Theoretical Computer Science 252 (2001) 151–175

Theoretical Computer Science

www.elsevier.com/locate/tcs

Sokoban: improving the search with relevance cuts ☆

Andreas Junghanns, Jonathan Schaeffer *

Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1

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 (irrelevant) 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. © 2001 Elsevier Science B.V. All rights reserved.

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 at base-level reasoning can effectively overcome the lack of meta-level reasoning by simply enumerating large portions of the search space. However, it is easy to identify a problem that is simple for a human to solve (using reasoning) but is exponentially large for a computer to

☆ This is a revised and enhanced version of an earlier paper A. Junghanns, J. Schaeffer, Relevance cuts: localizing the search, in: H.J. van den Herik, H. Iida (Eds.), Computers and Games, Lecture Notes in Computer Science, Vol. 1558, Springer, Berlin, 1999, pages 1–14.

* Corresponding author.

E-mail addresses: andreas@cs.ualberta.ca (A. Junghanns), jonathan@cs.ualberta.ca (J. Schaeffer).

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 [15]. For single-agent search, macro moves [11] are an example.

In this paper, we introduce *relevance cuts*, a meta-level reasoning enhancement for single-agent search. 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 application-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 forward 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 moves with local and non-local implications.

This paper introduces relevance cuts and demonstrates their effectiveness in the one-player maze-solving puzzle of Sokoban. For Sokoban we use a new influence metric that reflects the structure of the maze. A move is considered relevant only if the previous m moves influence it. The search is only allowed to make relevant moves with respect to previous moves and only a limited number of exceptions are 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.

The search tree size, and thus the search effort expended in solving a problem, depends on the depth of the search tree and the effective branching factor. Relevance cuts aim at reducing the effective branching factor. 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 searcher. [1] On a standard set of 90 test problems, relevance cuts allow *Rolling Stone* to increase the number of problems it can solve from 40 to 44. [2] 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

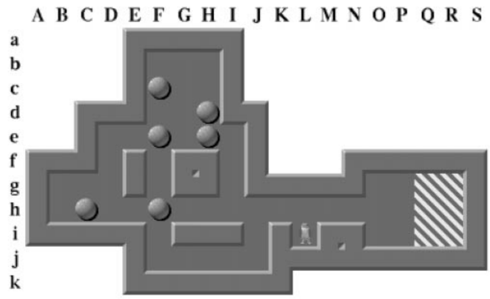
Sokoban is a popular one-player computer game. The game originated in Japan, and was apparently developed by *Thinking Rabbit* in 1982. The game's appeal comes from the simplicity of the rules and the intellectual challenge offered by deceptively easy problems.

Fig. 1 shows a sample Sokoban problem, the easiest instance of the standard 90-problem suite available at http://xsokoban.lcs.mit.edu/xsokoban.html. The playing area consists of rooms and passageways, laid out on a rectangular grid of size 20×20 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 at any time by one of a wall, stone or man. Getting all the stones to the goal squares can be quite challenging; doing this with the minimum number of man moves and/or stone pushes 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*" (assuming the maximum sized 20×20 problems), starting in the upper left corner. A move consists of pushing a stone from one square to another. For example, in Fig. 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 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

[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] Subsequent to this work, our best Sokoban solver was improved and can now solve 57 problems [17].



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 and a solution.

current position to the appropriate square to the stone push (for *Fh–Eh* the man would have to move from *Li* to *Gh* via the squares *Lh*, *Kh*, *Jh*, *Ih* and *Hh*).

The solution given in Fig. 1 is optimal with respect to the number of stone pushes. One could also solve the problem to identify the minimal number of man moves. In general, a single solution does not minimize both metrics. In this work, we are trying to minimize the number of stone pushes.

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. These human solutions also serve as excellent upper bounds on optimal solution lengths.

2.1. The computational challenge of Sokoban

Single-agent search (A*) has been extensively studied in the literature. There is 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 [12] 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,
3. moderate solution lengths, and
4. all moves are reversible.

Sokoban is a difficult problem domain for computers, and more challenging than previously studied domains, because of the following reasons:

1. Sokoban has a complex lower-bound estimator (O(N^3), given N goals). Unfortunately, even this expensive lower bound is not very effective. In other domains, such as the sliding tile puzzles or Rubik's Cube, a table lookup is often sufficient to deliver a high-quality lower bound.
2. The branching factor for Sokoban is large and variable (potentially over 100), whereas the 15-puzzle has an effective branching factor of roughly 2 and in Rubik's Cube it is less than 14 [12].
3. The solution to a Sokoban problem may be very long (some problems require over 600 moves to solve optimally). In contrast, the maximum solution length for the 15-puzzle is 80, while it is a mere 18 moves for Rubik's Cube.
4. The search space complexity for Sokoban is O(10^{98}) for problems restricted to a 20×20 area. Since the 15-puzzle and Rubik's Cube have both a smaller branching factor and smaller solution lengths, their search space complexity is considerably less (O(10^{13}) and O(10^{19}), respectively).
5. Sokoban problems require "sequential" solutions. Many of the subgoals interact, making it difficult to divide the problem into independent subgoals.
6. All moves are reversible in the 15-puzzle and Rubik's Cube. This is not true in Sokoban, where some moves are not (or not directly) reversible.

The graph underlying a Sokoban problem is directed, unlike the traditional single-agent search applications which have undirected graphs. Furthermore, there are parts of the graph that do not contain solutions. Both these properties are necessary and sufficient conditions for the presence of deadlocks – states that have no solution. In Fig. 1, pushing the stone *Fh–Fg* creates an unsolvable problem. It requires a non-trivial analysis to verify this deadlock. This is a small example, since deadlock configurations can be large and span the entire board. Identifying deadlock is critical to prevent futile searching.

