From: AAAI Technical Report WS-94-01. Compilation copyright © 1994, AAAI (www.aaai.org). All rights reserved.

When (not) to use Derivational Analogy: Lessons learned using APU

Sanjay Bhansali
School of EECS
Washington State University
Pullman, WA 99163
bhansali@eecs.wsu.edu

Mehdi T. Harandi
Department of Computer Science
University of Illinois
Urbana IL 61801
harandi@cs.uiuc.edu

Abstract.

We present an experience report in applying derivational analogy to speed up the performance of a prototype system called APU that synthesizes UNIX shell scripts from a problem specification. We present experimental results showing how the system performed, the quality of solutions obtained by using analogy, and how the system scaled up with size of case libraries. We discuss the implications of the results and describe properties of the knowledge representation and the domain that contributed to APU's performance.

Introduction

APU (Automated Programmer for UNIX) is a prototype system that can synthesize UNIX shell scripts from a logic-based specification of a program (given in terms of pre- and post-conditions). The synthesis engine in APU consists of a hierarchical planner that uses a set of rules to decompose a given problem specification into a set of simpler goals. These goals are in turn decomposed into sub-goals until ultimately each sub-goal can be solved by using a UNIX command. If a sub-goal cannot be solved by a using a UNIX command and cannot be decomposed further, the planner backtracks. As the planner attempts to solve a problem it records the derivation history of the rules applied so far. When it finally succeeds in solving a problem, the problem and its associated derivation may be added to APU's case library (this decision has be to made by a user). When new problems are encountered, APU uses a set of heuristics to retrieve an analogous problem from the case library and replays the derivation history of the retrieved problem to solve the new problem more efficiently. The retrieval heuristics were designed to estimate the closeness of two program implementations based on the closeness of their specifications. The retrieval heuristics match problem specifications based on the following features [2]:

1. **Overall solution strategy** - Problems are analogous if the same high-level strategy (e.g. *divide-and-conquer, generate-and-test,* etc. [1]) are applicable. This is detected by matching the logical form of the specifications and ignoring the specific functions, predicates, and objects.
2. **Systematicity** - Problems are analogous if their input and output arguments are parts of a common system of abstract relationship. This is adapted from the *Systematicity Principle* proposed by Gentner [5].
3. **Syntactic features** - Problems are considered analogous if they contain certain pre-defined keywords which dictate the form of the solution (e.g. a recursive solution, or a solution that involves asynchronous wait).
4. **Conceptual distance** - This is used to rank order retrieved analogs by comparing the semantic distance between corresponding concepts in a concept hierarchy. The closer the concepts the more analogous are the problems.

The Appendix gives an illustrative example showing how problems are specified and solved by APU. Details of the system, the planning algorithm, and the retrieval heuristics have been described elsewhere [1, 2]. In this paper we evaluate APU's performance from a case-based reasoning perspective and describe some of the lessons that we have learned from our experiences.

Motivation for Derivational Analogy

To evaluate a case-based system, we need to first specify the purpose for creating the system. Our initial motivation in creating APU was simply to develop an automatic programming system based on planning techniques. We realized quite early that such a system would have a large number of different kinds of rules and that a brute force search for finding a solution would result in unacceptable performance. To improve the planning process we created a sophisticated planning algorithm based on hierarchical planning. The two key features of our approach were:

- The planner assigned a criticality value to each outstanding sub-goal to determine which sub-goal to solve next. This ensured that failure paths were identified as early as possible.
- The rules in the rule-base were categorized according to their generality; when multiple rules were applicable, the planner always chose the most specific rule first. Besides reducing search, this also resulted in better solutions (i.e. more efficient shell scripts).

Although the resultant system was efficient, we believed we could obtain larger speed-ups by reusing the derivation of an analogous problem (when available) in solving a new problem. To test this hypothesis, we built an analogical reasoner component to APU, and created a body of rules and concepts to solve six representative problems (three pairs of analogous problems). We discovered that replay gave us speedups by as much as a factor of ten.

