Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

PRUNING DUPLICATE NODES IN DEPTH-FIRST SEARCH

Larry A. Taylor

October 1992
CSD-920049

Pruning Duplicate Nodes in Depth-First Search

Larry A. Taylor

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024
ltaylor@cs.ucla.edu

October 16, 1992

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 technique, 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 of the depth-first search is reduced, reducing the asymptotic complexity of the search.

Contents

1 Introduction—The Problem

Search techniques are fundamental to artificial intelligence. The success of an application often depends on implementing a search through a problem space with limitations on memory and time. Most often, there is a choice of operators to apply to a node, and the number of nodes grows exponentially.

Best-first searches, including breadth-first search, Dijkstra's algorithm, and A*, all depend on enough memory to store all generated nodes at the same time. This gives the whole class of algorithms exponential space complexity, making them impractical for many problems.

In contrast, depth-first searches, including iterative-deepening, 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 [11, 12, 9]. 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 on a problem may be orders of magnitude more than the number of nodes generated by a best-first search.

To illustrate, consider a search of a grid with the following 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 in fact only $O(r^2)$ distinct junctions are visited by a breadth-first search. Thus, a depth-first search for this problem has exponential complexity, while a breadth-first search has only polynomial complexity.

To reduce this effect, we would like to find a way to detect and prune duplicates 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.

Short of storing all the nodes, two partial solutions have been suggested [12]. In depth-first search, the path from the root to the current node is available. The current node can be compared to nodes on the path. 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 small fraction of duplicates can be found by comparing nodes on the current path.

Another solution is to store some subset of the nodes. Potential duplicate

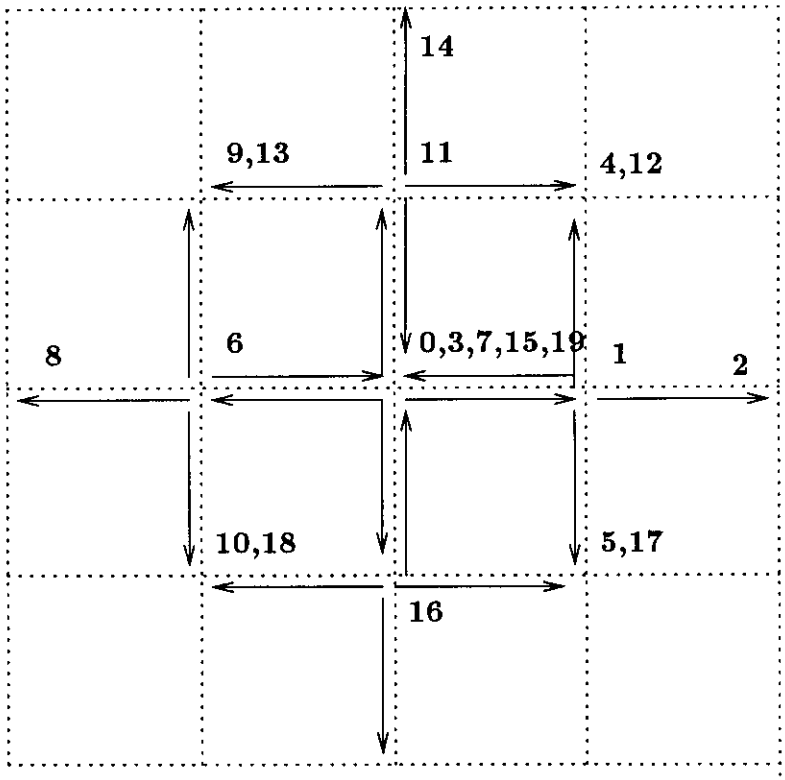

Figure 1: The grid search space, explored depth-first to depth 2.

checking can be performed against the stored list of nodes. However, this method can still detect only a fraction of all duplicates; the deeper the search path, the smaller the ratio of all potential duplicates is detected.

Depth-m search by Ibaraki [8] maintains an ordered list of next nodes to expand (best-first search), but limits the growth of the list. If a limit, m, is exceeded, the node list switches to depth-first exploration of new nodes. As a result, the space requirement is linear in the depth of the solution, instead of exponential. At the limit, the nodes available for duplicate checking in depth-m search are those of the current solution path, plus short branches from that path.

