Relevance Cuts: Localizing the Search

Andreas Junghanns, Jonathan Schaeffer

Department of Computing Science
University of Alberta
Edmonton, Alberta
CANADA T6G 2H1
{andreas, jonathan}@cs.ualberta.ca

Abstract. Humans can effectively navigate through large search spaces, enabling them to solve problems with daunting complexity. This is largely due to an ability to successfully distinguish between relevant and irrelevant actions (moves). In this paper we present a new single-agent search pruning technique that is based on a move's *influence*. The influence measure is a crude form of relevance in that it is used to differentiate between local (relevant) moves and non-local (not relevant) moves, with respect to the sequence of moves leading up to the current state. Our pruning technique uses the m previous moves to decide if a move is relevant in the current context and, if not, to cut it off. This technique results in a large reduction in the search effort required to solve Sokoban problems.
Keywords: single-agent search, heuristic search, Sokoban, local search, IDA*

1 Introduction and Motivation

It is commonly acknowledged that the human's ability to successfully navigate through large search spaces is due to their meta-level reasoning [4]. The relevance of different actions when composing a plan is an important notion in that process. Each next action is viewed as one logically following in a series of steps to accomplish a (sub-)goal. An action judged as irrelevant is not considered.

When searching small search spaces, the computer's speed in base-level reasoning can effectively overcome the lack of meta-level reasoning by simply enumerating large portions of the search space. However, it is a trivial matter to pose a problem to the computer that is easy for a human to solve (using reasoning) but is exponentially large to solve using standard search algorithms. We need to enhance computer algorithms to be able to reason at the meta-level if they are to successfully tackle these larger search tasks. In the world of computer games (two-player search), a number of meta-level reasoning algorithmic enhancements are well known, such as null-move searches [5] and futility cut-offs [11]. For single-agent search, macro moves [9] are an example.

In this paper, we introduce *relevance cuts*. The search is restricted in the way it chooses its next action. Only actions that are relevant to previous actions

can be performed, with a limited number of exceptions being allowed. The exact definition of relevance is domain dependent.

Consider an artist drawing a picture of a wildlife scene. One way of drawing the picture is to draw the bear, then the lake, then the mountains, and finally the vegetation. An alternate way is to draw a small part of the bear, then draw a part of the mountains, draw a single plant, work on the bear again, another plant, maybe a bit of lake, etc. The former corresponds to how a human would draw the picture: concentrate on an identifiable component and work on it until a desired level of completeness has been achieved. The latter corresponds to a typical computer method: the order in which the lines are drawn does not matter, as long as the final result is achieved.

Unfortunately, most search algorithms do not follow the human example. At each node in the search, the algorithm will consider all legal moves regardless of their relevance to the preceding play. For example, in chess, consider a passed "a" pawn and a passed "h" pawn. The human will analyze the sequence of moves to, say, push the "a" pawn to queen. The computer will consider dubious (but legal) lines such as push the "a" pawn one square, push the "h" pawn one square, push the "a" pawn one square, etc. Clearly, considering alternatives like this is not cost-effective.

What is missing in the above examples is a notion of *relevance*. In the chess example, having pushed the "a" pawn and then decided to push the "h" pawn, it seems silly to now return to considering the "a" pawn. If it really was necessary to push the "a" pawn a second time, why weren't both "a" pawn moves considered *before* switching to the "h" pawn? Usually this switching back and forth (or "ping-ponging") does not make sense but, of course, exceptions can be constructed.

In other well-studied single-agent search domains, such as the N-puzzle and Rubik's Cube, the notion of relevance is not important. In both these problems, the geographic space of moves is limited, i.e. all legal moves in one position are "close" (or local) to each other. For two-player games, the effect of a move may be global in scope and therefore moves almost always influence each other (this is most prominent in Othello, and less so in chess). In contrast, a move in the game of Go is almost always local. In non-trivial, real-world problems, the geographic space might be large, allowing for local and non-local moves.