Encouraged by these initial results, we decided to see if these results scaled up, particularly when the cost of retrieving analogs was factored in. (In the initial version there was just one plausible analog for each problem and the cost of retrieval was negligible).

In the rest of this paper, we describe the experimental evaluation of the replay component of APU, analyze the results, and discuss properties of the system and domain that contribute to the results.

Experimental Evaluation

We designed experiments to answer the following set of questions:

- How good are the heuristics in determining appropriate base analogs?
- How effective is derivational analogy in improving the overall problem-solving performance?
- How does the retrieval cost affect performance?
- How does quality of solutions obtained using replay compare with the quality of solutions obtained without replay?
- How does incomplete knowledge affect the performance of the system?

The answers to these questions were expected to give insights on whether, and under what conditions, derivational analogy is likely to improve the problem-solving performance of a system. Note that a significant difference between our system and most case-based reasoning systems (with some exceptions, e.g. [9, 10]) is that in our system the analogical reasoner was only one component of a sophisticated problem-solver. We believe that in many real-life domains there are other domain-specific problem-solving methods that can, and should be, exploited when considering ways of improving the efficiency of the system.

Generating the Data Set

We began by constructing a rule-base for eight problems that are typical of the kind of problems solved using shell scripts in this problem space. The problems included in the set were:

1) List all descendant files of a directory,
2) Find most/least frequent word in a file,
3) Count all files, satisfying certain constraints, in a directory,
4) List duplicate files under a directory,
5) Generate an index for a manuscript,
6) Delete processes with certain characteristics,
7) Search for certain words in a file, and
8) List all the ancestors of a file.

To generate the sample set, we first created a list of abstract operations that can be used to describe the top-level functionality of each of the above problem - *count, list, delete,* etc. - and a list of objects which could occur as arguments to the above operations *directory, file, system, line, word.* Then we created another list of the predicates and functions in our concept dictionary which relate these objects, e.g., *occurs, owned, descendant, size, cpu-time, word-length, line-number,*

Next, we used the signatures (i.e. number and type of arguments) of predicates represented in APU to generate all legal combinations of operations and argument types. For example, for the *count* operation, the following instances were generated: (*count file system*), (*count file directory*), (*count word file*), (*count character file*), (*count line file*), (*count string file*), (*count process system*).

In a similar fashion, a list of all legal constraints were generated, using the second list of predicates and functions. Examples of constraints generated are *(occurs file directory), (descendant directory directory), (= int (line-number word file)),* and (= string (owner file)). Constraints that were trivial or uninteresting were pruned away *(= int int).*

Next we combined these constraints with the top-level operations to create a base set of problems. We restricted each problem to have a maximum of three conjunctive constraints. From this set a random number generator was used to select thirty-seven problems, which together with the initial set formed our sample population of forty-five problems.

The final step consisted of translating the informal problem descriptions into a formal specification using the most "natural" or likely formulation of the problem. For example, for the problem *(most-frequent word file),* the corresponding post-condition with a word being the output variable and a file being the input variable is more likely, rather than the reverse (which would generate a program to find that file in which a given word is the most frequent).

Goodness of Heuristics

For the purposes of this workshop, the exact performace of the various heuristics is not relevant since the retrieval heuristics are very specific to this domain. Hence, we simply summarize the results here (for details see [2]):

- Using all four heuristics APU performed almost as well as a human (it failed to retrieve the best analog in only 5% of the cases).
- The *systematicity* and *solution strategy* were the two most important heuristics. Turning both off caused the system to miss the best analog in all but four cases, whereas each one by itself was able to retrieve the best analog in about 80-85% cases.
- Cases where APU missed the best analog were insightful in indicating the reasons why analogy works in APU (this will be discussed in detail later).

Speedup using Derivational Analogy

To measure speedup we built a case library by randomly selecting a set of ten base analogs from the sample set. From the same sample another set of problems were randomly selected to be the target problems. We measured the time to synthesize each of the target problems with and without using analogy. The experiment was repeated with different sets of base and target problems and the average speed-up was computed for all the experiments.

