Pruning Duplicate Nodes in Depth-First Search

Larry A. Taylor and **Richard E. Korf** *
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024
ltaylor@cs.ucla.edu

Abstract

Best-first search algorithms require exponential memory, while depth-first algorithms require only linear memory. On graphs with cycles, however, depth-first searches do not detect duplicate nodes, and hence may generate asymptotically more nodes than best-first searches. We present a technique for reducing the asymptotic complexity of depth-first search by eliminating the generation of duplicate nodes. The automatic discovery and application of a finite state machine (FSM) that enforces pruning rules in a depth-first search, has significantly extended the power of search in several domains. We have implemented and tested the technique on a grid, the Fifteen Puzzle, the Twenty-Four Puzzle, and two versions of Rubik's Cube. In each case, the effective branching factor of the depth-first search is reduced, reducing the asymptotic time complexity.

Introduction—The Problem

Search techniques are fundamental to artificial intelligence. Best-first search algorithms such as breadth-first search, Dijkstra's algorithm [Dijkstra, 1959], and A* [Hart *et al.*, 1968], all require enough memory to store all generated nodes. This results in exponential space complexity on many problems, making them impractical.

In contrast, depth-first searches run in space linear in the depth of the search. However, a major disadvantage of depth-first approaches is the generation of duplicate nodes in a graph with cycles [Nilsson, 1980]. More than one combination of operators may produce the same node, but since depth-first search does not store the nodes already generated, it cannot detect the duplicates. As a result, the total number of nodes generated by a depth-first search may be asymptotically more than the number of nodes generated by a best-first search.

*This research was partially supported by NSF Grant #IRI-9119825, and a grant from Rockwell International.

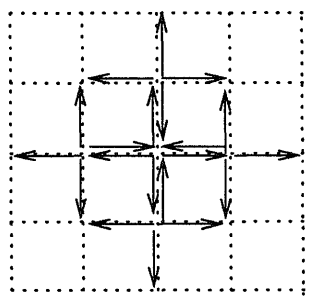

Figure 1: The grid search space, explored depth-first to depth 2, without pruning.

To illustrate, consider a search of a grid with the operators: Up, Down, Left and Right, each moving one unit. A depth-first search to depth r would visit 4^r nodes (figure 1), since 4 operators are applicable to each node. But only $O(r^2)$ distinct junctions are visited by a breadth-first search. Thus, a depth-first search has exponential complexity, while a breadth-first search has only polynomial complexity.

To reduce this effect, we would like to detect and prune duplicate nodes in a depth-first search. Unfortunately, there is no way to do this on an arbitrary graph without storing all the nodes. On a randomly connected explicit graph, for example, the only way to check for duplicate nodes is to maintain a list of all the nodes already generated. A partial solution to this problem is to compare new nodes against the current path from the root [Pearl, 1984]. This detects duplicates in the case that the path has made a complete cycle. However, as we saw in the grid example, duplicates occur when the search explores two halves of a cycle, such as up-left and left-up. Only a fraction of duplicates can be found by comparing nodes on the current path [Dillenburg and Nelson, 1993]. Other node caching schemes have been suggested [Ibaraki, 1978] [Sen and Bagchi, 1989] [Chakrabarti *et al.*, 1989] [Elkan, 1989], but their utility depends on the implementation (costs per node generation).

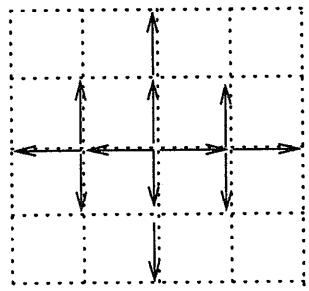

Figure 2: The grid search space, explored depth-first to depth 2, with pruning.

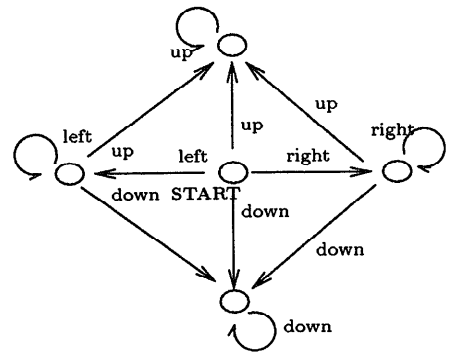

Figure 3: FSM corresponding to the search with FSM pruning, eliminating duplicate nodes.