This paper introduces relevance cuts and demonstrates their effectiveness in the one-player game Sokoban. For Sokoban we use a new influence metric that reflects the structure of the maze. A move is considered relevant if it is influencing all the previous m moves made. The search is only allowed to make relevant moves with respect to previous moves and only a limited number of exceptions is permitted. With these restrictions in place, the search is forced to spend its effort locally, since random jumps within the search area are discouraged. In the meta-reasoning sense, forcing the program to consider local moves is making it adopt a pseudo-plan; an exception corresponds to a decision to change plans. This results in a decrease of the average branching factor of the search tree.

For our Sokoban program *Rolling Stone*, relevance cuts result in a large reduction of the search space. These reductions are on top of an already highly efficient[1] searcher. On a standard set of 90 test problems, relevance cuts allow *Rolling Stone* to increase the number of problems it can solve from 39 to 44. Given that the problems increase exponentially in difficulty, this relatively small increase in the number of problems solved represents a large increase in search efficiency.

2 Sokoban and Related Work

Single-agent search (A*) has been extensively studied in the literature. There are a plethora of enhancements to the basic algorithm, allowing the application developer to customize their implementation. The result is an impressive reduction in the search effort required to solve challenging applications (see [10] for a recent example). However, the applications used to illustrate the advances in single-agent search efficiency are "easy" in the sense that they have some (or all) of the following properties:

1. effective, inexpensive lower-bound estimators,
2. small branching factor in the search tree, and
3. moderate solution lengths.

The sliding-tile puzzles are the best known examples of these problems. Problem domains such as these also have the important property that given a solvable starting state, every move preserves the solvability (although not necessarily the optimality).

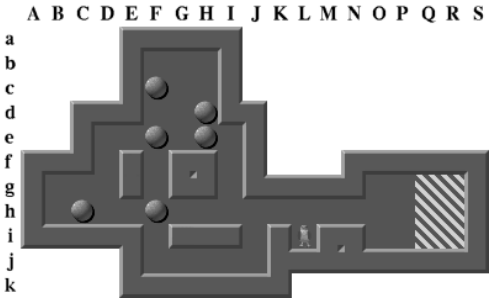
Sokoban is a popular one-player computer game. The game originated in Japan, although the original author is unknown. The game's appeal comes from the simplicity of the rules and the intellectual challenge offered by deceptively easy problems.

Figure 1 shows a sample Sokoban problem.[2] The playing area consists of rooms and passageways, laid out on a rectangular grid of size 20x20 or less. Littered throughout the playing area are *stones* (shown as circular discs) and *goals* (shaded squares). There is a *man* whose job it is to move each stone to a goal square. The man can only push one stone at a time and must push from behind the stone. A square can only be occupied by one of a wall, stone or man at any time. Getting all the stones to the goal squares can be quite challenging; doing this in the minimum number of moves is much more difficult.

To refer to squares in a Sokoban problem, we use a coordinate notation. The horizontal axis is labeled from "A" to "T", and the vertical axis from "a" to "t"

[1] Of course, "highly efficient" here is meant in terms of a computer program. Humans shake their heads in disbelief when they see some of the ridiculous lines of play considered in the search.

[2] This is problem 1 of the standard 90-problem suite available at http://xsokoban.lcs.mit.edu/xsokoban.html.



He-Ge Hd-Hc-Hd Fe-Ff-Fg Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Rh-Rg Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Qi-Ri Fc-Fd-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Qg Ge-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Rh Hd-He-Ge-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Pi-Qi Ch-Dh-Eh-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh

Fig. 1. Sokoban Problem 1 With One Solution

(assuming the maximum sized 20x20 problems), starting in the upper left corner. A move consists of pushing a stone from one square to another. For example, in Figure 1 the move *Fh-Eh* moves the stone on *Fh* left one square. We use *Fh-Eh-Dh* to indicate a sequence of pushes of the same stone. A move, of course, is only legal if there is a valid path by which the man can move behind the stone and push it. Thus, although we only indicate stone moves (such as *Fh-Eh*), implicit in this is the man's moves from its current position to the appropriate square to do the push (for *Fh-Eh* the man would have to move from *Li* to *Gh* via the squares *Lh*, *Kh*, *Jh*, *Ih* and *Hh*).