We observed that the average time to synthesize programs is reduced by a factor of 2 when derivational analogy is used [2]. This is a very modest performance gain compared to our initial estimations and compared to speedups obtained by other learning programs e.g. [6, 10].

The primary reason for this is the nature of the search space in APU. The planner uses sophisticated heuristics and domain-specific knowledge to ensure that its rule selection methodology greatly reduces the need for backtracking. As a result the planner spends most of its time in pattern matching and in selecting the right rule rather than on search. Thus, during replay the saving is obtained by eliminating the cost of rule selection rather than by

eliminating search. This leads us to conclude that *sophisticated problem-solving methods and domain-specific problem-solving knowledge, when available, may be more effective in improving problem-solving performance than replay*. In other words, an analogical reasoner should be considered as *one* technique among several possibilities when exploring methods for improving the performance of a problem solver.

Second, the largest speedups were obtained when parts of a solution could be copied. Most rules in APU's knowledge-base decomposed problems so that there was no interaction between sub-goals. As a result, whenever an identical sub-goal is observed in an analogous problem, APU can simply copy the derivation subtree beneath that sub-goal in the source problem. This implies that *in domains where problems are decomposable, with little or no interaction between sub-goals, and where similar sub-problems recur frequently, one would obtain high speed-ups by using replay*. This is in agreement with observations made by other researchers in CBR [7].

Our data also supported the following intuitive results about replay:

• speed-ups tend to be greater with larger problem sizes (owing to the reduced overheads of matching analogs as a fraction of the total problem-solving time).

• The degradation in performance when analogies failed was less severe than the speedups obtained when analogies matched suggesting that unless the number of mismatches are much greater than the number of matches, replay should prove effective.

Finally, our results are based on a random sampling of problems from a population. We conjecture that in many realistic applications problems do not have such a distribution and the 80-20 rules applies: 80% of the problems are generated from 20% of the problem space indicating that there are many instances where problems are similar compared to instances where they are not. In such cases, derivational analogy should yield higher performance gains. However, further empirical testing is needed to confirm this hypothesis.

Effect of Case Library size.

To determine how the time to retrieve analogs depends on the size of the case history library, we incrementally increased the size of the derivation history library (in steps of 5, selecting new problems at random), and measured the time taken to retrieve analogs for a fixed set of target problems. Figure 2 shows the result of one typical run of this experiment.

When problems are stored in the case library they are indexed using a set of keys that are derived from the problem specification using APU's retrieval heuristics. When a new problem specification is encountered, APU constructs a set of keys for the new problem using the same heuristics and then uses them to retrieve cases from the library. The conceptual distance heuristic is then used to rank order the retrieved cases. Thus, the retrieval strategy in APU can be thought of as a sophisticated hashing scheme. However, unlike a regular hashing scheme problems do not have a unique key and several different keys may be used to index the same problem. The experimental results showed that in such a situation the cost of retrieval increased almost linearly with the average number of problems per feature.

This provides a hypothesis as to when problems should be stored in the derivation history library: *If adding a set of problems to the library increases the ratio problems/feature, the new problems are probably similar to the problems already existing in the library, and hence should not be added to the case library*. On the other hand, if the ratio decreases or remains the same, the problems are probably different from the ones in the library and should be added.

An alternative approach to the utility problem that has been recently reported involves using a more efficient matching of learned rules [4].