Starting with a best-first search, the first nodes encountered may be stored, and then depth-first search may proceed from the set of stored nodes, checking each node to see if it has already been generated. The MREC algorithm [13] executes best-first search until memory is almost full, then performs IDA* below the stored frontier nodes. The memory is statically allocated to the first nodes generated. In heuristic search, new nodes generated will have better heuristic values as the goal is approached, so that the first nodes generated may not match nodes deeper in the search. On the other hand, finding duplicates within the first nodes generated means that they have been found high in the overall tree, which will increase the overall savings.

Alternatively, we may save a variable set of nodes, replacing old nodes with new ones as the search proceeds. The MA* algorithm [4] dynamically stores the best nodes generated so far, pruning nodes of the highest cost to maintain a maximum number of nodes in memory.

We will show 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 generating in the search's past, and nodes that will be generated in the future. This technique uses limited storage efficiently, uses only a constant time per node searched, and which reduces the effective branching factor, so that the reduction in duplicates pruned increases with the depth of the search.

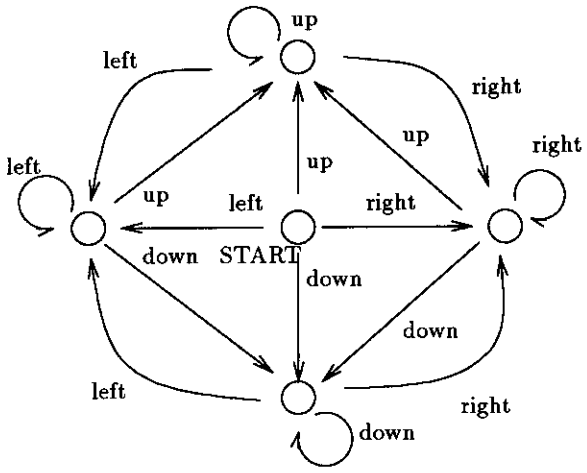

Figure 2: FSM eliminating inverse operators.

2 The FSM Pruning Rule Mechanism

2.1 Exploiting Structure

We can take advantage of the fact that most combinatorial problems are actually described implicitly. If a problem space is too large to be stored as an explicit graph, then the problem space can only 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 example, in the grid, the operator sequence Left-Right will always produce a node that has already been examined in a depth-first search. Rejecting inverse operator pairs, including in addition Right-Left, Up-Down, and Down-Up, reduces the branching factor by exactly one. The complexity is thus reduced from $O(4^r)$ to $O(3^r)$. Most depth-first search implementations already use this and similar optimizations, but we carry the principle further.

Inverse operators can be eliminated by a finite state machine (FSM) as shown in figure 2. 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 new node, change the state of the machine based on the new operator applied to the old state. For the machine in figure 2, the valid transitions are given by the arrows which specify the possible next operators that may be applied. Operators that are the inverse of the last operator applied do not appear. This prunes all subtrees below such redundant nodes. The time cost of this optimization is negligible.

Carrying the example 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 only once in a depth-first search to depth r: 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 3 shows a search to depth two carried out with these rules. Figure 4 shows an FSM that implements this search strategy. The search now has time complexity $O(r^2)$, reducing the complexity from exponential to quadratic.

The set of pruning rules we have just examined depend on the fact that some paths will always generate duplicates of nodes generated by other paths. These relations between paths can be discovered and characterized in different domains. In the following section, a method is presented for automatically learning a finite state machine that encodes such pruning rules from a description of the problem.

2.2 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. Then, the resulting set of strings is used to create the FSM.

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 figures 1, 5). 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 representations to make matches, and use the cost of the operator strings to make the choices between the pairs. Ties are broken