Unlike most single-agent search problems studied in the literature, a single Sokoban move can change a problem from being solvable to unsolvable. For example, in Figure 1, making the move *Fh-Fg* creates an unsolvable problem. It requires a non-trivial analysis to verify this deadlock. This is a simple example, since deadlock configurations can be large and span the entire board. Identifying deadlock is critical to prevent a lot of futile searching.

The standard 90 problems range from easy (such as problem 1 above) to difficult (requiring hundreds of stone pushes). A global score file is maintained showing who has solved which problems and how efficient their solution is (also at http://xsokoban.lcs.mit.edu/xsokoban.html). Thus solving a problem is only part of the satisfaction; improving on your solution is equally important.

Sokoban has been shown to be PSPACE-complete [2,3]. Dor and Zwick show that the game is an instance of a motion planning problem, and compare the game to other motion planning problems in the literature [3]. For example, Sokoban is similar to Wilfong's work with movable obstacles, where the man is allowed to hold on to the obstacle and move with it, as if they were one object

[12]. Sokoban can be compared to the problem of having a robot in a warehouse move a number of specified goods from their current location to their final destination, subject to the topology of the warehouse and any obstacles in the way. When viewed in this context, Sokoban is an excellent example of using a game as an experimental test-bed for mainstream research in artificial intelligence.

Sokoban is a difficult problem domain for computers because of the following reasons:

1. it has a complex lower-bound estimator ($O(n^3)$, given n goals),
2. the branching factor is large and variable (potentially over 100),
3. the solution may be very long (some problems require over 500 moves to solve optimally),
4. the search space complexity is $O(10^{98})$ for problems restricted to a 20x20 area only, and
5. some reachable states are unsolvable (deadlock).

For sliding-tile puzzles, there are algorithms for generating a non-optimal solution. In Sokoban, because of the presence of deadlock, often it is very difficult to find *any* solution.

Our previous attempts to solve Sokoban problems using standard single-agent search techniques are reported in [7]. There, using our program *Rolling Stone*, we compare the different techniques and their usefulness with respect to the search efficiency when solving Sokoban problems. IDA* [8] was augmented with a sophisticated lower bound estimator, transposition tables, move ordering, macro moves and deadlock tables. Even though each of the standard single-agent search enhancements we investigated resulted in significant improvements (often several orders of magnitude in search-tree size reduction), at the time we were able to solve only 20 problems of a 90-problem test suite.

In [6] we introduced a new search enhancement, *pattern searches*, a method that dynamically finds deadlocks and improved lower bounds. Since a single move can introduce a deadlock, before playing a move we perform a *pattern search* to analyze if deadlock will be introduced by that move. The pattern search attempts to identify the conditions for a deadlock and, if all the conditions are satisfied, saves a pattern of stones that is the minimal board configuration required for the deadlock. During the IDA* search, a new position can be matched with these patterns to see if it contains a deadlock. As a side benefit, these pattern searches can also identify arbitrary increases to the lower bound (e.g. a deadlock increases the lower bound to ∞).

The notion of bit (stone) patterns is similar to the Method of Analogies [1]. Pattern searches are a conflict-driven top-down proof of correctness, while the Method of Analogies is a bottom-up heuristic approximation.

Pattern searches allow us to now solve 39 of the 90 problems [6][3]. Although pattern searches can be enhanced to make them more efficient, we concluded that they are inadequate to successfully solve all 90 Sokoban test positions.