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

2.2. Related work

Sokoban has been shown to be PSPACE-complete [1, 3]. Dor and Zwick show that the game is an instance of a motion planning problem, and compare it 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 [16]. 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.

2.3. Rolling Stone

Our previous attempts to solve Sokoban problems using standard single-agent search techniques are reported in [7]. There, using our program *Rolling Stone*, we compared the different techniques and their effect on search efficiency. The following is a list of the major components of our program.

*IDA**: *Rolling Stone* uses the iterative-deepening A* (IDA*) search algorithm [10]. A* is potentially exponential in space. IDA* trades off the space requirements for time. A* and IDA* use an admissible lower bound on the distance from any state to the goal state. IDA* iterates by searching for a solution of a particular length. If one is found, the search is finished. Otherwise, the solution length is incremented and the search is repeated. The lower-bound estimator allows large portions of the search tree to be eliminated by proving that a solution cannot be found in the requisite number of moves.

Lower bound: To obtain an admissible estimate of the distance of a position to a goal, the Minimum-Cost, Perfect Bipartite Matching algorithm is used [13]. The matching assigns each stone to a goal and returns the total (minimum) distance of all stones to their goals. The algorithm is $O(N^3)$ in the number of stones N. IDA* with this lower bound cannot solve any of the test problems given one billion search nodes.

Transposition table: The search space of Sokoban is a graph, rather than a tree, implying that repeated positions and cycles are possible. A *transposition table* was implemented to avoid duplicate search effort. The transposition table maps positions using the exact stone locations and equivalent man locations taking man reachability into account. Using this enhancement can reduce the search tree size by several orders of magnitude.

Move ordering: The program orders the children of a node based on their likelihood of leading to a solution. *Move ordering* may reduce the search tree only in the last iteration.

Deadlock table: Pattern databases [2] are a recent idea that has been used successfully in domains like $(N \times N)$-puzzles and Rubik's Cube [12]. An off-line search enumerated all possible stone/wall placements in a 4×5 region and searched them to determine if deadlock was present. These results were stored in *deadlock tables*. During an IDA* search, the table is queried to see if the current move leads to a local deadlock. Thus, deadlock tables contain search results of partial problem configurations and are general with respect to all Sokoban problems. They reduce the effective branching factor by eliminating moves that provably lead to local deadlocks.

Tunnel macros: A Sokoban maze often contains "tunnels" (such as the squares *Kh*, *Lh*, *Mh* and *Nh* in Fig. 1). Once a stone is pushed into a tunnel, it must eventually be pushed all the way through. Rather than search to discover this over and over again, this sequence of moves can be collapsed into a single macro move. By collapsing several moves into one, the height of the search tree is reduced. *Tunnel macros* are identified by pre-processing.

Goal macros: Prior to starting the search, a preliminary search is used to find an appropriate order in which to fill in the goal squares. In many cases this is a non-trivial computation, especially when the goal area(s) has several entrances. A specialized search is used to avoid fill sequences that lead to deadlocks in the goal area. The knowledge about the goal area is then used to create goal macros, where stones are pushed directly from the goal area entrance(s) to their final goal square avoiding deadlocks. For example, in Fig. 1, square *Gh* is defined as the entrance to the goal area; once a stone reaches it, a single macro move is used to push it to the next pre-determined goal square. These macro moves significantly reduce the search depth required to solve problems and can dramatically reduce the search tree size. Whenever a goal macro move is possible, it is the only move considered; all alternatives are *forward pruned.*

Goal cuts: *Goal cuts* effectively push the goal macros further up the search tree. Whenever a stone can be pushed to a goal entrance square, none of the alternative moves are considered. The idea behind these cuts is that if one is confident about using macro moves, one might as well prune alternatives to pushing that stone further up in the search tree.

Pattern search: *Pattern searches* [6] are an effective way to detect lower-bound inefficiencies. Small, localized conflict-driven searches uncover patterns of stones that interact in such a way that the lower-bound estimator is off by an arbitrary amount (even infinite, in the case of a deadlock). These patterns are used throughout the search to improve the lower bound. Patterns are specific to a particular problem instance and are discovered on the fly using specialized searches. Patterns represent the knowledge about dynamic stone interactions that lead to poor static lower bounds, and the associated penalties are the corrective measures.

2.4. Conclusions

The net effect of combining all of the above enhancements results in 40 of the 90 problems being solved [6].[3] Pattern searches are the most expensive component, but also the most beneficial. Although they can be enhanced and made more efficient, we concluded that this would still be inadequate to successfully solve all 90 Sokoban test positions. 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

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

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 do not influence each other, then it is unlikely that they are 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.

A move is considered relevant only if the previous *m* moves influence it. The search is only allowed to make relevant moves with respect to previous moves, and only a few exceptions are 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.

3.1. Influence

An influence metric can be achieved in different, domain-specific ways. The following shows one implementation for Sokoban. Even though the specifics are not necessarily applicable to other domains, the basic philosophy of the approach is.

We approximate the influence of two moves on each other by the influence between the move’s *from* squares. The influence between two squares is determined using the notion of a “most influential path” between the squares. This can be thought of as a least-cost path, except that influence is used as the cost metric.

When judging how two squares in a Sokoban maze influence each other, Euclidean distance is not adequate. Taking the structure of the maze into account would lead to a simple geographic distance which is not proportional to influence either. For example, consider two squares connected by a tunnel; the squares are equally influencing each other, no matter how long the tunnel is. Elongating 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 the squares 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. See Fig. 2 for an example. Squares *A* and *B* are influencing one another less than squares *C* and *D*. There are many more possible ways to get from *A* to *B*, whereas squares *C* and *D* are more restricted because they are situated on a wall.