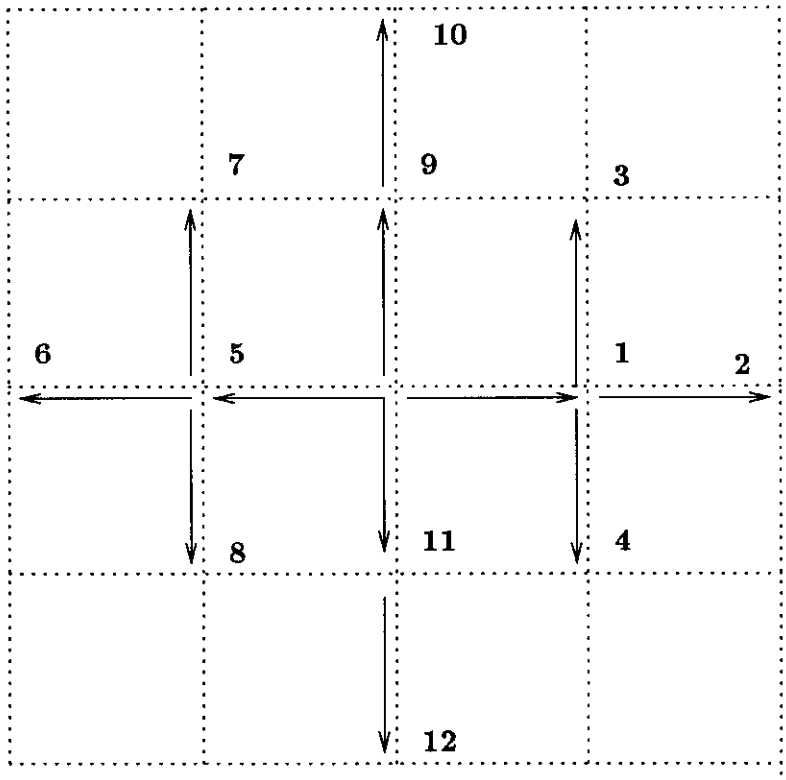

Figure 3: Grid search space, explored depth-first to depth 2, with pruning.

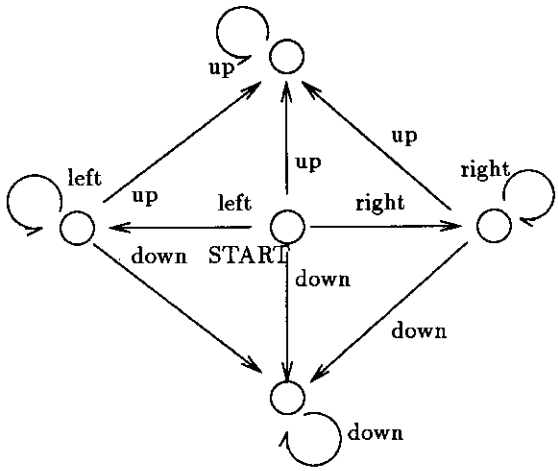

Figure 4: FSM corresponding to the search in figure 3.

arbitrarily but systematically, such as alphabetically. 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. In other words, 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. The other path contains precisely the other half of the matched pair of strings.

The 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 no duplicate string is a substring of any other.

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. 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 requirement for the breadth-first does not apply during the actual search itself. The actual depth, d, employed will depend on the actual constant space available.