[3] Note that [6] reports slightly different numbers than this paper, caused by subsequent refinements to the pattern searches and bug fixes.

Even with all the enhancements, and the cumulative improvements of several orders of magnitude in search efficiency, the search trees are still too deep and the effective branching factor too high. Hence, we need to find further ways to improve the search efficiency.

3 Relevance Cuts

Analyzing the trees built by an IDA* search quickly reveals that the search algorithm considers move sequences that no human would ever consider. Even completely unrelated moves are tested in every legal combination – all in an effort to prove that there is no solution for the current threshold. How can a program mimic an "understanding" of relevance? We suggest that a reasonable approximation of relevance is influence. If two moves are not influencing each other then they are very unlikely to be relevant to each other. If a program had a good "sense" of influence, it could assume that in a given position all previous moves belong to a (unknown) plan of which a continuation can only be a move that is relevant – in our approximation, is influencing whatever was played previously.

Thus, the general idea for relevance cuts is to prevent the program from trying all possible move sequences. Moves tried have to be relevant to previously executed moves. This can be achieved in different, domain specific, ways. The following shows one implementation for the domain of Sokoban. Even though the specifics aren't necessarily applicable to other domains, the basic philosophy of the approach is.

3.1 Influence

When judging how two squares in a Sokoban maze are influencing each other, Euclidean distance is not adequate. Taking the structure of the maze into account would lead to a simple geographic distance which is still not proportional with influence. For example, consider two squares connected by a tunnel; the squares are equally influencing each other, no matter how long the tunnel is. Figure 1 shows several tunnels of which one consists of the squares *Ff* and *Fg*. Prolonging the tunnel without changing the general topology of the problem would change the geographic distance, but not the influence.

The following is a list of properties we would like the influence measure to reflect:

Alternatives: The more alternatives that exist on a path between two squares, the less they influence each other. That is, squares in the middle of a room where stones can go in all 4 directions should decrease influence more than squares in a tunnel, where no alternatives exist.

Goal-Skew: Squares on the optimal path to any goal should have stronger influence than squares off the optimal path.

Connection: Two neighboring squares connected such that a stone can move between them should influence each other more than two squares connected such that only the man can move between them.

Tunnel: In a tunnel, influence remains the same: It does not matter how long the tunnel is (one could, for example, collapse a tunnel into one square).

Our first implementation of relevance cuts used small off-line searches to statically precalculate a (20x20)x(20x20) table containing the influence values for each square of the maze to every other square in the maze. Between every pair of squares, a breadth-first search is used to find the path(s) with the largest influence. The algorithm is similar to a shortest-path finding algorithm, except that we are using influence here and not geographic distance. The smaller the influence number, the more two squares are influencing each other.

Note that influence is not necessarily symmetric ($dist(a,b) \neq dist(b,a)$). A square close to a goal influences squares further away more than it is influenced by them. Furthermore, $dist(a,a)$ is not necessarily 0. A square in the middle of a room will be less influenced by each of its many neighbors than a square in a tunnel. To reflect that, squares in the middle of a room receive a larger bias than more restricted squares.

The exact numbers used in our implementation are the following (with the name of the wish-list item following in parenthesis). Each square on the path between the start and goal squares adds 2 for each direction (off the path considered) a stone can be pushed and 1 for each direction the man can go. Thus, the maximum one square can add for alternatives is 4 (alternatives). However, every square that is part of an optimal path towards any of the goals from the start square will add only half of that amount (goal-skew). If the connection from the previous square on the path to the current squares can be taken by a stone only 1 is added, else 2 (connection). If the previous square is in a tunnel, 0 is added (tunnel), regardless of all other properties.