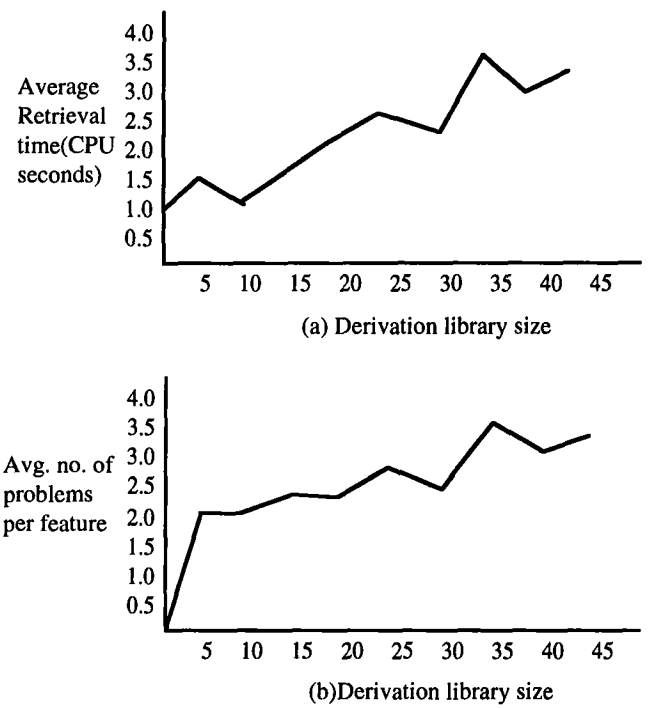


Figure 2. (a) Average time to retrieve analogs as a function of library size. (b) Average number of problems per feature as a function of library size.

Quality of solutions

It is well-known that there is a trade-off in the quality of solutions obtained versus time spent in searching for alternative solutions during replay [8]. One strategy that can be used to balance the trade-off is the following: *At each step during replay allow the problem-solver to search for an alternative, superior solution provided the search time does not exceed a certain threshold value.* However, it is not clear what the threshold value should be. In APU we experimented with different values of the threshold. The strategy that worked the best was: *only look for alternative 1-step derivations during replay*. Increasing the threshold beyond this value did not improve the quality of solutions

but degraded performance considerably. Likewise, ignoring alternative solutions and blindly replaying the derivation histories caused APU to miss several efficient solutions.

We believe that the explanation for this lies in the way rules are categorized in APU according to generality. When two or more rules are applicable to a problem, the most specific rule almost always leads to a better solution. APU currently has three levels of rules. The most specific rules are the ones that apply at the leaf nodes and solve a sub-goal by applying a UNIX command. Only these rules are considered as alternatives during replay. A generalization of this heuristic would be to consider alternative plans *whenever a more specific rule than the one being replayed is applicable* to the current sub-goal. We have not yet experimented with this heuristic.

A second experimental observation regarding quality of solutions was that the retrieval heuristics were important in determining quality (see [2] for an example). In APU, the replay algorithm only considers the best analog retrieved from the case library for solving a particular goal.[1] If that turns out to be an inappropriate analog, the replay algorithm either produces a partial solution or passes the problem back to the planner (this choice is controlled by a user - see below). In either case there is a penalty to be paid - either in terms of the solution (partial versus complete) or in terms of performance. In an earlier set of experiments we tuned the algorithm so that it always chose to produce the best partial solution it could. The experiment clearly showed the effects of the different heuristics in choosing the best solution and helped us to fine-tune the heuristic used to rank retrieved cases. This observation served to confirm our intuition that the retrieval methodology is critical in determining whether derivational analogy would be successful. Section 4.0 gives certain guidelines, based on our experience, on how to organize background knowledge about a domain to help in effectively retrieving base analogs.

Partial solutions

This is related to the issue of solution quality. We believe that it is unrealistic to assume that a completely automatic system could be constructed for many real-life problems because of the difficulty of ensuring completeness and correctness of problem-solving rules. Therefore, it is necessary to consider situations when the system is unable to completely solve a problem but can provide a reasonable partial solution that can be extended by the user. Note that there is an important difference between APU and some CBR systems in that in APU's domain there is a well-defined notion of a correct answer. CBR domains such as law, cooking, etc. do not have a unique "correct" answer. Each answer therefore has to be evaluated in the real world and the distinction between a complete and partial solution is irrelevant. If we insist that the system always provide a complete solution whenever such a solution exists, then the benefits of using replay can be completely negated! This is because whenever an analogy fails at a sub-goal and the planner cannot produce a solution for that sub-goal, the only way to know whether a solution exists is to back up all the way using the underlying planner and exhaust all possibilities. In such a situation derivational analogy is not useful since, in effect, the derivation history is being ignored.