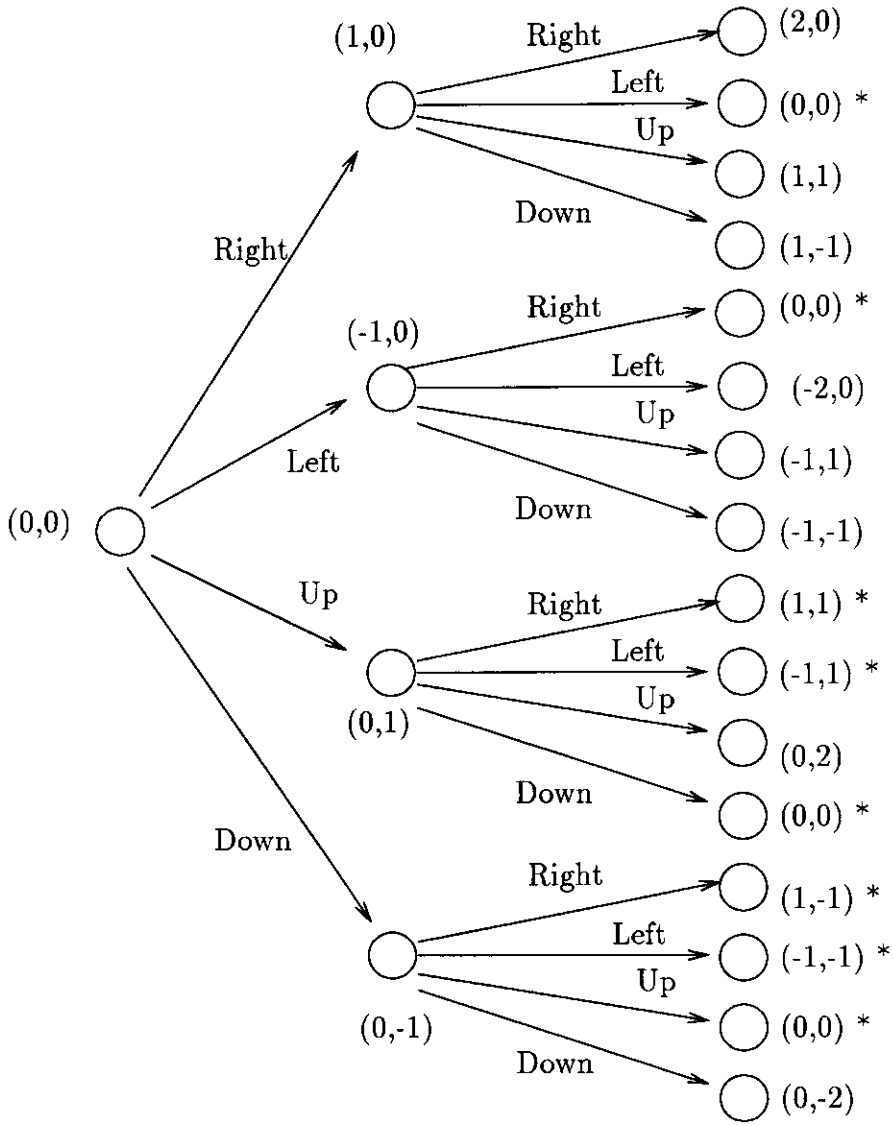

Figure 5: Searching the grid produces many duplicates, even at depth 2.

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 pays off for solutions of multiple problem instances. The results presented below give examples of such savings.

This set of 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. The reason is that another sequence of operators that have not been eliminated will reach the same nodes. 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).

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 [1, 2, 3]. In this algorithm, a *trie* (a transition diagram in which each state corresponds to a prefix of a keyword [3]) is constructed from the set of keywords (in this case, the duplicate operator strings). The trie is a tree in which the strings with shared prefixes are gathered together at the root, creating branches as each sequence differs from the others.

This bare skeleton represents a recognition machine for matching keywords that start at the beginning of the 'text' string. Keywords that begin somewhere in the middle of the string must also be recognized. The machine will follow along until it reaches an operator (symbol) that forces it to depart from the skeleton. The machine must be returned to some point in the trie. To calculate this point, the skeleton is filled in with a failure transition function. This function represents transitions for strings that mismatch (fail) the keyword paths of the trie. 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. In the construction of the skeleton, and the calculation of the rest of the transitions, the time and space required is at most $O(l)$, where l is the sum of the lengths of the keywords.

The task of the algorithm is straightforward, but we wish to clarify some points. We do not need to recognize a general set of regular expressions, but only those that recognize a set of fixed keywords in a text (the bibliographic

search problem). We have chosen a straightforward method that is well analyzed, efficient and popular (being used in the Unix utility command *fgrep* and in other places).

Naturally, we wish to prune a search at the earliest point possible. In a depth-first search, this means at the highest point in the tree. In terms of operator paths, we want to detect a duplicate on the shortest possible string. This consideration influences the choice of keyword recognition algorithms. In keyword recognition terms, we can use no symbols 'to the right'. We cannot consider substring recognition algorithms that depend on 'lookahead' symbols.

A trie constructed from the duplicate string pairs from the grid example is shown in figure 6. A machine for recognizing the grid space duplicate string set is shown in figure 4. 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. Figure 4 and figure 6 represent the same machine, but are complementary. Figure 4 is the full base FSM with the rejecting states and edges of figure 6 subtracted.

2.3 Using the FSM

Incorporating a FSM into a depth-first search is efficient in time and memory used. 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. Results for several domains are given below.

2.4 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

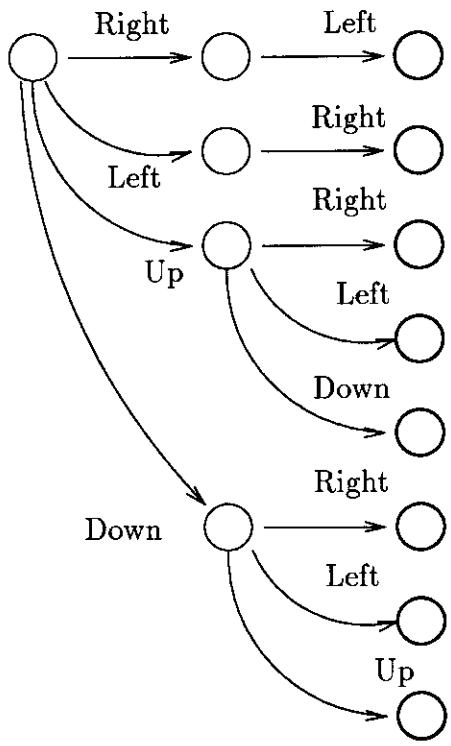

Figure 6: A trie data structure recognizing the duplicate operator strings found in the grid example. The heavy circles represent rejecting (absorbing) states.

