x digit x working-memory -> blank
    definition: (borrow-error)

15. borrow-across-zero: digit-0 x digit x working-memory -> digit
    definition (borrow-error)

16. borrow-across-zero-over-blank: digit-0 x blank x working-memory -> digit-0
    definition (borrow-error)

17. borrow-across-zero-over-zero: digit-0 x digit-0 x working-memory -> digit-0
    definition (borrow-error)

18. borrow-across-zero-touched-0-n=0: digit-0 x digit x working-memory -> digit-0
    definition (borrow-error subtract-error)

19. borrow-across-zero-touched-0-n=n: digit-0 x digit x working-memory -> digit
    definition (borrow-error subtract-error)

20. borrow-across-zero-touched-zero-is-10: digit-0 x digit x working-memory -> digit
    definition (borrow-error subtract-error)

21. borrow-add-decrement-instead-of-zero: digit-0 x digit x working-memory -> digit
    definition (borrow-error decrement-error)

22. borrow-add-is-ten: digit x digit x working-memory -> digit
    definition: (borrow-error)

23. borrow-diff-0-n=n&small-large=0: digit x digit x working-memory -> digit
    definition: (borrow-error subtract-error)

24. borrow-dont-decrement-top-smaller: digit x digit x working-memory -> digit
    definition: (decrement-error)

25. borrow-dont-decrement-unless-bottom-smaller: digit x digit x working-memory -> digit
    definition: (decrement-error)

26. borrow-from-bottom: digit x digit x working-memory -> digit
    definition: (decrement-error)

27. borrow-from-bottom-instead-of-zero: digit-0 x digit x working-memory -> digit
    definition: (decrement-error)

28. borrow-from-larger: digit x digit x working-memory -> digit
    definition: (decrement-error)

29. borrow-from-one-is-9: digit-1 x digit x working-memory -> digit
    definition: (decrement-error)

30. borrow-from-one-is-10: digit-1 x digit x working-memory -> digit
    definition: (decrement-error)

31. borrow-from-zero: digit-0 x digit x working-memory -> digit
    definition: (decrement-error)

32. borrow-from-zero-is-10: digit-0 x digit x working-memory -> digit
    definition: (decrement-error)

33. borrow-ignore-zero-over-blank: digit-0 x blank x working-memory -> blank
    definition: (decrement-error)

34. borrow-into-one=10: digit-1 x digit x working-memory -> digit
    definition (borrow-error)

35. borrow-no-decrement: digit x digit x working-memory -> digit
    definition: (decrement-error)

36. borrow-skip-equal: digit x digit x working-memory -> digit-0
    definition: (decrement-error)

37. borrow-treat-one-as-0: digit-1 x digit x working-memory -> digit
    definition: (decrement-error borrow-error)

38. borrow-unit-diff: digit x digit x working-memory -> digit-0
    definition: (borrow-error)

39. borrow-into-0-is-9: digit-0 x digit x working-memory -> digit
    definition: (borrow-error)

40. cant-subtract: digit x digit x working-memory -> blank
    definition: (subtract-error)

41. decrement-by-two-over-two: digit x digit-2 x working-memory -> digit
    definition: (decrement-error)

42. decrement-on-borrow: digit x digit x working-memory -> digit
    definition: (decrement-error)

43. decrement-1-to-11: digit-1 x digit x working-memory -> digit
    definition: (decrement-error)

44. diff-0-n=0: digit-0 x digit x working-memory -> digit-0
    definition: (subtract-error)

45. diff-1-n=1: digit-1 x digit x working-memory -> digit-1
    definition: (subtract-error)

46. diff-n-0=0: digit x digit-0 x working-memory -> digit-0
    definition: (subtract-error)

47. diff-n-n=n: digit x digit x working-memory -> digit
    definition: (subtract-error)

48. doesnt-borrow: digit x digit x working-memory -> blank
    definition: (borrow-error)

49. dont-decrement-zero: digit-0 x digit x working-memory -> digit
    definition: (decrement-error)

50. dont-decrement-zero-over-blank: digit-0 x blank x working-memory -> digit-0
    definition: (decrement-error)

51. dont-decrement-zero-over-zero: digit-0 x digit-0 x working-memory -> digit-0
    definition: (decrement-error)

52. double-decrement-one: digit-1 x digit x working-memory -> digit
    definition: (decrement-error)

53. forget-borrow-over-blanks: digit x blank x working-memory -> digit
    definition: (decrement-error)