We present a new technique for detecting duplicate nodes that does not depend on stored nodes, but on another data structure that can detect duplicate nodes that have been generated in the search's past, and nodes that will be generated in the future. This technique uses limited storage efficiently, uses only constant time per node searched, and reduces the effective branching factor, hence reducing the asymptotic time complexity of the search.

The FSM Pruning Rule Mechanism

Exploiting Structure

We take advantage of the fact that most combinatorial problems are described implicitly. If a problem space is too large to be stored as an explicit graph, then the it must be generated by a relatively small description. This means that there is structure that can be exploited. Precisely the problems that generate too many nodes to store are the ones that create duplicates that can be detected and eliminated. For this paper, a node in the search space is represented by a unique vector of values.

For example, the grid problem, the operator sequence Left-Right will always produce a duplicate node. Rejecting inverse operator pairs, including in addition Right-Left, Up-Down, and Down-Up, reduces the branching factor by one, and the complexity from $O(4^r)$ to $O(3^r)$. Inverse operators can be eliminated by a finite state machine (FSM) that remembers the last operator applied, and prohibits the application of the inverse. Most depth-first search implementations already use this optimization, but we carry the principle further.

Suppose we restrict the search to the following rules: go straight in the X-direction first, if at all, and then straight in the Y-direction, if at all, making at most one turn. As a result, each point (X,Y) in the grid is generated via a unique path: all Left moves or all Right moves to the value of X, and then all Up moves or all Down moves to the value of Y. Figure 2 shows a search to depth two carried out with these rules. Figure 3 shows an FSM that implements this search strategy. The search now has time complexity $O(r^2)$, reducing the complexity from exponential to quadratic.

Each state of this machine corresponds to a different last move made. The FSM is used in a depth-first search as follows. Start the search at the root node as usual, and start the machine at the START state. For each state, the valid transitions are given by the arrows which specify the possible next operators that may be applied. For each new node, change the state of the machine based on the new operator applied to the old state. Operators that generate duplicate nodes do not appear. This prunes all subtrees below such redundant nodes. The time cost of this optimization is negligible.

Next, we present a method for automatically learning a finite state machine that encodes such pruning rules from a description of the problem.

Learning the FSM

The learning phase consists of two steps. First, a small breadth-first search of the space is performed, and the resulting nodes are matched to determine a set of operator strings that produce duplicate nodes. The operator strings represent portions of node generation paths. Secondly, the resulting set of strings is used to create the FSM which recognizes the strings as a set of keywords. If we ever encounter a string of operators from this set of duplicates, anywhere on the path from the root to the current node, we can prune the resulting node, because we are guaranteed that another path of equal or lower cost exists to that node.

Exploratory Breadth-first Search Suppose we apply the search for duplicate strings to the grid space. In a breadth-first search to depth 2, 12 distinct nodes are generated, as well as 8 duplicate nodes, including 4 copies of the initial node (see figure 4(a)). We need to match strings that produce the same nodes, and then make a choice between members of the matched pairs of strings. We can sort the nodes by their representa-