3.2 Relevance Cut Rules

Given the above influence measure, we can now proceed to explain how to use that information to cut down on the number of moves considered in each position. To do this, we need to define *distant moves*. Given two moves, *m1.from-m1.to* and *m2.from-m2.to*, move $m2$ is distant with respect to move $m1$ if the from squares of the moves ($m1.from$ and $m2.from$) do not influence each other. More precisely, two moves influence each other if

$$InfluenceTable[\ m1.from\ ][\ m2.from\ ]\ <\ d$$

where *InfluenceTable* is the table of precalculated values and d is a tunable threshold.

Relevance cuts eliminate some moves that are distant from the previous moves played, and therefore are considered not relevant to the search. There are two ways that a move can be cut off:

1. If within the last m moves more than k distant moves were made. This cut will discourage arbitrary switches between non-related areas of the maze.
2. A move that is distant with respect to the previous move, but not distant to a move in the past m moves. This will not allow switches back into an area previously worked on and abandoned just briefly.

In our experiments, we set k to 1. This way, the first cut criterion will entail the second. The parameters d and m are set according to the following properties of the maze. The maximal influence distance, d, is set to half the average influence value from all squares to the squares on optimal paths to any goal, but not less than 6. The length of history used, m, is set to the average influence value of all squares to all other non-dead squares in the maze, but not less than 10.

3.3 Example

Figure 2 shows an example where humans immediately identify that solving this problem involves solving two separate sub-problems. Solving the left and right side of the problem is completely independent. An optimal solution needs 82 moves; *Rolling Stone*'s lower bound estimator returns a value of 70. Standard IDA* will need 7 iterations to find a solution (our lower-bound estimator preserves the odd/even parity of the solution length). In each of the iterations but the last, IDA* will try every possible (legal) move combination with moves from both sides of the problem. This way IDA* proves for each of the 6 iterations i that the problem cannot be solved with $70+2*i$ moves, regardless of the order of the considered moves. Clearly, this is unnecessary and inefficient. Solving one of the sub-problems requires only 4 iterations, since the lower bound is off by only 6. Considering this position as two separate problems will result in an enormous reduction in the search complexity.

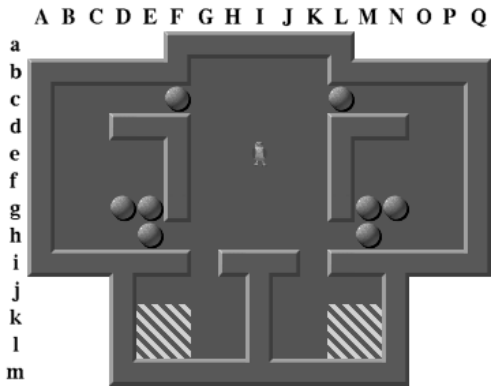


Fig. 2. Example Maze With Locality

Our implementation considers all moves on the left and on the right side as distant from each other. This way only a limited number of switches is considered during the search. Our parameter settings allow for only one non-local move per 9-move sequence. For this contrived problem, relevance cuts decrease the number of nodes searched from 32,803 nodes to 24,748 nodes while still returning an optimal solution (the pattern searches were turned off for simplicity). Although this is a significant reduction, it is only a small step towards achieving all the possible savings. For example, one of the sub-problems can be solved by itself in only 329 nodes! The difference between 329 and 32,803 illustrates why IDA* in its current form is inadequate for solving large, non-trivial real-world problems. Clearly, more sophisticated methods are needed.

3.4 Discussion

Further refinement of the parameters used are certainly possible and necessary if the full potential of relevance cuts is to be achieved. Some ideas with regards to this issue will be discussed in the future work section.

The overhead of the relevance cuts is negligible, at least for our current implementation. The influence of two moves can be established by a simple table lookup. This is in stark contrast to our pattern searches, where the overhead dominates the cost of the search for most problems.

4 Experimental Results

Rolling Stone has been tested using the 90-problem test set using searches limited to 20,000,000 nodes. Our previous best version of *Rolling Stone* was capable of solving 39 of the test problems. With the addition of relevance cuts, the number of problems solved has increased to 44[4]. Table 1 shows a comparison of *Rolling Stone* with and without relevance cuts for each of the 44 solved problems.