54. ignore-zero-over-blank: digit-0 x blank x working-memory -> blank
    definition: (decrement-error)

55. increment-over-larger: digit x digit x working-memory -> digit
    definition: (decrement-error)

56. increment-zero-over-blank: digit-0 x blank x working-memory -> digit-1
    definition: (decrement-error)

57. n-9=n-1-after-borrow: digit x digit-9 x working-memory -> digit
    definition: (borrow-error subtract-error)

58. n-n=1-after-borrow: digit x digit x working-memory -> digit-1
    definition: (subtract-error)

59. n-n=9-plus-decrement: digit x digit x working-memory -> digit-9
    definition: (subtract-error decrement-error)

60. small-large=0: digit x digit x working-memory -> digit-0
    definition: (subtract-error)

61. smaller-from-larger: digit x digit x working-memory -> digit
    definition: (transpose-error)

62. stops-borrow-at-zero: digit-0 x digit x working-memory -> digit
    definition: (borrow-error decrement-error)

63. sub-one-over-blank: digit x blank x working-memory -> digit
    definition: (borrow-error decrement-error)

64. treat-top-zero-as-nine: digit-0 x digit x working-memory -> digit
    definition: (borrow-error)

65. treat-top-zero-as-10: digit-0 x digit x working-memory -> digit
    definition: (borrow-error)

66. zero-after-borrow: digit x digit x working-memory -> digit-0
    definition: (subtract-error)

67. zero-instead-of-borrow: digit x digit x working-memory -> digit-0
    definition: (subtract-error)

References

Amarel, S. 1986. Program Synthesis as a Theory Formation Task. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (1986).

Anderson, J. R. and S. M. Kosslyn, ed. 1984. *Tutorials in Learning and Memory: Essays in Honor of Gordon Bower*. San Francisco: Freeman.

Anderson, J. R. and M. Matessa. 1991. An Incremental Baysian Algorithm for Categorization. In D. H. Fisher, M. J. Pazzani, and P. Langley (1991).

Ashley, K. D. 1988. Arguing by Analogy in Law: A Case-based Model. In D. H. Helman (1988).

Ashley, K. D. and E. L. Rissland. 1988a. Compare and Contrast, A Test of Expertise. *Case-Based Reasoning Workshop in Clearwater Beach, Fla.*, edited by J. L. Kolodner, Morgan Kaufmann.

Ashley, K. D. and E. L. Rissland. 1988b. Credit Assignment and the Problem of Competing Factors in Case-Based Reasoning. *Case-Based Reasoning Workshop in Clearwater Beach, Fla.*, edited by J. L. Kolodner, Morgan Kaufmann.

Bareiss, R. 1989. *Exemplar Based Knowledge Acquisition*. San Diego, Cal.: Academic Press.

Barletta, R. and W. Mark. 1988. Explanation-Based Indexing of Cases. *AAAI 88 in St. Paul, Minnesota*, Morgan Kaufmann.

Bateson, G. 1972. *Steps to an Ecology of Mind*. New York: Ballentine Books.

Birnbaum, L. and G. Collins. 1988. The Transfer of Experience Across Planning Domains Through the Acquisition of Abstract Strategies. *AAAI 88 in St. Paul, Minnesota*, Morgan Kaufmann.

Black, M. 1962. *Models and Metaphors*. Ithaca, NY: Cornell University Press.

Branting, L. K. 1989. Integrating Generalizations with Exemplar-Based Reasoning. *Case-Based Reasoning Workshop in Pensacola Beach, Fla.*, Morgan Kaufmann.

Brown, J. S. and R. R. Burton. 1978. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science* 2 : 155-192.

Brown, J. S. and K. VanLehn. 1980. Repair theory: a generative theory of bugs in procedural skills. *Cognitive Science* 4:379-426.

Burton, R. R. 1982. *Diagnosing bugs in a simple procedural skill.* in D. Sleeman and J. S. Brown (1982).

Carbonell, J. G. and Y. Gil. 1990. Learning by Experimentation: The Operator Refinement Method. In Y. Kodratoff and R. S. Michalski (1990).

Carbonell, J. G. 1983. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (1983).

Carbonell, J. G. 1986. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (1986).

Carbonell, J. G. and M. Veloso. 1988. Integrating Derivational Analogy into a General Problem Solving Architecture. *Case-Based Reasoning Workshop in* edited by J. L. Kolodner, Morgan Kaufmann.