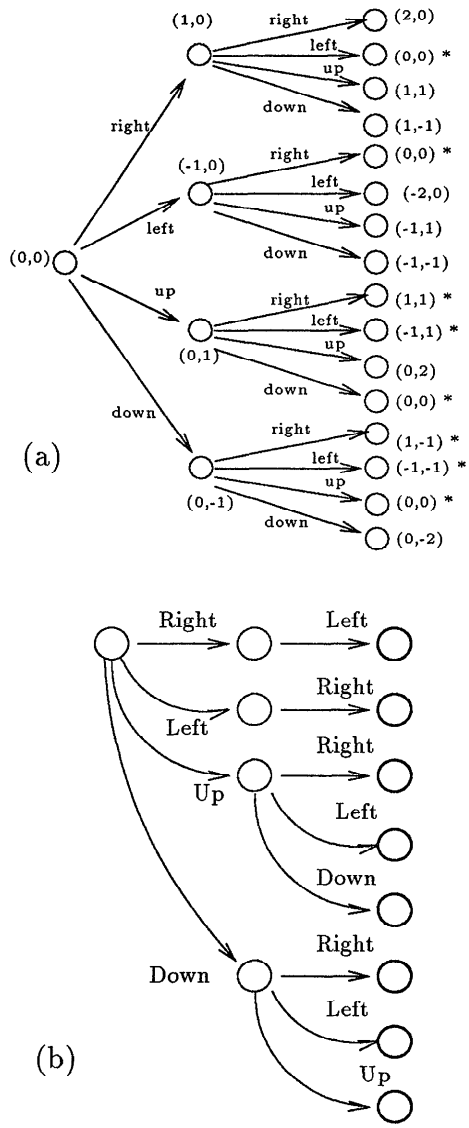

Figure 4: Tree and trie pair. (a) A tree corresponding to a breadth-first search of the grid space. Duplicates indicated by (*). (b) A trie data structure recognizing the duplicate operator strings found in the grid example. The heavy circles represent rejecting (absorbing) states.

tions to make matches, and use the cost of the operator strings to make the choices between the pairs. If we order the operators Right, Left, Up, Down, then the operator sequences that produce duplicate nodes are: Right-Left, Left-Right, Up-Right, Up-Left, Up-Down, Down-Right, Down-Left, and Down-Up (figure 4(a)).

This exploratory phase is a breadth-first search. We repeatedly generate nodes from more and more costlier paths, making matches and choices, and eliminating duplicates. The breadth-first method guarantees that duplicates are detected with the shortest possible operator string that leads to a duplicate, meaning that no duplicate string is a substring of any other. The exploratory phase is terminated by the exhaustion of available storage (in the case of the examples of this paper, disk space).

Construction of the FSM The set of duplicate operator strings can be regarded as a set of forbidden words. If the current search path contains one of these forbidden words, we stop at that point, and prune the rest of the path. Thus, we want to recognize the occurrence of these strings in the search path. The problem is that of recognizing a set of keywords (i.e., the set of strings that will produce duplicates) within a text string (i.e., the string of operators from the root to the current node).

This is the bibliographic search problem. Once the set of duplicate strings to be used is determined, we apply a well known algorithm to automatically create an FSM which recognizes the set [Aho *et al.*, 1986]. In this algorithm, a *trie* (a transition diagram in which each state corresponds to a prefix of a keyword) is constructed from the set of keywords (in this case, the duplicate operator strings). This skeleton is a recognition machine for matching keywords that start at the beginning of the 'text' string. The remaining transitions for mismatches are calculated by a 'failure transition function'. The states chosen on 'failure' are on paths with the greatest match between the suffix of the failed string and the paths of the keyword trie.

A trie constructed from the duplicate string pairs from the grid example is shown in figure 4(b). A machine for recognizing the grid space duplicate string set is shown in figure 3. Notice that the arrows for rejecting duplicate nodes are not shown. As long as the FSM stays on the paths shown, it is producing original (non-duplicate) strings. The trie for the keywords used in its construction contains these rejected paths.

Learning phase requirements All nodes previously generated must be stored, so the space requirement of the breadth-first search is $O(b^d)$, where b is the branching factor, and d is the exploration depth. The actual depth employed will depend on the space available for the exploration phase. The exploration terminates when b^d exceeds available memory or disk space. Duplicate checking can be done at a total cost of $O(N \log N) = O(b^d \log b^d) = O(db^d \log b)$ if the nodes

are kept in an indexed data structure, or sorting is employed. This space is not needed during the actual problem-solving search.

The time and space required for the construction of the FSM using a trie and 'failure transition function' is at most $O(l)$, where l is the sum of the lengths of the keywords [Aho *et al.*, 1986].

The breadth-first exploration search is small compared to the size of the depth-first problem-solving search. Asymptotic improvements can be obtained by exploring only a small portion of the problem space, as shown by the grid example. Furthermore, the exploratory phase can be regarded as creating compiled knowledge in a pre-processing step. It only has to be done once, and is amortized over the solutions of multiple problem instances. The results presented below give examples of such savings.

