Search in Analogical Reasoning

Joseph P. Vybihal
School of Computer Science
McGill University

and

Thomas R. Shultz
Department of Psychology
McGill University

ABSTRACT

Analogical reasoning is the process of finding and adapting old solutions to solve new problems. Unlike most analogy work, which has emphasized mapping the analogy to the target problem, we focussed on search for the analogy. Experiments with humans doing analogical reasoning uncovered a search strategy which we call Lambda Search, because of its up and down shape through long-term memory. Lambda Search begins by generalizing on the properties of the target problem and then eventually specializing on the examples of some higher level concept. These ideas were implemented in a computer program named Lambda. Simulations demonstrated that search lessened, and in some cases solved, the problems of mapping and tweaking.

ANALOGICAL REASONING

Analogical reasoning is known to have four major steps: (1) problem representation, (2) search, (3) mapping (with tweaking), and (4) procedural adaptation (also with tweaking) (Holland, Holyoak, Nisbett, & Thagard, 1986; Novick, 1988).

Problem representation is a critical part of analogical reasoning. Novick (1989) observed that novices and experts interpret problems differently depending on how well they understand the problem domain. Experts' greater ability to interpret problems and convert them into a more useful form increases problem solving effectiveness. Analogy programs with the capacity to create and modify their own problem representations are rare (but see Hammond, 1988).

Search is presumably required to find the most useful analogies to the target problem. Little progress has been made on search for analogies in systems with realistic sized long term memories. A possibly promising approach is the connectionist analogy retrieval program of Thagard and Holyoak (1988). In contrast, we attempt a symbolic level approach to analogy search compatible with the symbolic level treatments of other aspects of analogical reasoning.

A series of psychological experiments by Gentner and her colleagues (Skorstad, Falkenhainer, & Gentner, 1987) have demonstrated that people typically use surface, rather that deep, information to find analogies, thus ensuring that the analogies they find are not terribly useful. In contrast, Faries and Reiser (1988) have presented evidence that people in a problem solving context ignore surface information and instead find and use deep analogies through goal information. This suggests that Gentner's subjects did not find deep analogies because her experiments were not conducted in a problem solving context. Our work attempts to integrate both surface information and goal information into a single, seamless algorithm, and thus unify the approaches of Gentner and Reiser.

Mapping is the process that selects and copies important analogical information from the source to the target. Mapping establishes correspondences between pieces of knowledge that contain some similarity. Gentner (1986) has claimed that only structural criteria govern analogical mapping: one-to-one mappings and mappings among connected systems of predicate relations. Her Structure Mapping Engine (SME) algorithm uses a bottom-up approach with predicate calculus to create analogies (Skorstad et al., 1987).

Tweaking is the modification of elements of information that have been mapped if they cause inconsistency in the analogy. There are three aspects of tweaking: (1) avoiding items that do not need to be tweaked, (2) identifying the portions of items that need to be tweaked, and (3) actually tweaking. An interesting finding of our research is that competent analogy search can routinely handle the first two aspects of tweaking, and sometimes can even accomplish the third aspect.

Procedural adaptation uses the computed analogy to solve the target problem. Any inconsistencies discovered during use must be tweaked again, and in this manner the last draft of the analogy is created.

PSYCHOLOGICAL EXPERIMENTS

We conducted a number of experiments to determine the search process used by humans doing analogical reasoning (Vybihal, 1989). All of these experiments had the purpose of inspiring our programming rather then providing a rigorous psychological demonstration. Two of these experiments are described here: one on goal-less search and another on goal-driven search.

Goal-less Analogical Search

Goal-less analogy tasks ask the subject to produce an analogy to something in the absence of any other problem solving goal. This is basically the technique used in Gentner's psychological experiments (Skorstad et al., 1987).

Method

Three university students were asked to give an analogy for two cars crashing together and to think aloud while doing so. We drew graphs from the taped protocols by drawing a node for each concept mentioned and drawing a directed arc from that concept to any other concept that directly descended from it in the protocol. For example, if the protocol read: "If x then y", a node was drawn for x, another node for y, and a directed arc from x to y. Then the directed arcs were arranged so that those referring to a generalization, abstraction, or categorization pointed upward, whereas arcs referring to events, instances, or specializations pointed downward. These are not diagrams of the structure of memory, but rather of the search process as reported by the subjects.

Results

Examination of these graphs indicated that people can perform goal-less analogies and that they do so using a characteristic search pattern. One subject's protocol is presented as a representative example: "mm ... mm ... sounds, impact, people inside injured, two moving objects crashing together ... two snooker balls ... no, no ... two potatoes smashing together ... because crushing."