For each program version in Table 1, the third column gives the number of IDA* iterations that the program took to solve the problem. Note that problems #9, #21 and #51 are now solved non-optimally, taking at least one iteration longer than the program without relevance cuts. This confirms the unsafe nature of the relevance cuts. However, since none of the problems solved before is lost and 5 more are solved, the gamble paid off. Long ago we abandoned our original goal of obtaining optimal solutions to Sokoban problems. The size of the search space dictates radical pruning measures if we want to have any chance of solving some of the tougher problems.

Of the 5 new problems solved, #11 is of interest. Without relevance cuts, only 17 IDA* iterations could be completed within our pre-set limit of 20,000,000 nodes. Relevance cuts allow *Rolling Stone* to search 19 iterations and solve the

[4] Note that we "cheat" with problem #46, as we allow it to go 47,000 nodes beyond the 20 million node limit. A bug fix pushed it beyond the 20 million limit and we wanted it to count in the statistics. We tested all the unsolved problems without the relevance cuts to 50 million nodes and no other problem was solved.

#	without relevance cuts			with relevance cuts		
	top level nodes	total nodes	# iterations	top level nodes	total nodes	# iterations
1	32	270	2	32	270	2
2	200	3,251	2	177	2,764	2
3	392	10,486	2	301	10,395	2
4	394	10,556	1	392	10,554	1
5	5,999	121,502	3	6,079	152,082	3
6	170	1,593	3	151	1,574	3
7	12,378	156,334	5	6,821	68,202	5
8	152,919	3,066,098	6	89,838	1,806,540	6
9	11,572	234,454	5	14,963	307,006	8
10	834,147	16,678,800	4	340,935	6,815,512	4
11	> 998,299	> 20,000,000	17	337,143	6,759,590	19
17	1,160	14,891	7	1,250	14,740	7
19	890,100	17,829,863	9	88,575	1,814,505	9
21	38,371	765,392	9	47,854	970,776	10
34	> 1,651,897	> 20,000,000	9	227,525	4,495,028	9
38	333,257	844,882	42	236,351	748,024	42
40	> 998,236	> 20,000,000	7	311,618	6,239,163	8
43	112,610	2,270,703	8	60,120	1,215,022	8
45	729,333	14,646,623	9	250,157	5,059,596	9
46	> 2,022,198	> 20,000,000	12	1,004,325	20,047,197	15
49	> 17,074,823	> 20,000,000	11	2,303,495	3,047,672	13
51	725	2,611	1	2,049	18,368	2
53	182	3,737	1	185	3,740	1
54	283,609	4,381,171	9	255,827	3,884,844	9
55	1,696,996	3,194,830	3	603,190	1,349,908	3
56	4,318	34,429	6	3,817	31,388	6
57	61,900	1,084,732	5	45,339	797,766	5
60	5,929	116,103	3	1,252	24,403	3
62	2,720	71,578	5	1,984	46,534	5
63	10,195	197,922	3	21,131	390,422	3
64	194,846	3,900,639	10	32,706	652,857	10
65	364	12,971	5	364	12,971	5
67	239,515	2,177,787	13	160,936	2,103,866	13
68	128,716	2,651,559	11	16,814	355,306	11
70	841,495	15,003,603	3	94,792	1,949,842	3
72	1,908	43,260	5	3,168	72,647	5
73	11,371	247,816	3	14,791	292,799	3
78	75	809	1	75	783	1
79	362	4,017	5	200	3,512	5
80	805	15,513	1	2,081	48,220	1
81	1,251	40,806	4	1,074	29,698	4
82	8,500	181,571	5	4,643	97,406	5
83	635	15,423	1	389	13,106	1
84	272,160	521,068	4	153,220	443,508	4
	> 29,637,064	>190,559,653		6,748,129	72,210,106	

Table 1. Experimental Data

problem. Given that the cost of an extra iteration is large (and can typically be a factor of about 5,000 *per iteration* [6]), a gain of 2 iterations represents a massive improvement.