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.

We can prove that following these rules means that 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 some circumstances, condition (3) can be slightly relaxed. If the goal is broadly defined, then more than one goal node may exist. Condition (3) would then require that A and B be equivalent with respect to the goal condition, even if the nodes created are not identical. Implicit or partially specified goals will have some relevant and some irrelevant portions of their representations.

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 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 [5] to generate the duplicate operator strings, instead of a breadth-first search.

2.5 Operator Preconditions

The FSM pruning method in the form used here depends on treating every node in the problem space alike. All operators are assumed to be applicable at all times. However, the Fifteen Puzzle (figure 7) contains a significant boundary condition. For example, when the blank is in the upper left corner, moving the blank Left or Up is not valid.

Given a particular position for the blank square, some operator strings are valid and some are not. This is the "operator precondition" problem, and we must solve it for a particular search space to use the FSM pruning. To restate the problem, some operator strings that produce matching nodes, starting from the same beginning node, may have different preconditions. For example, the strings of figure 8 will produce the same node if executed with the blank in a center square of the Fifteen Puzzle, but not if the blank starts at the right edge. The direction indicates the movement of the blank. String B accomplishes in two columns what string A needs three columns to do. Therefore A is not applicable in all situations that B is applicable, so B cannot be considered a duplicate string for A.

	1	2
3	4	5
6	7	8

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure 7: The Eight Puzzle, the Fifteen Puzzle, and the Twenty-Four Puzzle, each in a goal configuration. All moves are made by swapping the blank tile with the contents of an adjacent tile. The solution to a tile puzzle configuration consists of a sequence of moves to a goal configuration.

A : Up-Right-Down-Down-Left-Up-Left-Down-Right-Right-Up-Up-Left
B : Left-Down-Right-Up-Up-Left-Down-Right-Down-Left-Up-Up-Right

Figure 8: Two operator strings for the Fifteen Puzzle that differ in necessary preconditions.

For the Fifteen Puzzle, we deal with the problem in two steps. In order to create a generalized beginning position, it was necessary to explore the "Forty-eight Puzzle" (figure 9). That is, the blank was put at the center of a 7x7 array for the start of the search. Any sequence of moves possible on the Fifteen Puzzle board (which is 4x4) is possible starting from the center of the Forty-eight Puzzle board, regardless of where the blank starts in the Fifteen Puzzle.

The second part of the problem is to guarantee that given all possible blank starting positions in the Fifteen Puzzle, no string of a pair is pruned as a duplicate if the original string is not valid. To deal with this situation, a routine was written to test the "bounding box" of the actions of a pair of strings, A and B. B can be a duplicate if it is a match of A, and A has a bounding box contained within or identical to that of B. In this implementation, all such B strings are deleted from the candidate duplicate set, so that only operator strings that are applicable at all nodes remain.

By eliminating some possible duplicates, we preserve the correctness and optimality of the solution, while lowering the power of duplicate detection. In the Twenty-Four Puzzle, only sixteen duplicates were eliminated by this

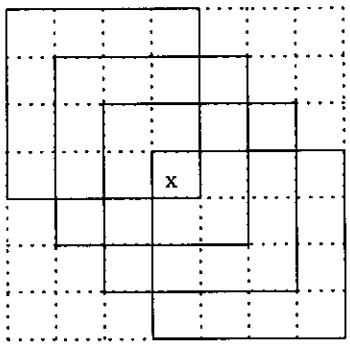

Figure 9: How to represent all possible starting positions of the blank in the Fifteen Puzzle.

test, all of length thirteen. These duplicate eliminations could be recovered if blank position were incorporated into the FSM. In this case, reaching a rejection state would be conditioned not only on the operator string, but also on the position of the blank. Other domains may have more complicated preconditions.

3 Experimental Results

3.1 The Fifteen Puzzle

Positive results were obtained using the FSM method combined with a Manhattan Distance heuristic in searching for solutions of random Fifteen Puzzle instances. Heuristic functions, including the Manhattan Distance, are estimates of remaining costs to the goal of a problem. The Manhattan Distance is computed from the sum of the costs of moves of all the individual tiles needed to separately move them to their goal positions. For the tile puzzles, this is the sum of absolute values of the differences, in columns and rows, of the location for each tile and the location of that tile in the goal configuration. This "city block" value is a relaxation that ignores interactions. Heuristic functions that never overestimate the costs guarantee an optimal solution.