This subject listed the problem's properties and then tried to match those attributes with some other event in memory. This is a two way hierarchical search that generalized over properties and then used those properties to specialize onto a solution. The other two subjects did this as well. Unlike the other subjects, this subject proposed a solution to the problem, and then rejected it for a better solution. The interesting result is that the better solution had an extra property that matched to it as compared to the properties that matched in the first solution. This suggests that solutions are evaluated on the basis of the number of properties that match, with a better solution possessing more matches.

Goal-driven Analogical Search

Goal-driven analogy tasks ask the subject to solve a problem that may be analogous to other, already solved problems. This is essentially what Faries and Reiser (1988) have done in their analogy experiments.

Method

Three university computer science students were each given 10 problems to solve in elementary Lisp programming. Then they were given a new Lisp problem and asked to solve it given what they had learned from the previous 10 problems. The new problem was superficially similar to 1 of the 10 previous problems, and similar in goal to 1 or 2 of the other 10 previous problems. While the subjects were solving the problems, they were asked to think aloud. Each subject was also asked to give a label or a title for each problem. Each problem was couched in a brief story with a distinct human interest context; this was to provide plenty of surface information.

The 10 initial problems were: (a) accounting firm - goal: display first item in a list, (b) family historian goal: return ascending and descending version of a list, (c) political campaign - goal: return a list with the last element moved to the front of the list, (d) theologian - goal: insert an item into the second position of a list, (e) author - goal: test if a list is a palindrome, (f) librarian - goal: test if an item is in a list, if not add it to the list, (g) physicist - goal: count the elements in a list, (h) AIDS - goal: test if a list is empty; (i) math professor - goal: print out the Fibonacci number sequence given an input number, and (j) psychotic publisher - goal: remove the last element from a list.

The target problem: "An AIDS researcher detects HIV using a machine that monitors cell tissue every one-hundredth of a second. Each time the machine detects an HIV virus it increments a counter of all HIV's it has seen until it has counted to 100; then it concludes that the patient has AIDS. Write a LISP function that increments a counter every time it detects an element in a list (the element is an H for HIV detected) until 100 where it ceases to increment. The function is passed a list like (H H H H) for all HIVs detected."

Results

The protocols were diagrammed as in the previous experiment. The labels, solution, and protocol from one representative subject are presented here. The labels, in an order corresponding to that used above, are car, up&down, rotate_right, in_second, palindrome, set_add, cardinality, is_empty, fibonacci, shift_right.

Solution: (defun has_aids (l) (cond ((>= count_atoms (l) 100) t)(t f)))

Verbal Protocol: "Same as AIDS research (problem) ... rings (a) bell on (the) is-empty (problem - the AIDS problem) I think, yes, same type of problem ... no! ... counts number instead of saying found one ... ok ... (new problem counts to) 100 and then detects ... um ... look for something in a list and keep count of them to 100, then you stop ... ok ... (that means) get cardinality of list. If over 100 then ... if you want to use other functions .. use cardinality and simple test with respect to threshold of 100 ... so that's (the) count_atoms (problem) ... call it has_aids ... use the cond function and return true or false ... refer to Fibonacci for structure of COND."

This subject noticed that the problem was similar to problem (h) (the other AIDS problem) but failed to continue this line of search because the new problem's goal was not similar to the goal in the other AIDS problem. Then the subject defined the new problem's goal as using cardinality, which reminded him of the physicist problem (g) and a conditional test that, in turn, reminded him of the Fibonacci problem (i). Using these two pieces of information, he created his solution.

This subject solved the target problem using only goal information. Surface information was never even mentioned except at the beginning where the match failed because the analogous goal was not similar to the target goal. Goal information completely out-weighed surface information. The same pattern existed for the other two subjects.

Tweaking was observed in this protocol when the Fibonacci problem was used to complete this version of Count Atoms. The subject observed a difference between the target AIDS problem and the analogous physicist problem. Upon finding the Fibonacci problem that reduced this difference, the subject extracted only the critical part of the Lisp code needed to complete the Count Atoms

solution. Finally, even that part of the solution needed to be tweaked slightly. Although it is not possible to ascertain from a verbal protocol alone whether or not tweaking involved search, it is true that the program described below accomplishes these first two tweaks (changing the physicist analogy by finding the Fibonacci problem, and extracting the critical part of the Fibonacci problem) by search alone, the last tweak requiring some still unspecified process.