The tree size for each program version given in Table 1 is broken into two numbers. *Top-level nodes* refers to that portion of the search tree that IDA* is applied to. *Total nodes* includes the top-level nodes and the pattern search nodes. Clearly, for some problems (such as #45) the cost of performing pattern searches overwhelms the search effort, whereas in other problems (such as #53) they are a small investment. Further details on pattern searches and when they are executed can be found in [6].

The magnitude of the top-level nodes can be misleading; superficially it looks like these problems can be "trivially" solved with few nodes. Using standard IDA* with our sophisticated lower bound estimator fails to solve *any* of the 90 test problems within our limit of 20,000,000 nodes. Consequently, we added a plethora of enhancements to the program, including transposition tables, macro moves, move ordering and deadlock tables, *each* of which is capable of reducing the search tree size by one or more orders of magnitude [7]! Thus the small top-level node counts reported in the table are the result of extensive improvements to the search algorithm.

Relevance cuts reduce the number of top-level nodes by at least a factor of 4.5. Note that since the program not using relevance cuts cannot solve 5 problems, this factor may be a gross underestimation of the actual impact.

With respect to the total search nodes, relevance cuts improve search efficiency by almost a factor of three. Again, this is a lower bound. In particular, problem #11 still requires an enormous amount of search, given that it still has 2 iterations to go before it can find the solution.

Comparing node numbers of individual searches is difficult because of many volatile factors in the search. For example, a relevance cut might eliminate a branch from the search justifiably, but a pattern search there would have uncovered valuable information that would have been useful for reducing the search in other parts of the tree. Problem #80 is one such example: despite the relevance cuts the node count goes up from 99 to 123 nodes; an important discovery was not made and the rest of the search increases. However, the overall trend is in favor of the relevance cuts. An excellent example is problem #70: the top level node count is cut down to 3,006 nodes and a solution is found. Previously 579,037 nodes were considered without finding a solution.

Figures 3 and 4 plot the amount of effort to solve a problem, using the numbers from Table 1 sorted by total nodes. An additional data point is given with a curve that shows what the program's performance was with all the standard single-agent search techniques implemented, before pattern searches where added. Figure 3 shows the impact of the relevance cuts. The exponential growth in difficulty with each additional problem solved is dampened, allowing for more problems solved with the same number of nodes. Figure 4 is a logarithmic representation of Figure 3. The figure more clearly shows that up to about the 25th problem (ordered according to number of nodes needed to solve) there is very

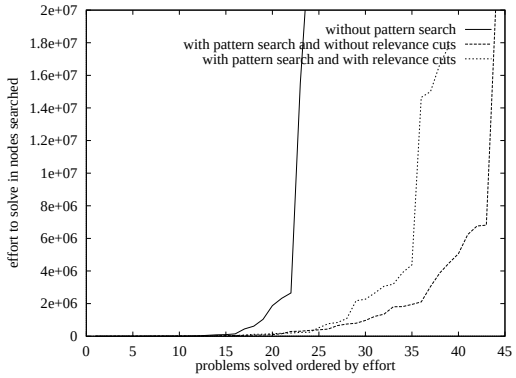


Fig. 3. The Effect of Relevance Cuts

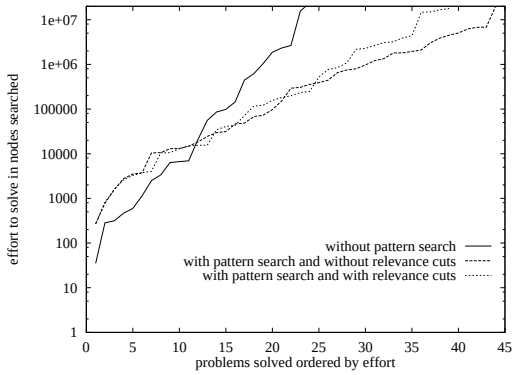


Fig. 4. The Effect of Relevance Cuts (Log Scale)

little difference in effort required; the relevance cuts do not save significant portions of the small search trees. However, with larger search trees, the success of relevance cuts gets more pronounced.