Goal-skew: For a given square *sq*, any squares on the optimal path from *sq* to a goal should have stronger influence than squares off the optimal path. For example, in Fig. 3 square *B* is influenced by *C* more than it is by *A*. The location of the goals is important.

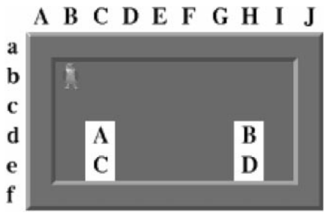


Fig. 2. The number of alternatives changes the influence.

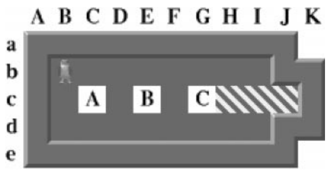


Fig. 3. The location of the goals matters.

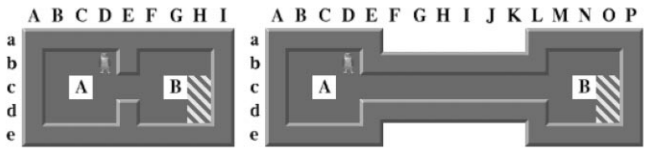


Fig. 4. Tunnels and influence.

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. In Fig. 2, square A influences C less than C influences A, because stones can only move towards C, and not towards A.

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). Fig. 4 shows such an example: two problem mazes that are identical, except for the length of the tunnel. Influence values should not change because of the length of the tunnel.

Our implementation of relevance cuts uses small off-line searches to statically pre-calculate a $(20 \times 20) \times (20 \times 20)$ table (*InfluenceTable*) 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 influence each other.

Note that influence is not necessarily symmetric

$$\mathit{InfluenceTable}[a,b] \neq \mathit{InfluenceTable}[b,a].$$

A square close to a goal influences squares further away more than it is influenced by them. Furthermore, *InfluenceTable*$[a, a]$ 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.

Our approach is quite simple and can undoubtably be improved. For example, influence is statically computed. A dynamic measure, one that takes into account the current positions of the stones, would be more effective.

Our implementation runs a shortest-path finding algorithm to find the largest influence between any pair of squares. The first is referred to as the *start* square; the second as the *destination* square. Each square on a path between the start and destination squares contributes points depending on how it influences that path. The more points associated with a pair of squares, the less the squares influence each other. The exact numbers used to calculate influence are the following:

Alternatives: A square s on a path will have two neighboring squares that are *not* on the path. For each of the neighboring squares, n, the following points are added: 2 points if it is possible to push a stone (if present) from s to n; 1 point if it is only possible to move a man from s to n; and 0 if n is a wall. Thus, the maximum number of points that one square can contribute for alternatives is 4.

Goal-skew: However, if s is on an optimal path from the start square to any of the goals in the maze, then the alternatives points are divided by two.

Connection: The connection between consecutive squares along a path is used to modify the influence. If a stone can be pushed in the direction of the destination square, then 1 point is added. If only the man can traverse the connection between the squares (moving towards the destination square), then 2 points are added.

Tunnel: If the previous square on a path is in a tunnel, 0 points are added, regardless of the above properties.

Fig. 5 is used to illustrate influence. For a subset of squares in the figure, Table 1 shows the influence numbers. In this example, the program automatically determines

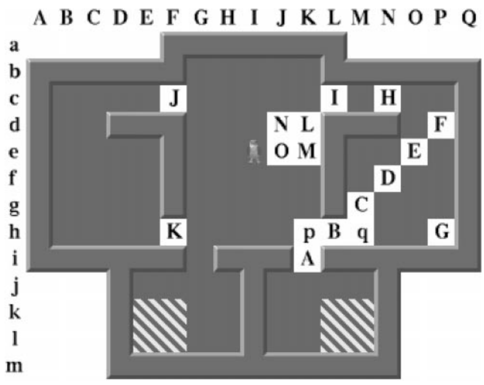

A B C D E F G H I J K L M N O P Q
a b c d e f g h i j k l m
J I H N L F O M E D C K p B q G A

Fig. 5. Example squares.

Table 1
Example influence values

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*	*O*
A	1	6	10	18	19	21	13	17	17	24	12	12	10	18	16
B	4	1	5	13	14	18	8	16	16	22	10	11	9	17	15
C	7	4	1	9	10	15	9	15	15	25	13	14	12	20	18
D	11	8	5	3	9	14	12	14	14	29	17	18	16	24	22
E	13	10	7	7	2	7	9	7	7	26	19	12	14	18	20
F	23	19	17	18	10	2	13	6	6	25	34	11	13	17	19
G	12	7	9	15	14	11	1	15	15	34	23	19	17	25	23
H	10	10	14	17	9	5	16	1	1	11	16	3	4	7	8
I	10	10	14	17	9	5	16	1	1	11	16	3	4	7	8
J	16	16	20	27	19	15	23	11	11	1	10	11	12	11	14
K	10	10	14	22	23	26	17	22	22	16	1	17	15	23	21
L	8	8	12	20	14	10	15	6	6	21	14	1	2	7	8
M	7	7	11	19	16	12	14	8	8	23	13	3	1	9	7
N	12	12	16	24	18	14	19	10	10	16	18	5	6	3	6
O	11	11	15	23	20	16	18	12	12	18	17	7	5	8	3

Table 2
Example influence calculation

InfluenceTable[*C*, *A*]	*C*	→	*q*	→	*B*	→	*p*	→	*A*	Influence
Alternatives	1	0	2	0	0	0	4	0	0	
Connection	1	1	2	1	0	1	4	1	0	
Tunnel	1	1	2	1	0	0	4	1	0	
Goal-skew	1	1	1	1	0	0	2	1	0	**7**
InfluenceTable[*A*, *C*]	*A*	→	*p*	→	*B*	→	*q*	→	*C*	Influence
Alternatives	2	0	4	0	0	0	1	0	1	
Connection	2	1	4	1	0	1	1	2	1	
Tunnel	2	0	4	1	0	0	1	2	1	
Goal-skew	1	0	4	1	0	0	1	2	1	**10**

that an influence relationship >8 implies that two squares are *distant* with respect to each other. How this threshold is determined is described in the next section.