The Fifteen Puzzle was explored breadth-first to a depth of 14 in search-

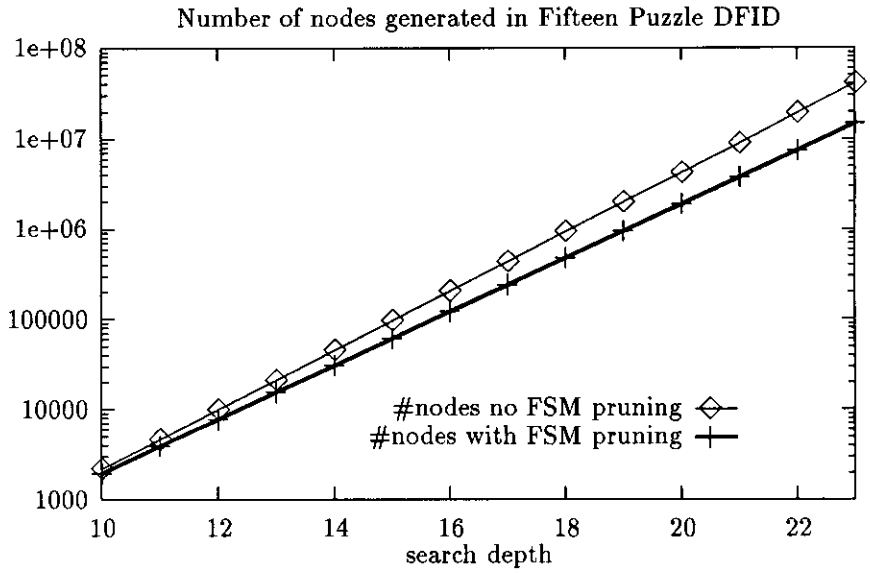

Figure 10: Nodes generated in the Fifteen Puzzle, with and without pruning with an FSM table explored to depth 14.

ing 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 thousand words. The inverse operators were found automatically at depth 2. The thousands of other duplicate strings discovered represent other non-trivial cycles.

The branching factor in a brute-force search was tested (figure 10). The branching factor 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 effective branching factor decreased as the depth of the discovery search was 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.

Iterative-deepening A* using the Manhattan Distance heuristic was applied to the 100 random Fifteen Puzzle instances used in [9]. In iterative-deepening A*, depth-first expansion is bounded by a cost function on the

nodes, $f(x) = g(x) + h(x)$. The total cost estimate of any node x, $f(x)$, is the sum of the costs incurred to reach the node, $g(x)$, and the estimated costs to the goal, $h(x)$. 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 other techniques which use extra memory to save nodes. 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. Korf [10] used MREC [13] storing 100,000 nodes, reducing node generations by 41% over IDA*, but running 64% slower per node. Korf's implementation [10] of MA* [4] on the Fifteen Puzzle ran 20 times as slow as IDA*, making it impractical to use for solving randomly generated problem instances.

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. Furthermore, the majority of the pruning benefit reported occurs with the shorter strings (found early in the exploration phase), and could be implemented with a FSM with 45 to a few hundred states.

In addition to the experiments finding optimal solutions, weighted iterative-deepening A* (WIDA*) was employed [10]. In WIDA*, the cost function f is altered by placing higher weights on anticipated future costs than already incurred path costs; $f(x) = g(x) + wh(x)$, where $w >= 1$. The search weighting is characterized by the proportion of the weight given to h. $w = 1.0$ weighting guarantees an optimal solution. Higher weighting allows suboptimal solutions to be found faster. This is the expected effect of relaxing the optimality criteria.

A second effect, which I call "duplicate bloom," is the increase in the number of nodes generated as the solution length gets longer. The two competing effects are illustrated in the results found by Korf [10]. The average optimum solution length for the test problems was 53.05, while an average of 359 million nodes were generated for each random Fifteen puzzle. The weighted heuristic functions found longer solutions while generating fewer nodes as the weight on h was increased. Korf found that WIDA* produced fewer nodes

as the weighting increased in favor of h, as it should for all related heuristic search methods (figure 11). At $w = 3.0$ weighting, for instance, an average of only 59 thousand nodes were generated, for average solution lengths of 98.23. 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 a solution length of 580.93.