Using the FSM

Incorporating a FSM into a problem-solving depth-first search is efficient in time and memory. For each operator application, checking the acceptance of the operator consists of a few fixed instructions, e.g., a table lookup. The time requirement per node generated is therefore $O(1)$. The memory requirement for the state transition table for the FSM is $O(l)$, where l is the total length of all the keywords found in the exploration phase. The actual number of strings found, and the quality of the resulting pruning, are both functions of the problem description and the depth of the duplicate exploration.

Necessary Conditions for Pruning

We must be careful in pruning a path to preserve at least one optimal solution, although additional optimal solutions may be pruned. The following conditions will guarantee this. If A and B are operator strings, B can be designated a duplicate if: (1) the cost of A is less than or equal to the cost of B, (2) in every case that B can be applied, A can be applied, and (3) A and B always generate identical nodes, starting from a common node.

If these conditions are satisfied, then if B is part of an optimal solution, then A must also be part of an optimal solution. We may lose the possibility of finding multiple solutions of the same cost, however.

In all the examples we have looked at so far, all operators have unit cost, but this is not a requirement of our technique. If different operators have different non-negative costs, we have to make sure that given two different operator strings that generate the same node, the string of higher cost is considered the duplicate. This is done by performing a uniform-cost search [Dijkstra, 1959] to generate the duplicate operator strings, instead of a breadth-first search.

So far, we have assumed that all operators are applicable at all times. However, the Fifteen Puzzle contains operator preconditions. For example, when the blank is in the upper left corner, moving the blank Left or Up is not valid.

Such preconditions may be dealt with in two steps. For purposes of the exploratory search, a generalized search space is created, which allows all possible operator strings. For the Fifteen Puzzle, this means a representation in which all possible strings in a 4x4 board are valid. This can be visualized as a "Forty-eight Puzzle" board, which is 7x7, with the initial blank position at the center.

The second step is to test the preconditions of matched A and B strings found in the exploratory search. B is a duplicate if the preconditions of A are implied by the preconditons of B. For the Fifteen puzzle, the preconditions of a string are embodied in the starting position of the blank. If the blank moves farther in any direction for string A than for string B, then there are starting positions of the blank for which B is valid, but not A. In that case, B is rejected as a duplicate. A similar logical test may be applied in other domains. A more complete description is given in [Taylor, 1992].

Experimental Results

The Fifteen Puzzle

Positive results were obtained using the FSM method combined with a Manhattan Distance heuristic in solving random Fifteen Puzzle instances.

The Fifteen Puzzle was explored breadth-first to a depth of 14 in searching for duplicate strings. A set of 16,442 strings was found, from which an FSM with 55,441 states was created. The table representation for the FSM required 222,000 words. The inverse operators were found automatically at depth two. The thousands of other duplicate strings discovered represent other non-trivial cycles.

The branching factor is defined as $\lim_{d\to\infty} N(d)/N(d-1)$, where $N(d)$ is the number of nodes are generated at depth d. The branching factor in a brute-force search with just inverse operators eliminated is 2.13. This value has also been derived analytically. Pruning with an FSM, based on a discovery phase to depth 14, improved this to 1.98. The measured branching factor decreased as the depth of the discovery search increased. Note that this is an asymptotic improvement in the complexity of the problem solving search, from $O(2.13^d)$ to $O(1.98^d)$, where d is the depth of the search. Consequently the proportional savings in time for node generations increases with the depth of the solution, and is unbounded. For example, the average optimal solution depth for the Fifteen puzzle is over 50. At this depth, using FSM pruning in brute-force search would save 97.4% of nodes generated.

Iterative-deepening A* using the Manhattan Distance heuristic was applied to the 100 random Fifteen Puzzle instances used in [Korf, 1985]. With only the inverse operators eliminated, an average of 359 million

nodes were generated for each instance. The search employing the FSM pruning generated only an average of 100.7 million nodes, a savings of 72%.