Cheesman, P. 1990. On Finding the Most Probable Model. In J. Shrager and P. Langley (1990b).

Clancy, W. J. 1985. Heuristic Classification. *Artificial Intelligence* 27 (3): 289-350.

Cohen, P. R. and E. A. Feigenbaum. 1982. *The Handbook of Artificial Intelligence.* Vol. III. Los Altos, California: W. Kaufmann, Inc.

Davies, T. R. 1988. Determination, Uniformity, and Relevance: Normative Criteria for Generalization and Reasoning by Analogy. In D. H. Helman (1988).

Davies, T. R. and S. J. Russell. 1987. A Logical Approach to Reasoning by Analogy. *IJCAI 87 in Milan*, Morgan Kaufmann.

DeJong, G. and R. Mooney. 1986. Explanation-Based Learning: An Alternative View. *Machine Learning* 1 (2): 145-176.

Dershowitz, N. 1983. Programming by Analogy. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (1983).

Evans, T. G. 1968. A program for the solution of geometric analogy intelligence test questions. In M. Minsky (1968).

Everitt, B. 1981. *Cluster Analysis.* London: Heinemann.

Falkenhainer, B. C. 1990a. Explanation and Theory Formation. In J. Shrager and P. Langley (1990b).

Falkenhainer, B. C. 1990b. A Unified Approach to Explanation and Theory Formation. In J. Shrager and P. Langley (1990b).

Falkenhainer, B. C., Kenneth D. Forbus, and D. Gentner. 1989. The Structure Mapping Engine: Algorithm and Examples. *Artificial Intelligence* 41 (1): 1-64.

Falkenhainer, B. C. and R. S. Michalski. 1990. Integrating Quantitative and Qualitative Discovery in the ABACUS System. In Y. Kodratoff and R. S. Michalski (1990).

Feigenbaum, E. A. 1961. The simulation of verbal learning behavior. *Proceedings of the Western Joint Computer Conference.*

Fisher, D. H. 1987. Knowledge Acquisition Via Incremental Conceptual Clustering. *Machine Learning* 2 : 139-172.

Fisher, D. H. and P. Langley. 1985. Approaches to Conceptual Clustering. *Ninth International Joint Conference on Artificial Intelligence in Los Angeles, Cal.*, Morgan Kaufmann.

Fisher, D. H., M. J. Pazzani, and P. Langley, ed. 1991. *Concept Formation:Knowledge and Experience in Unsupervised Learning*. San Mateo, Cal.: Morgan Kaufmann.

Forbus, K. D. 1984. Qualitative Process Theory. *Artificial Intelligence* 24 :(1-3)

Gentner, D. 1983a. Flowing waters or teeming crowds: Mental models of electricity. In D. Gentner and A. L. Stevens (1983).

Gentner, D. 1983b. Structure-Mapping: A Theoretical Framework for Analogy. *Cognitive Science* 7 : 155-170.

Gentner, D. 1989. The Mechanisms of Analogical Learning. In S. Vosniadou and A. Ortony (1989).

Gentner, D. and A. L. Stevens, ed. 1983. *Mental Models*. Hillsdale, N. J.: L. Erlbaum.

Gick, M. L. and K. L. Holyoak. 1983. Schema Induction and Analogical Transfer. *Cognitive Psychology* 15 : 1-38.

Gluck, M. A. and J. E. Corter. 1985. Information, uncertainty and the utility of categories. *Seventh Annual Conference of the Cognitive Science Society in Irvine, Cal.*, L. Erlbaum.

Goodman, N. 1954. *Fact, Fiction and Forecast*. 4th ed., Cambridge, Mass.: Harvard University Press.

Greiner, R. 1988a. Abstraction-Based Analogical Inference. In D. H. Helman (1988).

Greiner, R. 1988b. Learning by Understanding Analogies. *Artificial Intelligence* 35 (1): 81-126.

Hall, R. P. 1989. Computational Approaches to Analogical Reasoning: A Comparative Analysis. *Artificial Intelligence* 39 (1): 39-120.

Hammond, K. 1989a. On Functionally Motivated Vocabularies: An Apologia. *Case-Based Reasoning Workshop*. Morgan Kaufmann,

Hammond, K., ed. 1989b. *Proceedings: Case-Based Reasoning Workshop*. San Mateo, Cal.: Morgan Kaufmann.

Helman, D. H., ed. 1988. *Analogical Reasoning: Perspectives from Artificial Intelligence, Cognitive Science and Philosophy*. Dordrecht: Kluwer Academic.