With FSM pruning, the average nodes generated for the same set at $w = 3.0$ is reduced to 29 thousand nodes and an average solution length of 94.69, while at $w = 19.0$, the average nodes is reduced to 5,590, a reduction of 99.4%, and the solution length is 573.71. The FSM pruning method appears to effectively eliminate the "duplicate bloom" effect on long solutions. In figure 11, WIDA* with FSM pruning is shown to generate fewer nodes that WA*. For an iterative-deepening depth-first search, the largest number of nodes are generated in the last layer. The exact number of nodes generated can be regarded as a random number between 1 and the maximum size of this last layer, based on the effective branching factor, because the the search stops at the first solution. Because of this, WIDA* sometimes generates fewer nodes than WA*, which is asymptotically optimal in the number of nodes generated.

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.

3.2 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 FSM pruning method has allowed us to produce the shortest solutions ever found, through a great reduction in the number of duplicate nodes generated compared to IDA* without FSM. A generalized beginning position, similar to that used for the Fifteen Puzzle above, was generated by using a 9x9 array ("Eighty Puzzle" board) with the blank starting in the middle. The exploration phase generated strings up to 13 operators long. A set of 4,201 duplicate strings was created, which produced a FSM with 15,745 states. The table implementation of the FSM used 63 thousand words.

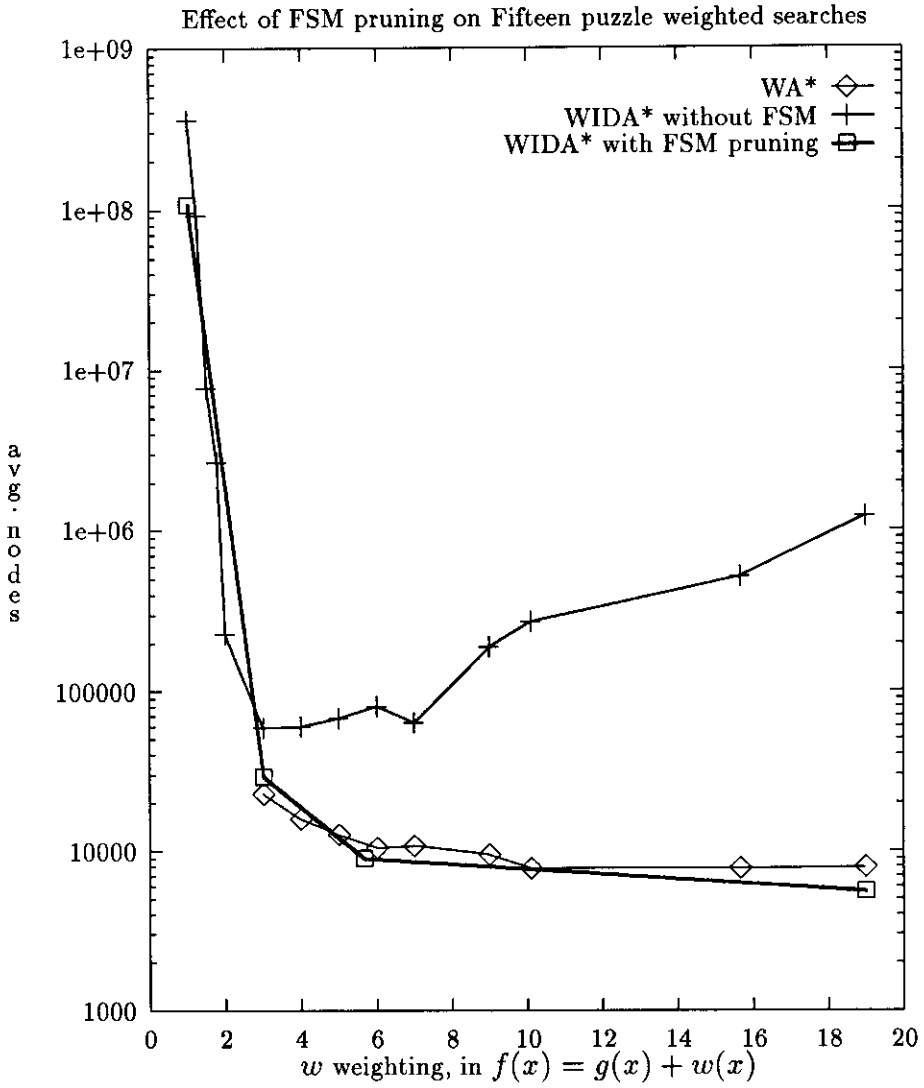

Figure 11: Duplicate bloom: Average nodes generated per problem using weighted heuristic search on the Fifteen puzzle: WIDA* without pruning, WIDA* with pruning, and Weighted A* (WA*).