In this example, square A is influencing squares B and C. However, only B is influencing A (the non-symmetric property). The table shows that there are several regions with high locality, whereas most of the entries indicate non-local relationships. Given the high percentage of non-local entries in the table, one might expect relevance cuts to eliminate most of the search tree. This is not quite true, in that a sequence of local moves can result in the start and end squares of the move sequence not being local with respect to each other.

Consider calculating the influence between squares A and C, as well as C and A (see Table 2). The table entries correspond to the contribution of each of the influence properties. The table indicates the influence scores for squares A, B, C, and the intermediate squares p and q, as well as for the connection between the squares (indicated by the arrows). Each line modifies the previous line (adding new values or changing existing values). The final influence, the sum of the preceding columns, is shown in the last column.

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, $m1$ and $m2$, move $m2$ is said to be 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

$$\mathit{InfluenceTable}[m1.from, m2.from] <= \mathit{infthreshold},$$

where *infthreshold* is a tunable threshold.

Relevance cuts eliminate some moves that are distant from the previous moves played (i.e. do not influence), 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.

To reflect differences in mazes, the parameters *infthreshold* and m are set at the beginning of the search. The maximal influence distance, *infthreshold*, is computed as follows:

1. Compute the average value for all entries *InfluenceTable*$[x, y]$ satisfying the condition that square y is on an optimal path from x to any goal.
2. The average is too high. Scale it back by dividing it by two.
3. To ensure that the cuts are not too aggressive, *infthreshold* is not allowed to be less than 6.

The length of history used, m, is calculated as follows:

1. Compute the average value for all entries *InfluenceTable*$[x, y]$ satisfying the condition that a stone on square y can be pushed to a goal (e.g. in Fig. 5, squares F and G would not be included).
2. To ensure that the cuts are not too aggressive, m is not allowed to be more than 10.

By varying *infthreshold* and m in the definition of relevance, the cutting in the search tree can be made more or less aggressive. The desired aggressiveness is application dependent, and should be chosen relative to the quality of the relevance metric used.

3.3. Example

Fig. 6 shows an example where humans immediately identify that solving this problem involves considering two separate sub-problems. The solution to the left and right sides of the problem are completely independent of each other. 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, meaning it iterates by 2 at a time). IDA* will try every possible (legal) move combination, intermixing moves from both sides of the problem. This way IDA* proves for each of the first 6 iterations ($i = 0..5$) 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.

Our implementation considers all moves on the left side as distant from those on the right, and vice versa. 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). The savings (25%) appear relatively small because the transposition table catches repeated positions (many of which may be the result of irrelevant moves) and eliminates them from the search. Although the relevance cuts provide a welcome reduction in the search effort required, it is only a small step towards achieving all the possible savings. For example, each of the sub-problems can be solved by itself in only 329 nodes! The difference between 329×2 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.

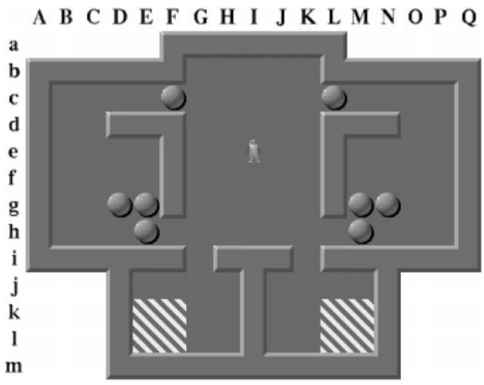


Fig. 6. Example maze with locality.

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 Section 6.

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. A closer look at relevance cuts

The goal of using relevance cuts is to reduce the search tree size. This is achieved by eliminating legal moves from the search, thereby reducing the effective branching factor of the tree. As with many other (unsafe) forward pruning techniques, this could potentially remove solutions or postpone their discovery. Therefore, aggressive pruning can increase the search effort by requiring additional search to find a non-pruned solution. A solution could be found in the same IDA* iteration, or could result in an additional iteration being started. A good heuristic for relevance is key to finding the right balance between tree reduction and the risk of eliminating solutions.

4.1. Relevance cuts in theory

To better understand the implications of relevance cuts, we will now try to apply Korf's theoretical model [12] to our algorithm. Section 5.2 discusses how well the model predicts the practical performance of our algorithm.

The number of nodes considered in a standard IDA* search is given by the following formula. This is a generalization of Korf's result [12].

$$n \approx \underbrace{\sum_{i=h(root)}^{d-1} b^{i-e}}_{complete\ iterations} + \underbrace{\frac{b^{d-e}}{1+s_d}}_{last\ (partial)\ iteration}, \tag{1}$$

where n is the total number of nodes, d is the length of optimal solutions, $h(root)$ is the heuristic value of the root node ($<=d$), b is the effective branching factor, e is the average heuristic value of the interior nodes in the tree, and s_d is the number of solutions with (optimal) length d.

In this formula, the variable depth search tree is approximated as a fixed depth tree. With no lower bound information, $h(position) = 0$, the search tree would be of size $O(b^d)$. An average lower bound of e reduces this exponent to $d - e$.

The first part of the formula represents the sum of the sizes of all the iterations that have no solution in them. The second part is the size of the last iteration. It assumes that the solutions are uniformly distributed throughout the leaf nodes. Thus, if there is only one unique solution path, that solution will be found, on average, half way through the search of the last (d) iteration.