This compares favorably with the savings in node generations achieved by node caching techniques applied to IDA*. The FSM uses a small number of instructions per node. If it replaces some method of inverse operator checking, then there is no net decrease in speed. The method of comparing new nodes against those on the current search path involves a significant increase in the cost per node, with only a small increase in duplicates found, compared to the elimination of inverse operators [Dillenburg and Nelson, 1993]. Although it pruned 5% more nodes, the path comparison method ran 17% longer than inverse operator checking. MREC [Sen and Bagchi, 1989], storing 100,000 nodes, reduced node generations by 41% over IDA*, but ran 64% slower per node [Korf, 1993]. An implementation of MA* [Chakrabarti *et al.*, 1989] on the Fifteen Puzzle ran 20 times as slow as IDA*, making it impractical for solving randomly generated problem instances [Korf, 1993].

The creation of the FSM table is an efficient use of time and space. Some millions of nodes were generated in the breadth-first search. Tens of thousands of duplicate strings were found, and these were encoded in a table with some tens of thousands of states. However, as reported above, this led to the elimination of billions of node generations on the harder problems.

In addition to the experiments finding optimal solutions, weighted iterative-deepening A* (WIDA*) was also tested [Korf, 1993]. This is an iterative-deepening search, with a modified evaluation function, $f(x) = g(x) + wh(x)$, where $g(x)$ is length of the path from the root to node x, $h(x)$ is the heuristic estimate of the length of a path from node x to the goal, and w is the weight factor [Pohl, 1970]. Higher weighting allows suboptimal solutions to be found faster. This is the expected effect of relaxing the optimality criteria.

A second effect that beyond a certain value, increasing w increased the number of nodes generated, rather than decreasing it [Korf, 1993]. At $w = 3.0$, for instance, an average of only 59,000 nodes were generated. However, above $w = 7.33$, the number of nodes generated again increased. For a run at $w = 19.0$, WIDA* without pruning generates an average of 1.2 million nodes for each of 100 random puzzle instances. With FSM pruning, the average number of nodes generated is reduced to 5,590, a reduction of 99.4%. The results reported here support the hypothesis that this effect is caused by duplication of nodes through the combinatorial rise of the number of alternative paths to each node.

The Twenty-Four Puzzle

The FSM method was also employed on the Twenty-Four Puzzle. To date, no optimal solution has been found for a randomly generated Twenty-Four Puzzle instance. The exploration phase generated strings up to 13 operators long. A set of 4,201 duplicates strings was created, which produced a FSM with 15,745 states, with a table implementation of 63,000 words.

Weighted Iterative-Deepening A* (WIDA*) was applied to 10 random Twenty-Four Puzzle instances. Previously, average solution lengths of 168 moves (with 1000 problems, but at $w = 3.0$), were the shortest solutions found to that time [Korf, 1993]. With FSM duplicate pruning in WIDA*, the first ten of these problems yielded solutions at $w = 1.50$ weighting. They have an average solution length of 115, and generated an average of 1.66 billion nodes each. These solutions were found using the Manhattan Distance plus linear conflict heuristic function [Hansson *et al.*, 1992], as well as FSM pruning. Without pruning, time limitations would have been exceeded.

The effectiveness of duplicate elimination can be measured at $w = 3.0$ with and without FSM pruning. With Manhattan Distance WIDA* heuristic search, an average of 393,000 nodes each were generated for 100 random puzzle instances. With Manhattan Distance WIDA* plus FSM pruning, an average of only 22,600 were generated, a savings of 94.23%.

Rubik's Cube

For the 2x2x2 Rubik's cube, one corner may be regarded as being fixed, with each of the other cubies participating in the rotations of three free faces. Thus, there are nine operators. The space was explored to depth seven, where 31,999 duplicate strings were discovered. An FSM was created from this set which had 24,954 states. All of the trivial optimizations were discovered automatically as strings of length two. In a brute-force search, there would be a branching factor of 9. Eliminating the inverse operators and consecutive moves of the same face, this is reduced to 6. With the FSM pruning based on a learning phase to depth seven, a branching factor of 4.73 was obtained.