APU finesses this problem by querying the user whether to backtrack or continue whenever it encounters a goal for which there are no applicable rules. However, we believe that this is a serious limitation of derivational analogy in domains where the problem solving knowledge is relatively incomplete and complete solutions (whenever they exist) are highly desirable.

Why does analogy work in APU?

As with most AI systems, the effectiveness of APU depends heavily on the representation of the domain. The key features of APU's representation scheme are the abstraction hierarchies of objects, predicates, and functions and the formulation of the rules in terms of these abstractions [2]. In this section we briefly discuss how the abstraction hierarchies are formed, and what properties of the abstraction hierarchies, the rule base, and the analogical detection mechanism determine the effectiveness of APU.

Two basic guidelines in forming the abstraction hierarchy in our system are the following:

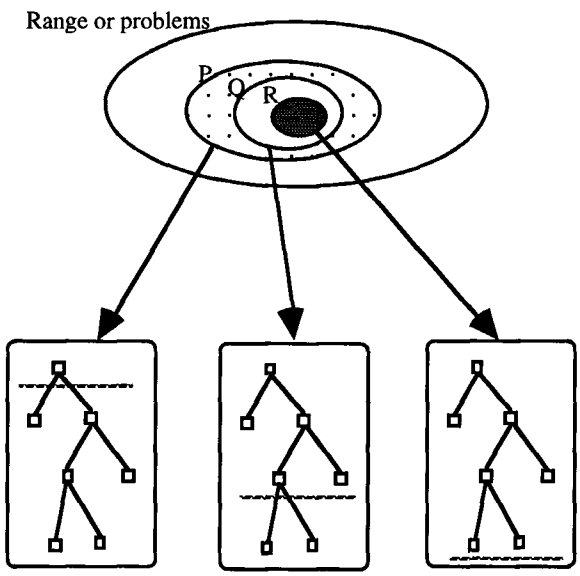
(1) If a common function or predicate can be applied to objects *A* and *B*, consider classifying *A* and *B* under a more general object. For example, the operation *sort-in-alphabetical* order can be applied to a set of characters, words, or lines; hence *characters, words,* and *lines* are grouped into a more general object *line-object.* (2) If a plan for achieving two goals expressed using predicates (or functions) *f* and *g* share common sub-goals, consider classifying *f* and *g* into a more general predicate (function). For example, a plan for finding the largest element in an unordered collection of elements (using some ordering operator) and a plan for finding the smallest element in an unordered collection of elements share the common sub-goal of first sorting the collection of elements. Therefore the predicates *largest* and *smallest* may be grouped under a common predicate called *extremum.*

One of the prerequisites for analogy to work is that there be a large proportion of general rules, i.e. rules formulated in terms of general objects, predicates and functions (hereby called *concepts*) in the abstraction hierarchy. Otherwise, if we only had specific rules written in terms of specific concepts (forming the leaves of the abstraction hierarchy), there would be very little analogical transfer of a solution derivation from one problem to another.

Secondly, for analogy to succeed *the features that are used to retrieve analogous problems should be good predictors of the sequence of rules needed to solve the problem.* Figure 2 shows the relationship between sets of problems, applicable derivations, and the features used to detect analogous problems. Different features would correspond to different sizes of the subsets of problems and

[1] We discovered that trying more than one analog for the same goal resulted in unacceptable performance.

the applicable derivations. A feature would be most predictive if the subset Q coincides with subset R; such a feature could be used to retrieve all, and only those, problems for which the entire derivation is applicable. However, if R is very small, such a feature would not be general enough for analogy. On the other hand if Q coincided with the set P, then the feature would be very general, but would be a poor predictor of the subsequent rules to apply. An ideal feature for analogy is one that maximizes both the subset of problems which it identifies (for generality) as well as the part of the solution derivation that is applicable to them (for effectiveness).