Relevance cuts modify the equation in two ways. First, the iterations without solutions are reduced in size. This is achieved by eliminating moves from consideration, in effect reducing the branching factor. Second, there is the possibility that additional search will be needed if the first solution happens to be eliminated by a relevance cut. Thus, on iterations $>= d$ the savings from the reduced branching factor can be (partially) offset by having to do extra work. If all solutions at depth d happen to be cut off, then at least one more iteration is required (and possibly more). Eq. (1) is modified to reflect both ways that relevance cuts affect the search:

$$n \approx \underbrace{\sum_{i=h(root)}^{d-1} (b-r(x))^{i-e}}_{complete\ \ iterations} + \underbrace{\sum_{i=d}^{d+a(x)-1} (b-r(x))^{i-e}}_{additional\ \ full\ \ iterations} + \underbrace{\frac{(b-r(x))^{d+a(x)-e}}{1+(1-p(x))*s_{d+a(x)}}}_{last\ \ (partial)\ \ iteration} \tag{2}$$

$$\approx \underbrace{\sum_{i=h(root)}^{d+a(x)-1} (b-r(x))^{i-e}}_{complete\ \ iterations} + \underbrace{\frac{(b-r(x))^{d+a(x)-e}}{1+(1-p(x))*s_{d+a(x)}}}_{last\ \ (partial)\ \ iteration}, \tag{3}$$

where x is the aggressiveness of the cuts (in our relevance metric, this corresponds to changing m or *infthreshold*), $r(x)$ is the average branching factor reduction as a function of the aggressiveness, and $p(x)$ is the probability that a solution is cut from the search tree, assuming these probabilities are independent. $a(x)$ is the expected number of additional iterations, and it depends on the aggressiveness of the relevance cuts. $S_{d+a(x)}$ is the number of solutions at depth $d+a(x)$.

The effectiveness of relevance cuts in reducing the search tree size depends solely on the aggressiveness of the cuts, which controls the branching factor reduction and the penalty incurred for missing a solution. Increasing the aggressiveness of the cuts will decrease the number of nodes searched in the complete iterations (iterations $< d$), but will increase the risk of solutions being cut. When solutions are cut, not only can the last iteration potentially grow, but we might actually introduce new iterations when all the solutions contained in an iteration are pruned. Hence, relevance cuts can introduce non-optimal solutions, or postpone discovery of the solution beyond the resource limits.

The performance tuning effort must therefore be directed towards finding the right balance between savings (reduced search tree size) and cost (the overhead of having to search further than should be needed).

4.2. Randomizing relevance cuts

In a deterministic environment, where relevance cuts follow the exact same rules for the same situation, the search will always cut out solutions that depend on a maneuver mistakenly considered "irrelevant". Given that relevance cuts will make mistakes (albeit, hopefully, at a very low rate), some mechanism must be introduced to avoid worst-case scenarios, such as eliminating *all* solutions.

A solution is to introduce randomness into the relevance cut decision. If a branch is to be pruned by a relevance cut, a random number can be generated to decide whether to go ahead with the cut or not. The randomness reflects our confidence in the relevance cuts. For example, the random decision can be used to approve 100% of all possible relevance cuts (corresponding to the scheme outlined thus far, confident that not all solutions will be eliminated), down to 0% (which implies no confidence – relevance cuts will never be used). Somewhere between these two extremes is a percentage of cuts that balances the reductions in the search tree size with the overhead of postponing when a solution is found.

5. Experimental results

Rolling Stone has been tested using the 90-problem test set using searches limited to 25,000,000 nodes.[4] Our previous best version of *Rolling Stone* was capable of solving 40 of the test problems within this tree size limit. With the addition of relevance cuts (no random cutting), the number of problems solved has increased to 44. Table 3 shows a comparison of *Rolling Stone* with and without relevance cuts for each of the 44 solved problems. For comparison purposes, the 4 extra problems solved with relevance cuts had their non-relevance-cuts search continued beyond 25,000,000 nodes. This resulted in 3 of the problems being solved, with the fourth problem's search curtailed at 100,000,000 nodes (#49).[5]

For each program version in Table 3, the second column gives the number of IDA* iterations that the program took to solve the problem. Note that problems #9, #11, #21, #46, #49 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 cuts. However, since none of the problems solved before is lost and 4 more are solved within the 25,000,000 node limit, 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.

[4] Although this might seem like a small search, one should keep in mind that the Sokoban lower-bound function is an $O(N^3)$ algorithm. Instead of examining several million positions per second (as our implementation of the 15-puzzle can achieve), the program can only examine roughly 10,000 positions per second on a Silicon Graphics Origin 2000 with a 195 MHz processor. The node counts given are *total nodes*. In our previous papers, we break this number down into *top-level* and *pattern search* nodes [6].

[5] Subsequent runs have been to 300,000,000 nodes without finding a solution. This information is not included in Table 3 since it skews the sums even more than they are already.