Weighted Iterative-Deepening A* (WIDA*) was applied to 10 random Twenty-Four Puzzle instances. Korf [10] was able to achieve average solution lengths of 168 moves (with 1000 problems, but at $w = 3.0$), which is believed to be the shortest solutions found to that time. With FSM duplicate pruning in WIDA*, the first ten problems in Korf's list have 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. They are the shortest solutions ever found for the Twenty-Four Puzzle. These solutions were found using the Manhattan Distance plus linear conflict heuristic function [7], as well as FSM pruning. Linear conflict is a known way to boost the Manhattan distance heuristic by counting tiles that may be the right row or column, but in the wrong order. A minimum number of moves may be found that are necessary to reorder each row and column, but are not counted in the Manhattan Distance heuristic. With Manhattan Distance plus linear conflict, many more nodes are pruned while preserving admissibility (optimal solutions).

The effectiveness of duplicate elimination can be measured at $w = 3.0$ weighting with and without FSM pruning. With Manhattan Distance (only) WIDA* heuristic search, an average of 393 thousand nodes each were generated for 100 random puzzle instances, with an average solution length of 219.10. With Manhattan Distance WIDA* plus FSM pruning, an average of only 22.6 thousand nodes were generated, a savings of 94.23%. The average solution length was 217.32.

This node generation reduction of over an order of magnitude leads to hope that an optimal solution to a randomly generated Twenty-Four Puzzle instance may be found soon. Experiment shows that the FSM pruning technique usually finds solutions of slightly lower cost under given conditions as well. The reason for this may be the pruning of high-cost paths as part of the duplicate elimination.

A comparison was made to test the effect of pruning on problems with shorter solution paths. 1,000 random Eight Puzzle instances were solved using the Manhattan Distance heuristic, both with FSM pruning and without FSM pruning, but with the inverse operators rejected. For the Eight Puzzle, only 33.81% of nodes were saved by FSM pruning. This supports the conjecture that FSM pruning is more effective for larger problems.

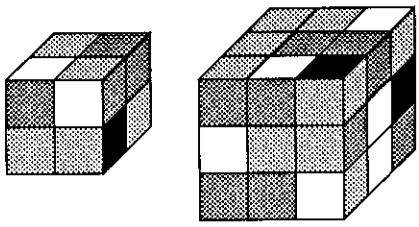

Figure 12: Rubik's cubes, 2x2x2 and 3x3x3.

3.3 Rubik's Cube

The Rubik's Cube puzzle (figure 12) consists of a subdivided cube with brightly colored tiled faces. The goal is to get faces to show only one color on each of the six sides of the cube. A move is a twist of one of the faces of the cube. The cubies are the small cubes which show on the faces, and whose rotations form a large group. [6] describes a standard notation and discusses the mathematics of the 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 7, 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. For instance, LL^{-1} is a one-quarter clockwise twist of the Left face of the cube, followed by the inverse operator, a one-quarter counter-clockwise twist of the same face.

In a brute-force search, there would be a branching factor of 9.0. Eliminating the inverse operators and consecutive moves of the same face, this is reduced to 6.0. 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 eighteen operators, which are always applicable. For the large cube, it is assumed that all faces rotate (so that no corner is fixed), but that the center cubies of each face are fixed. Move sequences such as LR and RL produce duplicates, because the operators R and L move faces that do not intersect.

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 18.00 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 the FSM pruning based on a learning phase to depth seven, a branching factor of 13.26 was obtained.

4 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 pre-processing 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.

5 Acknowlegements

Thanks to my advisor, Richard Korf, for overall direction and editing, and for use of programs and data. This research was partially supported by NSF

Grant #IRI-9119825, and a grant from Rockwell International.

References

[1] Alfred V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986. The Dragon Book.

[4] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41:197–221, (1989/90).

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

[6] Alexander H. Frey, Jr. and David Singmaster. *Handbook of Cubik Math.* Enslow Publishers, Hillside, New Jersey, 1982.

[7] Othar Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, September 15 1992.

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

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

[10] Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 1992. to appear.

[11] Nils J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufman Publishers, Inc., Palo Alto, Calif., 1980.

[12] Judea Pearl. *Heuristics.* Addison-Wesley Publishing Company, Reading, Mass., 1984.

[13] Anup K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1:297–302, 1989. (IJCAI-89).