(a) (b) (c)

Figure 2. The relationship between problems, plans, and features used to detect analogous problems. P is the set of all problems to which the first rule in a derivation applies, Q is the subset of problems that match the feature used for plan retrieval, and R is the subset of problems for which the entire derivation is applicable.

In APU, when a rule is used to decompose a goal, the bindings of the rule variables to the goal expressions determine the subsequent sub-goals and thus, implicitly, the subsequent sequence of rules to apply. Therefore the bindings of the rule variables provide a feature for analogy detection. One can imagine three ways in which these bindings can be used to predict other goals on which the same sequence of rules would apply. At one extreme, one could completely ignore the bindings of variables, and say that for any goal expression which matches the rule, the original sequence of rules should apply. However, if the rule is very general, it may be poorly correlated to the subsequent sequence of rules to be used, and thus the analogy is likely to fail as often as it succeeds. This corresponds to case (a) in figure 2. At the other extreme, one could use the exact bindings and say that if a rule matches another goal with the corresponding variables bound to the same expressions (up to variable renaming) then the original sequence of rules would apply. This corresponds to case (c) in figure 2 and is not general enough for analogy. A third, intermediate approach is to extract certain features that characterize the bindings and use them to predict the sequence of subsequent applicable rules. If the features that are used to characterize the bindings are both general (to permit analogical transfer of solutions to many other problems) and well correlated with the subsequent rules needed to solve the problem, then they can be fruitfully used to detect analogous problems (case (b) in figure 2).

The key to APU's success is that its retrieval heuristics use a set of features that provide precisely such a characterization of the variable bindings. The ontology of intermediate concepts represented in the abstraction hierarchy plays a central role in the characterization of the bindings. Because of the methodology used to construct the abstraction hierarchy, there is a strong correlation between the intermediate level concepts used to characterize variable bindings and the subsequent rule sequence that is used, and at the same time the characterization is general enough to be applicable to several different problems.

A property of the domain that contributes to APU's success is the availability of a rich set of planning operators (i.e. the UNIX commands and subroutines) that make it possible to represent the objects in the domain in terms of abstraction hierarchies. Thus, when a plan is formulated in terms of general objects, there is a high correlation between the various plans obtained by instantiating the general object by specific ones in the abstraction hierarchy, and their completions. For example, in APU *files* and *directories* are classified under the abstract object *directory-object* and most plans are formulated in terms of *directory-object*. The analogy between files and directories is effective because for most UNIX commands that operate on *files* there is a corresponding (possibly same) command that operates on *directories*.

Conclusion

Derivational analogy was proposed as a mechanism that could be used to reduce problem-solving effort [3]. However, the usefulness of this approach needs to be experimentally validated in non-trivial and novel domains. We have described our experiences and some lessons learned in using derivational analogy to synthesize UNIX shell scripts. These lessons provide some insights on when derivational analogy is likely or not likely to be a useful technique, and what issues need to be examined more carefully in attempting to scale up to real-world problems.

Our current work includes expanding the rule-base in APU to include more domain-independent, problem-solving (as opposed to UNIX-specific) rules to increase the repertoire of problems that can be solved using a fixed set of UNIX commands. This will increase the search space of APU. Our conjecture is that in such a situation, the benefits of using replay would increase significantly.

An interesting side-effect of implementing the analogical reasoner was that it served as a valuable tool for knowledge acquisition and refinement. In trying new problems APU often surprised us by failing to synthesize solutions for problems although an obviously analogous problem could be solved. Examining the node at which replay failed quickly revealed rules that were missing, or overly specific, or objects/predicates missing in APU's concept hierarchy. We plan to investigate the role of derivational analogy as a knowledge acquisition technique by exploiting replay failures to propose new rules and new concepts in the concept hierarchy.

References

[1] Bhansali, S., Domain-based program synthesis using planning and derivational analogy. 1991, Department of Computer Science, University of Illinois at Urbana-Champaign.

[2] Bhansali, S. and M.T. Harandi, *Synthesis of UNIX programs using derivational analogy*. Machine Learning, 1993. **10**(1).