Table 3
Experimental data

	Without relevance cuts		With relevance cuts	
No.	Total nodes	No. Iterations	Total nodes	No. Iterations
1	270	2	270	2
2	3251	2	2764	2
3	10,486	2	10,395	2
4	10,556	1	10,554	1
5	121,502	3	152,082	3
6	1593	3	1574	3
7	156,334	5	68,202	5
8	3,066,098	6	1,806,540	6
9	234,454	5	307,006	8
10	16,678,800	4	6,815,512	4
11	49,087,621	18	6,759,590	19
17	14,891	7	14,740	7
19	17,829,863	9	1,814,505	9
21	765,392	9	970,776	10
34	24,109,262	9	4,495,028	9
38	844,882	42	748,024	42
40	37,346,264	8	6,239,163	8
43	2,270,703	8	1,215,022	8
45	14,646,623	9	5,059,596	9
46	44,535,166	13	20,121,052	15
49	$> 100,000,000$	11	3,047,672	13
51	2611	1	18,368	2
53	3737	1	3740	1
54	4,381,171	9	3,884,844	9
55	3,194,830	3	1,349,908	3
56	34,429	6	31,388	6
57	1,084,732	5	797,766	5
60	116,103	3	24,403	3
62	71,578	5	46,534	5
63	197,922	3	390,422	3
64	3,900,639	10	652,857	10
65	12,971	5	12,971	5
67	2,177,787	13	2,103,866	13
68	2,651,559	11	355,306	11
70	15,003,603	3	1,949,842	3
72	43,260	5	72,647	5
73	247,816	3	292,799	3
78	809	1	783	1
79	4017	5	3512	5
80	15,513	1	48,220	1
81	40,806	4	29,698	4
82	181,571	5	97,406	5
83	15,423	1	13,106	1
84	521,068	4	443,508	4
	$> 345,637,966$		72,283,961	

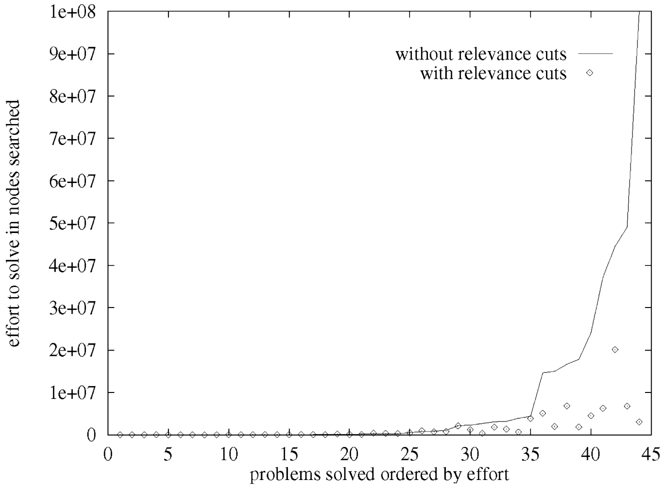


Fig. 7. The effect of relevance cuts.

Of the 4 new problems solved, #49 is of interest. Without relevance cuts, *Rolling Stone* is in its 11th IDA* iteration when the extended limit of 100,000,000 nodes is reached. We know from human solutions that solutions can be found in the 11th iteration. Relevance cuts allow *Rolling Stone* to find a solution using only 3,047,672 total nodes, but the search needs 13 iterations, producing a non-optimal solution.

Table 3 shows that relevance cuts improve search efficiency by a factor of 5. This is a lower bound. Clearly, the numbers are dominated by problem #49. If the node counts for #49 are ignored, then the savings are roughly a factor of 3.5.

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. However, by doing so a pattern search may now not be done that could have uncovered valuable information that might 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 15,513 to 48,220; 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 #19: the node count is cut by roughly a factor of 10.

In Fig. 7, the amount of effort to solve a problem, with and without relevance cuts, is plotted. The numbers from Table 3 are used, sorted by the number of nodes searched by the version without relevance cuts. The figure shows that the exponential growth in difficulty with each additional problem solved is being dampened by relevance cuts, allowing for more problems being solved with the same search constraints. For the 25 "easiest" problems, there is very little difference in effort required; the relevance cuts do not save significant portions of small search trees. However, as the searches become larger, the success of relevance cuts gets more pronounced.

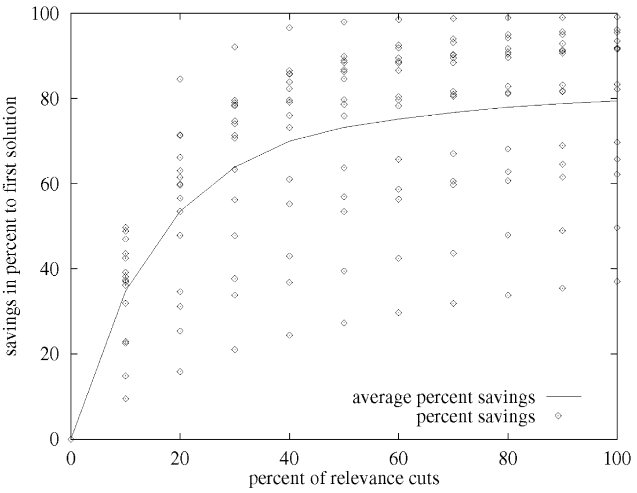


Fig. 8. Relevance cuts savings.

5.1. Randomizing relevance cuts

The numbers presented so far deal with a version of a program that executes 100% of the relevance cuts. A version of *Rolling Stone* was instrumented to simulate the effects of different degrees of randomization, varying from 0% (all relevance cuts are ignored) to 100% (all relevance cuts are used). Thus, the level of, say, 80% corresponds to randomly accepting 80% of the cuts, while rejecting 20% of them.

Fig. 8 illustrates the search tree savings potential for relevance cuts. The graph presents for various degrees of randomness (from 0% to 100% in 10% increments) the percent of the search tree that can be saved by the relevance cuts. For each search, the relative savings are plotted. Of the 44 solved problems, only searches where the 0% entry required at least 10,000 nodes were included. The small search trees ($<10,000$ nodes) were excluded from this and subsequent graphs, since these trees tend to have very few opportunities for savings. For example, problem #1 is already a paltry 270 nodes; there is neither need nor room for further improvement. Each of the data points in a column corresponds to one of the 15 problems that passed our filter. The line represents the average of all the savings.