In all of our experiments, we have 15/15 subjects who demonstrated at least part of this up and down search pattern through LTM. In some cases, the generalization part of the search may be omitted, due to a direct match between the target and the analogy. Because of its characteristic up and down shape, we call this pattern Lambda Search.

THE LAMBDA PROGRAM

The Lambda program was designed to implement the main ideas discovered in the preceding psychological experiments. The program takes as input a LTM database of concepts and a single WM problem concept. It then tries to solve the problem using the Lambda Search process. Lambda Search first tries a direct solution, which is then specialized if it is not exact. If a direct solution fails, a generalization search occurs until a sufficiently good match is found. This match is then specialized to improve the fit of the solution. When a solution is found or too many tries have occurred, the program terminates and displays both the solution to the problem (if any) and statistics concerning the various memory accesses made. Generalization search moves upward in the LTM hierarchy by using the properties slots of concepts, whereas specialization search moves downwards through this hierarchy by employing the examples slots of concepts.

Memory Structure

In our program, the structures of LTM and WM are centered on the idea of a *Concept*. The Concept is a data structure based on frames. It supports other familiar AI structures such as scripts and production rules in a uniform way.

Concept

An important part of the meaning of a concept is its relation to other concepts in memory (Lindsay & Norman, 1977). Concepts in the Lambda program contain the four features of name, class, properties, and examples. Each value contained within a feature is called an element. Each element is not only a descriptive value but also the name of another Concept, giving a recursive quality to a set of Concepts.

With Concepts, one can represent events and ideas and access and interrelate other events and ideas. Each idea can be accessed instantly given its name. Indexing and knowledge are united since the knowledge contained within a Concept can also be used as a key to another Concept. Information can be added or modified within a given Concept structure by simply adding or deleting the properties, examples, class, or name elements. Concepts can be used to form hierarchies of interrelated knowledge containing generic definitions (an entire Concept) and particular instances of generic definitions (indexed by the generic Concept's examples elements). In this manner, memory can be organized in a general to specific hierarchy.

Concept structures inherently create three types of hierarchies accessed through class, properties, and examples. These hierarchical paths have a restriction that they must not produce circuits and the properties paths do not include examples paths and visa versa. The class hierarchy produces a path of ancestors from the current Concept to some more general Concept from which information can be inherited. The properties and examples hierarchies produce paths to other related Concepts that help describe the current Concept, which therefore allows information contained in those Concepts to be inherited for expansion of the current Concept's description.

The Concept data structure is implemented using frames with a unique frame name and three slots for class, properties, and examples. All values and facets are represented with predicate calculus

statements. A facet together with all its values, and any single value can be inserted or deleted from a Concept, thus giving Concepts a dynamic quality. Any number of facets can exist. The properties list is a more general description (category information) of the Concept while the examples list is a more specific description (specific instances, events and actions) of the Concept.

```
Concept Name = Tart(English)
        Class = pie
        pr = composed_of(crust(soft),apples), characteristics(small,baked).
        ex = example(tart(Joseph's),tart(mother's)).
end Concept.
```

In this example *pr* is the properties slot. *Composed-of* is a facet and *crust(soft)* and *apples* are two values of *composed-of*. The comma specifies that *characteristics* is another facet with two values. *Ex* is the examples slot and it contains one facet: *example*.

Within this formalism, a script (Schank, 1982) is a specialization of the Concept data structure. The script is represented by a Concept name, but prefixed by the key word *script*. The Concept class and property list are used normally, but the examples list is different, in that an event_sequence facet must be included within the slot along with any other information stored there. The order of the elements within the event_sequence facet indicates the order of their execution. Each element in the event_sequence facet is a line of code that will get executed in sequence unless a branching statement is encountered that directs control elsewhere. The elements are predicate clauses where the predicate is the name of a procedure and the clause's variables are the procedure's parameters. Scripts have an important use in the Lambda program. For example, Lambda Search is implemented using a script.

The power of the Concept data structure lies in four areas. It can add and delete values and facets which make the data structure dynamic. It uses predicate calculus to enable a complete and expressive language to represent knowledge. It possesses a built-in triple hierarchical memory structure that facilitates knowledge association and search. Lastly, the Concept structure can represent knowledge in chunks that are related to other chunks.