[3] Carbonell, J.G. *Derivational analogy and its role in problem solving*. in *Third National Conference on Artificial Intelligence*. 1983. Washington, D.C.

[4] Doorenbos, R.B. *Matching 100,000 Learned Rules*. in *Proceedings of AAAI-93*. 1993. Washington, D. C.

[5] Gentner, D., *Structure-mapping: A theoretical framework for analogy*. Cognitive Science, 1983. **7**(2), p. 155-170.

[6] Kambhampati, S., *A validation-structure based theory of plan modification and reuse*. Artificial Intelligence, 1992. **55**(2-3), p. 193-258.

[7] Koton, P.A. *Evaluating case-based problem solving*. in *Proc. of a Workshop on Case-based Reasoning*. 1989. Pensacola Beach, FL.

[8] Mostow, J., *Design by derivational analogy: issues in the automated replay of design plans*. Artificial Intelligence, 1989. **40**, p. 119-184.

[9] Thagard, P. and K.J. Holyoak. *Why indexing is the wrong way to think about analog retrieval*. in *Proc. of a Workshop on Case-based Reasoning*. 1989. Pensacola Beach, FL.

[10] Veloso, M. and J.G. Carbonell, *Derivational analogy in Prodigy: automating case acquisition, storage and utilization*. Machine Learning, 1993. **10**(3), p. 249-278.1. Bhansali, S. and M.T. Harandi, *Synthesis of UNIX programs using derivational analogy*. Machine Learning, 1993. **10**(1).

Appendix

The following example gives a flavor of the kinds of problems and how they are solved by APU. In the following, the output of APU is in normal font, the user's response is in **bold**, and our annotations are in *italics*.

Beginning of session
> (get-problem)
Enter problem description:-
NAME: **max-word**
INPUT: **?f :file**
OUTPUT: **?w :word**
PRECONDITION: **true**
POSTCONDITION: **(most-frequent ?w (BAG (?x :word) :ST (occurs ?x ?f)))**
This is a problem of finding the most commonly used word in a text-file.
> (solve-problem)
Failed to find an analogous problem.
Solving problem using planner.
APU first tries to solve a problem by analogy. There are no analogs for this problem in APU's knowledge-base. So this problem is solved by calling the underlying planner.
Expanding sub-goal: (most-frequent
?w
(BAG (?x :word) :ST (occurs ?x ?f)))
A sub-goal is expanded by using transformational rules and reducing it to a sequence of one or more sub-goals.
Expanding sub-goal:
(= ?s1 (SET (?w1 :word ?n:integer) :ST
(and (occurs ?w1 ?f)
(= ?n (count-of ?w1 (BAG (?x :word)
:ST (occurs ?x ?f)))))))
Expanding sub-goal:
(= ?s2 (SET (?w2 :word) :ST (occurs ?w2 ?f)))
Expanding sub-goal:
(= ?n1 (SET (?x1 :integer) :ST (= ?x1 (count-of ?var3
(BAG (?x2 :word) :ST (occurs ?x2 ?f))))))
Unable to solve sub-goal:
(= ?n1 (SET (?x1 :integer) :ST (= ?x1 (count-of ?var3
(BAG (?x2 :word) :ST (occurs ?x2 ?f))))))
Choose one of the following: 1. Backtrack 2. Continue
APU starts solving the problem in a top-down manner by decomposing sub-goals to simpler sub-goals, choosing at each point the most critical sub-goal. It discovers that it cannot solve or further decompose the sub-goal of computing the count of the number of times a given word occurs in a file and asks the user whether to backtrack or continue and generate a partial solution.
> 2
The user opts for a partial solution.
The rest of the trace is omitted.
...
Finished generating plan. A solution to the problem is:
MAX-WORD:

```
cat ?f |                    # list the given input file
tr -s ' ' '\012' |          # replace spaces by NEWLINE
tr -s '        ' '\012' |   # replace tabs by NEWLINE
sort |                      # sort the list
uniq > /tmp/file728         # remove duplicates
WHILE read ?var3            # for all words in the list
DO
  [SET (?x1 :integer) # get count of word ?var3 in file ?f
      :ST (= ?x1 (count-of ?var3 (BAG (?x2 :word) :ST
                                 (occurs ?x2 ?f)))) > ?n1]
  FOR ?var4 IN ?n1
  DO                        # print (word, word-count) in a file
     echo $?var3 $?var4 >> /tmp/file729
  DONE
DONE < /tmp/file728         # the input to the while loop
sort +2 -r /tmp/file729 |   # sort on the second field
head -1 > /tmp/file730      # select first element of list
```

```
    set ?w=`awk '{ print $1 }' /tmp/file730`   # select 1st field
```