The figure shows that roughly 80% of the search tree can be eliminated by relevance cuts. Further, one need only perform 40% of the cuts to reduce the search by 70%. Thus, even a small amount of cutting can translate into large savings. Keep in mind that these numbers reflect our implementation; different implementations could do better or worse than portrayed here.

To put this in perspective, one might suggest that the relevance cuts are just a fancy way of randomly cutting branches in the search tree. An additional experiment was performed where cutting was done randomly, in line with the frequency of relevance cuts. The result was some savings for a small amount of cutting, but as the frequency of cutting increased, so did the search tree sizes! By cutting randomly, more solution paths were being eliminated from the search, increasing the likelihood of having to search more iterations.

Eq. (3) essentially broke the relevance cuts search nodes into two components. The first was the search effort required to reach the first solution. Clearly, relevance cuts provably reduce this portion of the search since some branches are not explored. In fact, Fig. 8 is portraying exactly these savings. However, these savings can be offset by the second component, the additional effort needed to find a non-cut-off solution.

Of the 44 problems solved, 6 have non-optimal solutions. Hence, roughly 13% of the problems are non-optimal. As stated earlier, solution quality is not a concern since, given the difficulty of the problem domain, *any* solution is welcome. The significance of these non-optimal solutions is discussed in the next subsection.

5.2. Relevance cuts in theory revisited

Let's revisit Eq. (3). These generic formulas contains several assumptions, some of which are explicitly stated in [12], while others are implicit. In theory, we should be able to use our experimental data to confirm these equations. Of interest in Eq. (3) is that the term

$$(b-r)^{d-e} \tag{4}$$

dominates the calculation. We know d (the optimal solution length), and we can measure b, r and e. *Rolling Stone* has been instrumented to measure these quantities.

Fig. 9 shows the average b and r for all 90 solved problems, sorted in order of increasing b. These statistics were gathered at nodes in the search that were visited by

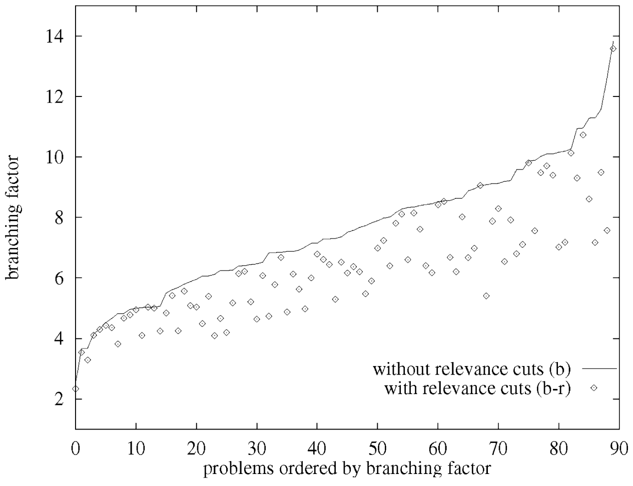


Fig. 9. Measuring b and r.

both programs (one with relevance cuts; the other without). Searches were limited to a maximum of 25 million nodes. In other words, nodes which were visited only by the non-relevance cuts program were not averaged in. As can be seen, the reduction in branching factor varies dramatically, depending on the problem.

Measuring e, the average heuristic value of the interior nodes in the tree, showed little difference with/without relevance cuts.

Plugging d, e, b and r into Eq. (4) produced a large discrepancy between the predicted tree size and the observed tree size. Since d is constant in both versions of the program, and e is effectively a constant, the improvements of relevance cuts rests solely on r, the reduction in the branching factor. However, in most cases the observed savings are significantly *larger* than the predicted savings.

Korf's formula, which led to Eq. (3), has the implicit assumption that the branching factor is relatively uniform throughout the tree. Certainly this was true for the sliding-tile puzzle he was studying. But Sokoban has different properties. In particular, the branching factor can swing wildly from move to move. As well, our data shows that the branching factor tends to be smaller near the root of the tree (too many obstacles in the puzzle) and, as the problem simplifies (log jams get cleared, stones get pushed to their goal squares), the branching factor increases until near the end of the game when there are few stones left to move. In addition, the data shows that the relevance cuts tend to occur early in the search, rather than later. Hence, the majority of the savings from relevance cuts come from the smaller branching factor b near the root of the tree combined with a larger branch reduction. Korf's formula only considers averages over the entire tree, whereas any bias towards the root of the tree can produce larger observed reductions.

The other component of Eq. (3) is the additional search effort required when relevance cuts miss the first solution. Earlier, it was suggested that the probability of searching an extra iteration was quite high (13%). This suggests that the relevance cuts are being too extreme in their cutting. *Rolling Stone* was instrumented to keep searching subtrees that would have been eliminated by a relevance cut to determine if a solution path lay in that subtree. Fig. 10 shows that only about 0.2% of the cuts eliminate a solution. Note that some problems have a relatively high error rate; these results come from the problems that have small searches, where the total number of cuts is small and a single error can skew the percentages.

A relevance cut error rate of 0.2% might seem high. However, consider that these cuts are done throughout the tree, including near the root. Given that a cut near the root of the search will eliminate huge portions of the search space, and few of these cuts eliminate any optimal solution, the cuts must be doing a good job of identifying irrelevant portions of the search.