Working Memory

Working memory (WM) in Lambda has two parts: (1) an area that accepts new Concepts from the external world that are eventually stored in LTM, and (2) an area that accesses knowledge from LTM. WM is viewed as a list of pointers with an activation value (Anderson, 1983). As long as the activation value is greater than zero, the pointer holds onto the newly accepted Concept or the LTM Concept being pointed to. WM can be infinite in length except that it has a large overhead rehearsal factor (Lindsay & Norman, 1977). Decaying activation values conserve computational resources by limiting the size of WM.

Working memory (WM) has two control processes. One process is called *Maintenance Rehearsal* (Lindsay & Norman, 1977; Anderson, 1983). Its function is to maintain the activation of Concepts and to degenerate the activation of Concepts contained within WM according to a set of rules listed below. The other process is called *Elaborative Rehearsal* (Lindsay & Norman, 1977; Reiser, 1986); its purpose is learning. Learning is the storing of new information contained within WM into LTM. Learning is performed by *chunking*, which inserts one Concept's name into the properties or examples list of another Concept.

The following rules, similar to those used by Anderson (1983), govern Maintenance Rehearsal:

1. Activation of memory: A Concept is activated by attaching a pointer to it from WM. The Concept is given a maximum activation value of 1.0.
2. Maintenance of activation: A Concept is maintained as active in WM according to the following formula:

$N(t) = 1.0$, if Concept accessed in WM, or $N(t) = N(t-1) - K$, if Concept is not accessed

Where:

$N(t)$ is a given Concept in WM at time t

t is the latency since the Concept was last accessed within WM

$K = 0.2^{(-t/10)}$ is a rapid decay function

$0 >= N(t) <= 1$

$N(t) = 1.0$ for maximum amount of activation

$N(t) = 0.0$ Concept is deactivated

3. Spread of activation: The Concepts are processed according to their activation order (N(t)) or by the execution of an activated script.
4. Conflict resolution: If more than one Concept has the same activation value, then the Concept closer to the head of the list is executed first.

Long-term Memory

LTM is a storage area in which any Concept contained there can be instantly accessed given the Concept's name. LTM is implemented with a two-three tree (Aho, Hopcroft, & Ullman, 1983) having Concept names as indices. LTM Concepts can also be accessed by obtaining an element within a Concept's slot and using it as a search index. LTM is monotonic and therefore Concepts once stored there cannot be deleted. Modifications or deletions to a Concept's properties or examples list element are permitted, but the deletion of an entire Concept is not permitted. If an entire Concept is to be changed, a new Concept should be installed. The old Concept is then labeled as modified. This leaves trace information in LTM concerning changes in state. This is similar to humans remembering an erroneous idea they used to hold.

Search

Lambda Search is a three stage cyclic search of memory structures. Each Concept currently being processed is a potential structure that may be used in the creation of a resultant analogy in WM. As each Concept is being processed, evaluations are made as to how appropriate it is to the currently developing analogy and whether it should be included in the analogy.

The matching criteria in Lambda Search are obtained from the input problem. The input problem's properties are scanned for a goal. If a goal is found, it is used as the matching criterion. If no goal is found, then the entire list of properties (i.e., surface information) is used as matching criteria.

Lambda Search follows three steps:

1. Direct Solution. Upon statement of the problem, a direct solution is attempted. If a direct solution is found by a match using all the parameters in the matching criteria, this is called an exact match and the search terminates. If a direct solution is found where only a majority of its parameters match with the given analogy problem, this is called an inexact match. If a direct, but inexact solution is obtained, then a specialization search is performed in an attempt to complete the partial solution (step 3). Up to the point of this specialization search, direct solutions are based on memory retrieval rather than on analogy.
2. Generalization Search. If there is a failure in finding an exact or inexact solution in step 1, then there is generalization on the problem across the properties of concepts. Previous failed attempts are used as guides in the generalization. This generalization of the problem continues until an exact or inexact match is found with the given matching criteria.

3. Specialization Search. Upon reaching a satisfactory level of generalization, where an exact or inexact match has been found, a solution is developed by specializing (traveling across the examples slot of concepts) to find the best fitting solution. This specialization search builds on the possible inexact solution of stage two and attempts to obtain the greatest proportion of matches with the matching criteria's parameters.

These three search steps are cyclic, in that, if the process fails at some point or when extra information is needed, the program redoes the steps at a new starting point. The new starting point is selected by a reconstruction of the goal involved in the problem.