5 Conclusions and Future Work

Relevance cuts provide a crude approximation of human-like problem-solving methods by forcing the search to favor local moves over global moves. This simple idea provides large reductions in the search tree size, at the expense of possibly returning a longer solution. Given the breadth and depth of Sokoban search trees, finding optimal solutions is a secondary consideration; finding *any* solution is challenging enough.

There are several ideas on how to improve the effectiveness of relevance cuts.

- Use different distances depending on crowding. If many stones are crowding an area, it is likely that the relevant area is larger than it would be with less stones blocking each other.
- The current influence measure can most likely be improved. A thorough investigation of all the parameters used could lead to substantial improvements.
- There are several parameters used in the relevance cuts. The setting of those is already dependent of properties of the maze. These parameters are critical for the performance of the cuts and are also largely responsible for increased solution lengths. More research on those details is needed to fully exploit the possibilities relevance cuts are offering.
- So far, *Rolling Stone* is painting locally, but is not yet "object oriented". If a flower and the bear are close, painting both at the same time is very likely. Better methods are needed to further understand subgoals, rather than localizing by area.

Although relevance cuts introduce non-optimality, this is not an issue. Once humans solve a Sokoban problem, they have two choices: move on to another problem (they are satisfied with the result), or try and re-solve the same problem to get a better solution. *Rolling Stone* could try something similar. Having solved the problem once, if we want a better solution, we can reduce the probability of introducing non-optimality in the search by decreasing the aggressiveness of the relevance cuts. This will make the searches larger but, on the other hand, the last iteration does not have to be searched, since a solution for that threshold was already found.

Relevance cuts are yet another way to significantly prune Sokoban search trees. We have no shortage of promising ideas, each of which potentially offers another order of magnitude reduction in the search tree size. Although this sounds impressive, our experience suggests that each factor of 10 improvement seems to only yield another 4 or 5 problems being solved. At this rate, we will have do a lot of research if we want to successfully solve all 90 problems!

6 Acknowledgements

The authors would like to thank the German Academic Exchange Service, the Killam Foundation and the Natural Sciences and Engineering Research Council of Canada for their support.

References

1. G. Adelson-Velskiy, V. Arlazarov, and M. Donskoy. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6(4):361–371, 1975.
2. J. Culberson. Sokoban is PSPACE-complete. Technical Report TR97-02, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1997. ftp://ftp.cs.ualberta.ca/pub/TechReports/1997/TR97-02.

3. D. Dor and U. Zwick. SOKOBAN and other motion planning problems, 1995. At: http://www.math.tau.ac.il/~ddorit.
4. M. Ginsberg. *Essentials in Artificial Intelligence*. Morgan Kaufman Publishers, San Francisco, 1993.
5. G. Goetsch and M.S. Campbell. Experiments with the null-move heuristic. In T.A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 159–181, New York, 1990. Springer-Verlag.
6. A. Junghanns and J. Schaeffer. Single-agent search in the presence of deadlock. In *AAAI-98*, pages 419–424, Madison/WI, USA, July 1998.
7. A. Junghanns and J. Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In *Advances in Artificial Intelligence*, pages 1–15. Springer Verlag, 1998.
8. R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
9. R.E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.
10. R.E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI–97*, pages 700–705, 1997.
11. J. Schaeffer. *Experiments in Search and Knowledge*. PhD thesis, Univ. of Waterloo, Canada, 1986.
12. G. Wilfong. Motion planning in the presence of movable obstacles. In *4th ACM Symposium on Computational Geometry*, pages 279–288, 1988.