Hesse, M. 1966. *Models and Analogies in Science*. Notre Dame, Indiana: University of Notre Dame Press.

Hinton, G. I. 1990. Connectionist Learning Procedures. In Y. Kodratoff and R. S. Michalski (1990).

Holland, J. H. 1986. Escaping Brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (1986).

Holland, J. H., K. J. Holyoak, R. E. Nisbett, and P. R. Thagard. 1986. *Induction: Processes of Inference, Learning and Discovery*. Cambridge, Mass.: MIT Press.

Holyoak, K. J. 1984. Mental models in problem solving. In J. R. Anderson and S. M. Kosslyn (1984).

Holyoak, K. J. 1985. The pragmatics of analogical transfer. *The Psychology of Learning and Motivation* 19 : 59-87.

Kedar-Cabelli, S. 1988a. Analogy - From a Unified Perspective. In D. H. Helman (1988).

Kedar-Cabelli, S. 1988b. Toward a Computational Model of Purpose Directed Analogy. In A. E. Prieditis (1988).

Kodratoff, Y. and R. S. Michalski, ed. 1990. *Machine Learning: An Artificial Intelligence Approach*. Vol. III. Man Mateo, CA: Morgan Kaufmann.

Kolodner, J. L., ed. 1988. *Proceedings: Case-Based Reasoning Workshop*. San Mateo, Cal.: Morgan Kaufmann.

Kolodner, J. L. 1993. *Case-Based Reasoning*. San Mateo, Cal.: Morgan Kaufmann.

Kolodner, J. L. 1983. Maintaining organization in a dynamic long term memory. *Cognitive Science* 7 : 243-280.

Kolodner, J. L. 1988a. Extending Problem Solver Capabilities Through Case-Based Inference. *Case-Based Reasoning Workshop in* edited by J. L. Kolodner, Morgan Kaufmann.

Kolodner, J. L. 1988b. Retrieving Events from a Case Memory: A Parallel Implementation. *Case Based Reasoning Workshop in* edited by J. L. Kolodner, Morgan Kaufmann.

Kolodner, J. L. 1989. Judging Which is the "Best" Case for a Case-Based Reasoner. *Case-Based Reasoning Workshop in Pensacola Beach, Fla.*, Morgan Kaufmann.

Kolodner, J. L., R. L. Simpson Jr., and K. Sycara-Cyranski. 1985. A Process Model of Case Based Reasoning. *IJCAI 85 in Los Angeles, Cal.*, edited by A. Joshi, Morgan Kaufmann.

Lakoff, G. and M. Johnson. 1980. *Metaphors We Live By*. Chicago: University of Chicago Press.

Lakoff, G. and M. Turner. 1989. *More Than Cool Reason: A Field Guide to Poetic Metaphor*. Chicago: University of Chicago Press.

Leake, D. B. 1991. An Indexing Vocabulary for Case-Based Explanation. *AAAI-91* MIT Press.

Lebowitz, M. 1980. Generalization and Memory in an Integrated Understanding System. Ph.d Thesis, Yale University.

Lebowitz, M. 1986. Concept Learning in a Rich Domain: Generalization Based Memory. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (1986).

Lebowitz, M. 1990. The Utility of Similarity-based Learning in a World Needing Explanation. In Y. Kodratoff and R. S. Michalski (1990).

Luger, G. F. 1994. Cognitive Science: The Science of Intelligent Systems. Cambridge, Mass: Academic Press.

Luger, G. F. and W. A. Stubblefield. 1993. *Artificial Intelligence: Structures and strategies for complex problem solving*. Redwood City, California: Benjamin Cummings.

Michalski, R. S. and R. E. Stepp. 1983. Learning from Observation: Conceptual Clustering. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (1983).

Michalski, R. S., J. G. Carbonell, and T. M. Mitchell, ed. 1983. *Machine Learning: An Artificial Intelligence Approach*. Vol. I. Palo Alto, Cal.: Tioga.

Michalski, R. S., J. G. Carbonell, and T. M. Mitchell, ed. 1986. *Machine Learning: An Artificial Intelligence Approach*. Vol. II. Los Altos, Cal.: Morgan Kaufmann.

Minsky, M. 1968. *Semantic Information Processing*. Cambridge, Mass: MIT Press.

Minton, S. 1988. *Learning Search Control Knowledge.* Dordrecht: Kluwer Academic Publishers.