PROGRAM PERFORMANCE

The program has been tested on 15 problems with a LTM that averaged 20 potential source analogs from a wide variety of domains. The program produced both goal-less and goal-driven analogies similar to those of our human subjects. In searches for analogies, it used goals when available and resorted to surface information when it had to. Following is an example of input and output for a version of the classic radiation problem (Holland et al., 1986). The target problem of destroying a tumor with a ray powerful enough to destroy intervening tissue may be solved by finding the analogy of dividing an army into small groups so as to avoid overloading fragile bridges.

Frame Name: problem, Frame Class: problem(input)
Properties List:
goal[destroy,tumor]
large(ray)
danger(tissue)
many(guns)
Examples List:
<<EMPTY>>

Returned Solution:

Frame Name: *ANALOGY, Frame Class: solution
Properties List:
goal[destroy,tumor] / goal[destroy,castle]
large(ray) / large(army)
danger(tissue) / danger(bridges)
many(guns) / many(bridges)
Examples List:
solution[divide(army),use(army),direction(multiple)]

In goal-directed search such as this, surface information is unimportant, but the goal is important. Therefore, this solution was chosen only because of its goal. The solution facet, in the examples list, was copied from the analogy as a solution. This solution needs to be tweaked if it is to be used as a solution for the tumor target problem. The tweaking, not yet implemented, would substitute *army* for *ray* in the solution facet of the examples list.

Lambda Search predicts that analogies will be found having no properties in common but the goal. In other words, an analogy to a problem may be selected only by a goal match, with none of the other predicates in the properties list matching. The radiation problem does not demonstrate this, but other test runs on analogies from the research literature did.

We also asked human subjects to evaluate the analogies generated by the program for similarity to human-generated solutions. Results indicated that more similarity was perceived when the program operated under goal-directed search than under goal-less search. Subjects also rated goal-directed solutions as being more effective than goal-less solutions. The program also generated better analogies when consistent goal information was provided, thus simulating Novick's (1988) data

on expert-novice differences. It was assumed that experts would know more about the goals in a domain and would be less influenced by surface information than would novices.

CONCLUSIONS

Lambda search, working on the Concept data structure, simulated human analogical reasoning not only in solutions obtained but also in method. The program accomplished both goal-driven and goal-less tasks, thus unifying the approaches of Gentner and Reiser in a natural and seamless way. The Concept data structure proved to be versatile and easy to use for representing many problems and for storing hierarchies of knowledge relations.

ACKNOWLEDGEMENTS

This research is based on a Masters thesis submitted by Vybihal to the School of Computer Science of McGill University. The research was supported in part by grants to Shultz from the Natural Sciences and Engineering Research Council of Canada and IBM Canada. Please direct correspondence to Thomas R. Shultz, Department of Psychology, McGill University, 1205 Dr. Penfield Ave., Montréal, Québec, Canada H3A 1B1. Email: inoa@musicb.mcgill.ca

REFERENCES

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983) *Data structures and algorithms*. Reading, MA: Addison-Wesley.

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

Faries, J. M., & Reiser, B. J. (1988). Access and use of previous solutions in a problem solving situation. *Proceedings Cognitive Science Society*, **10**, 433-439.

Gentner, D. (1986). *Analogical inference and analogical access*. Unpublished manuscript, University of Illinois.

Hammond, K. J. (1988). Analogical reasoning as a by-product of problem-solving. *Proceedings Cognitive Science Society*, **10**, 302-303.

Holland, J. H., Holyoak, K. J. , Nisbett, R. E., & Thagard, P. (1986). *Induction: Processes of inference, learning, and discovery*. Cambridge, MA: MIT Press.

Lindsay, P. H., & Norman, D. A. (1977). *Human information processing*. New York: Academic Press.

Novick, L. R. (1988). Analogical transfer, problem similarity, and expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, **14**, 510-520.

Reiser, B. J. (1986). The encoding and retrieval of memories of real-world experiences. In J. Galambos, R. Abelson, & J. B. Black (Eds.) *Knowledge structures*, pp. 71-99. Hillsdale, NJ: Lawrence Erlbaum Associates.

Schank, R.C. (1982) *Dynamic memory: A theory of reminding and learning in computers and people*. New York: Cambridge University Press.

Skorstad, J., Falkenhainer, B., & Gentner, D. (1987). Analogical processing: A simulation and empirical corroboration. *Proceedings AAAI*, **6**, 322-326.

Thagard, P., & Holyoak, K. (1988). Analogical problem solving: A constraint satisfaction approach. *Proceedings Cognitive Science Society*, **10**, 299-300.

Vybihal, J. P. (1989). *Search and knowledge representation in analogical reasoning*. Unpublished Masters Thesis, McGill University.