Look for other solutions? (y/n) **n**
Add the problem to the derivation history library? (y/n) **y**
Saving derivation history ... done
APU adds the above problem and its derivation to its derivation history library. This can now be used by the system to solve analogous problems. This is illustrated next.

> (get-problem)
Enter problem description:-
NAME: **max-file**
INPUT: **?d :directory**
OUTPUT: **?f :file**
PRECONDITION: **true**
POSTCONDITION: **(most-frequent ?f (BAG (?x :file) :ST (descendant ?x ?d)))**
In this problem the user wants to find the most common file name, among all files that are descendants of a given directory.
>(solve-problem)
Found analogy! Exploring further ...
APU automatically detects the max-word problem as the best analog for the current problem, retrieves its derivation from the derivation history library and replays that derivation.
Found analogy! Exploring further ...
The analogy continues for the decomposed sub-goals.
Found analogy! Exploring further ...
Found analogy! Exploring further ...
Analogy failed at sub-goal: (= ?s3 (SET (?F :file) :ST (descendant ?f1 ?d)))
The earlier rule is no longer applicable to this sub-goal.
No alternate rule available! Solving problem using the planner.
No alternative rules are stored in the derivation history. So APU passes this sub-problem back to the planner.
Found analogy! Exploring further ...
APU discovers another analogy to this problem. Presumably it has solved a similar problem earlier, whose derivation has been stored in the derivation history library.
Found exact match! Copying sub-tree ... done.
The earlier problem was identical to the current problem, so the solution is simply copied.
Finished generating plan.
Found analogy! Exploring further ...
Having successfully solved the sub-problem the planner passes control back to the analogical reasoner which continues to apply the original analogy.
Unable to solve sub-goal:
(= ?n1 (SET (?x1 :integer) :ST (= ?x1 (count-of ?var3 (BAG (?x2 :file) :ST(descendant ?x2 ?d))))))
APU again fails to solve or decompose the problem of computing the count of the number of times a given file occurs in a collection of files. No alternative rules are applicable. So APU queries the user.
Choose one of the following: 1. Backtrack 2. Continue
> 2
Found analogy! Exploring further ...
Analogy continues on rest of the sub-goals.
The rest of the derivation is omitted.
...
Finished generating plan.
MAX-FILE:

```
    descendant ?d |          # get descendants of directory ?d
    sort |                   # sort the list of descendants
    uniq > /tmp/file801      # remove duplicates
    WHILE read ?var3
    DO            # get count of ?var3 in list of descendants
     [SET (?x1 :integer) :ST (= ?x1 (count-of ?var3
        (BAG (?x2 :word) :ST (descendant ?x2 ?d)))) > ?n1]
     FOR ?var4 IN ?n1
     DO                      # print (word,word-count) in a file
        echo $?var3 $?var4 >> /tmp/file802
     DONE
    DONE < /tmp/file801  # input to the while loop
    sort +2 -r /tmp/file802 | # sort on the second field
    head -1 > /tmp/file803   # select first element of list
    set ?w=`awk '{ print $1 }' /tmp/file803`# select first field
DESCENDANT:
    if test -f $1
    then
      echo $1                # output the filename
    else                     # else input is a directory
      FOR var IN `ls $1`     # for all objects in the directory
      DO
        descendant $1/$i     # call the routine recursively
      DONE
    fi
```

End of session