Mitchell, M. 1993. *Analogy-making as Perception*. Cambridge, Mass.: MIT Press.

Mitchell, T. M., R. M. Keller, and S. T. Kedar-Cabelli. 1986. Explanation-Based Generalization: A Unifying View. *Machine Learning* 1 (1): 47-80.

Niiniluoto, I. 1988. Analogy and Similarity in Scientific Reasoning. In D. H. Helman (1988).

Novik, L. 1988. Analogical transfer: Processes and Individual Differences. In D. H. Helman (1988).

O'Rorke, P., S. Morris, and D. Schulenburg. 1990. Theory Formation by Abduction: A Case Study Based on the Chemical Revolution. In J. Shrager and P. Langley (1990b).

Prieditis, A. E., ed. 1988. *Analogica*. Los Altos, California: Morgan Kaufmann.

Quinlan, J. R. 1986. Induction of Decision Trees. *Machine Learning* 1 (1): 81-106.

Rajemony, S. A. 1990. A Computational Approach to Theory Revision. In J. Shrager and P. Langley (1990b).

Ross, B. 1989. Remindings in Learning and Instruction. In S. Vosniadou and A. Ortony (1989).

Rothbart, D. 1988. Analogical Information Processing Within Scientific Metaphors. In D. H. Helman (1988).

Rumelhart, D. E., J. L. McClelland, and The PDP Research Group. 1986. *Parallel Distributed Processing.* Cambridge, Mass.: MIT Press.

Russell, S. J. 1988. Analogy by Similarity. In D. H. Helman (1988).

Russell, S. J. 1989. *The use of Knowledge in Analogy and Induction.* San Mateo, California: Morgan Kaufmann.

Schank, R. 1982. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge: Cambridge University Press.

Seifert, C. M. 1988. Goals in Reminding. *Case Based Reasoning Workshop.* edited by J. L. Kolodner, Morgan Kaufmann.

Shannon, C. 1948. A mathematical theory of communication. *Bell System Technical Journal* :

Shrager, J. 1987. Theory Change via View Application in Instructionless Learning. *Machine Learning* 2 (3): 247-276.

Shrager, J. and P. Langley. 1990. Computational Approaches to Scientific Discovery. In J. Shrager and P. Langley (1990).

Shrager, J. and P. Langley. 1990b. *Computational Models of Scientific Discovery and Theory Formation*. San Mateo, California: Morgan Kaufmann.

Sleeman, D. and Brown, J. S. 1982. *Intelligent Tutoring Systems.* New York: Academic Press.

Sycara, K. 1988. Using Case-Based Reasoning for Plan Adaptation and Repair. *Case Based Reasoning Workshop* edited by J. L. Kolodner, Morgan Kaufmann.

Tavenec, P., ed. 1970. *Problems of the Logic of Scientific Knowledge*. Dordrecht: D. Reidel.

Thagard, P. 1988. *Computational Philosophy of Science*. Cambridge, Mass.: MIT Press.

Thompson, K. and P. Langley. 1991. Structured Concept Formation. In D. H. Fisher, M. J. Pazzani, and P. Langley (1991).

Turner, M. 1988. Categories and Analogies. In D. H. Helman. (1988).

Uemov, A. I. 1970. The basic forms and rules of inference by analogy. In P. Tavenec (1970).

VanLehn, K. 1990. *Mind Bugs: The origins of procedural misconceptions*. Cambridge, MA: MIT Press.

Vosniadou, S. and A. Ortony, ed. 1989. *Similarity and Analogical Reasoning*. Cambridge: Cambridge University Press.

Watanabe, S. 1969. *Knowing and Guessing: A Formal and Quantitative Study*. New York: Wiley.

Way, E. C. 1991. *Knowledge Representation and Metaphor*. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Wilkins, D. C. 1990. Knowledge Base Refinement as Improving an Incorrect and Incomplete Domain theory. In Y. Kodratoff and R. S. Michalski (1990).

Williams, R. S. 1988. Learning to Program by Examining and Modifying Cases. *Case-Based Reasoning Workshop* edited by J. L. Kolodner, Morgan Kaufmann.

Winston, P. 1980. Learning and Reasoning by Analogy. *Communications of the ACM* 23 (12): 689-702.

Winston, P. 1986. Learning by augmenting rules and accumulating censors. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (1986).

Wolstencroft, J. 1989. Restructuring, Reminding and Repair: What's Missing from Models of Analogy. *AICOM* 2 (2): 58-71.