For the full 3x3x3 cube, each of six faces may be rotated either Right, Left, or 180 degrees. This makes a total of 18 operators, which are always applicable. The space was explored to depth 6, where 28,210 duplicate strings were discovered. An FSM was created from this set which had 22,974 states. All of the trivial optimizations were discovered automatically as strings of length two. A number of interesting duplicates were discovered at depths 4 and 5 representing cycles of length 8. For the Rubik's cube without any pruning rules, the branching factor is 18 in a brute-force depth-first search (no heuristic). By eliminating inverse operators, moves of the same face twice in a row, and half of the consecutive moves of non-intersecting faces, the branching factor for depth first search is 13.50. With FSM pruning based on a learning phase to depth seven, a branching factor of 13.26 was obtained.

Related Work

Learning duplicate operator sequences can be compared to the learning of concepts in explanation-based learning (EBL) [Minton, 1990]. These techniques share an exploratory phase of learning, capturing information from a small search which will be used in a larger one. The purpose of these operator sequences, however, is not accomplishment of specific goals, but the avoidance of duplicate nodes. In machine learning terms, we are learning only one class of control information, i.e., control of redundancy [Minton, 1990]. We are learning nothing about success or failure of goals, or about goal interference, at which EBL techniques are directed. The introduction of macros into a search usually means the loss of optimality and an increase in the branching factor; eliminating duplicate sequences preserves optimality, and reduces the branching factor.

EBL-aided searches may be applied to general sets of operators, while the FSM technique of this paper is limited to domains which have sets of operators that may be applied at any point.

Several EBL techniques have used finite state automata for automatic proof generation [Cohen, 1987], or have used similar structures for the compaction of applicability checking for macros [Shavlik, 1990].

Conclusions

We have presented a technique for reducing the number of duplicate nodes generated by a depth-first search. The FSM method begins with a breadth-first search to identify operator strings that produce duplicate nodes. These redundant strings are then used to automatically generate a finite state machine that recognizes and rejects the duplicate strings. The FSM is then used to generate operators in the depth-first search. Producing the FSM is a preprocessing step that does not affect the complexity of the depth-first search. The additional time overhead to use the FSM in the depth-first search is negligible, although the FSM requires memory proportional to the number of states in the machine. This technique reduces the asymptotic complexity of depth-first search on a grid from $O(3^r)$ to $O(r^2)$. On the Fifteen Puzzle, it reduces the brute force branching factor from 2.13 to 1.98, and reduced the time of an IDA* search by 70%. On the Twenty-Four Puzzle, a similar FSM reduced the time of WIDA* by 94.23%. It reduces the branching factor of the 2x2x2 Rubik's Cube from 6 to 4.73, and for the 3x3x3 Cube from 13.50 to 13.26.

References

Aho, A. V.; Sethi, R.; and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

Chakrabarti, P. P.; Ghose, S.; Acharya, A.; and de Sarkar, S. C. 1989. Heuristic search in restricted memory. *Artificial Intelligence* 41:197–221.

Cohen, W. W. 1987. A technique for generalizing number in explanation-based learning. Technical Report ML-TR-19, Computer Science Department, Rutgers University, New Brunswick, NJ.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Dillenburg, J. F. and Nelson, P. C. 1993. Improving the efficiency of depth-first search by cycle elimination. *Information Processing Letters* (forthcoming).

Elkan, C. 1989. Conspiracy numbers and caching for searching and/or trees and theorem-proving. In *Proceedings of IJCAI-89.* 1:341–346.

Hansson, O.; Mayer, A.; and Yung, M. 1992. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences* 63(3):207–227.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Ibaraki, T. 1978. Depth_m search in branch and bound algorithms. *International Journal of Computer and Information Science* 7:315–343.

Korf, R. E. 1985. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence.* To appear.

Minton, S. 1990. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence* 42(2–3):363–91.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence.* Morgan Kaufman Publishers, Inc., Palo Alto, Calif.

Pearl, J. 1984. *Heuristics.* Addison-Wesley, Reading, Mass.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1:193–204.

Sen, A. K. and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of IJCAI-89.* 1:297–302.

Shavlik, Jude W. 1990. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning* 5:39–70.

Taylor, Larry A. 1992. Pruning duplicate nodes in depth-first search. Technical Report CSD-920049, UCLA Computer Science Department, Los Angeles, CA 90024-1596.