Infrequently eliminating solutions (0.2%) may seem important if there are few solutions. In fact, our experience with Sokoban shows that there are many optimal solutions for every problem. The number of solution paths grows exponentially with any additional search beyond the optimal solution length. For example, consider a d-ply optimal

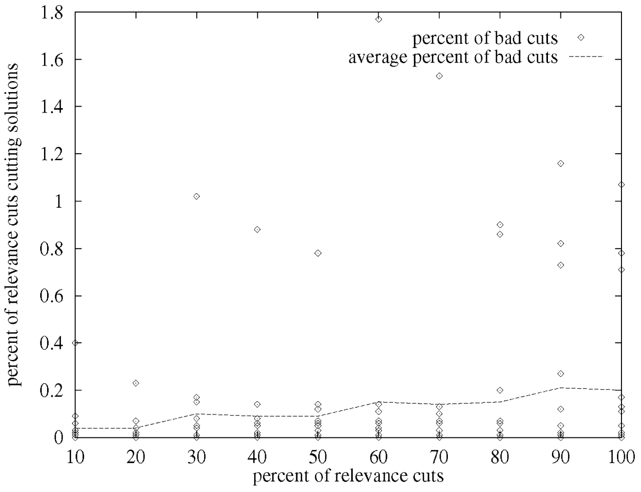


Fig. 10. Percent of relevance cuts eliminating solutions.

solution. If we now consider solutions of length $d+2$,[6] then we can randomly insert an irrelevant move (and its reverse move) into the solution path, giving $O(d*b)$ more solution paths.

Eq. (3) assumes that the probability of a solution being cut is independent from any other solution being cut. Unfortunately, that is a simplifying assumption that does not hold for Sokoban. Since Sokoban problems have been composed to be challenging to humans (and, inadvertently, computers as well), most problems in our test suite contain specific maneuvers that are *mandatory* for all solutions. In other words, every solution to some of the problems requires a specific sequence of moves to be made. We call these maneuvers *solution articulation sequences*.

A solution articulation sequence is illustrated in Fig. 11. It shows the set of move sequences that are solutions to the problem of getting from the start state to the goal state. First, there are many possible sequences of moves (possibly even move transpositions) until a specific maneuver is required. Then a fixed sequence of moves is required (the solution articulation sequence). Having completed the sequence, then many different permutations of moves can be used to reach the goal(s). Note that a problem may have multiple solution articulation sequences. As well, there may be classes of solutions, with each class having a different set of articulation sequences.

Relevance cuts use a sequence of moves (the past m moves) to decide whether to curtail the search or not. If the moves forming the solution articulation sequence happen to meet the criterion for a relevance cut, then it will be falsely considered

[6] In general, this would be $d+1$. However, since Sokoban solutions preserve odd/even parity, solutions increase by two pushes at a time.

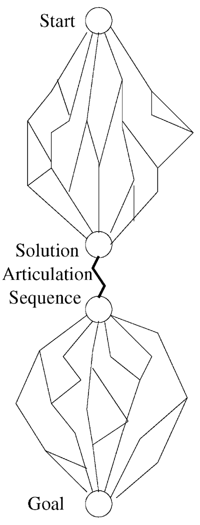


Fig. 11. Solution articulation sequence.

“irrelevant”. Consequently, many solution paths will be eliminated from the search. One can construct a scenario by which *all* solutions could be removed from the search.

Solution articulation sequences illustrate that the assumed solution independence property is, in fact, incorrect. Coming up with a realistic model is difficult. The solutions tend to be distributed in clusters. Many clusters of solutions are, essentially, the same solution with minor differences (such as move transpositions or, for non-optimal solutions, irrelevant moves added).

Although the number of optimal solutions appears high from our experiments, relevance cuts are vulnerable to solution articulation sequences. Hence, a single cut has the potential for eliminating *all* solutions. Randomization seems to be an effective way of handling this problem.

5.3. Summary

Relevance cuts have been shown experimentally to result in large reductions in the effort required to solve Sokoban problems. Given the exponentially increasing nature of the search trees, solving an extra 4 problems represents a substantial improvement.

Although it would be nice to have a clean analytic model for Sokoban searches that could be used to predict search effort, this is proving elusive. Although a model for single-agent search exists [12], it is inadequate to handle the non-uniformity of Sokoban. In the past, numerous analytic models for tree-searching algorithms have appeared in the literature. They are all based on simplifying assumptions that make the analysis tractable, but result in a model that mimics an artificial reality. Historically, these models correlate poorly with empirical data from real-world problems. An interesting recent example from two-player search can be found in [14].

6. 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.

We have numerous ideas on how to improve the effectiveness of relevance cuts. Some of them include:

– 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 fewer stones blocking each other. Dynamic influence measures should be better than static approaches.
– There are several parameters used in the relevance cuts. The setting of those is already dependent on 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 these details is needed to fully exploit the possibilities relevance cuts are offering.
– Using the analogy from Section 1, one could characterize *Rolling Stone* as "painting" locally but not yet painting in an "object oriented" way. 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 to do a lot of research if we want to successfully solve all 90 problems!

Acknowledgements

This research has been supported by the Killam Foundation and the Natural Sciences and Engineering Research Council of Canada (NSERC). Computational resources were provided by the Multimedia Advanced Computational Infrastructure initiative (MACI).

References

[1] J. Culberson, Sokoban is PSPACE-complete, Technical report TR97-02, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1997, ftp.cs.ualberta.ca/pub/TechReports/1997/TR97-02.
[2] J. Culberson, J. Schaeffer, Searching with pattern databases, in: G. McCalla (Ed.), A1'96 Advances in Artificial Intelliegnce, Springer, Berlin, 1996, pp. 402–416.
[3] A. Dor, U. Zwick, SOKOBAN and other motion planning problems, 1995, at: http:/www.math.tau.ac.il/ ∼ ddorit.
[4] M. Ginsberg, Essentials in Artifical Intelligence, Morgan Kaufman, San Francisco, 1993.
[5] G. Goetsch, M.S. Campbell, Experiments with the null-move heuristic, in: T.A. Marsland, J. Schaeffer (Eds.), Computers, Chess, and Cognition, Springer, New York, 1990, pp. 159–181.
[6] A. Junghanns, J. Schaeffer, Single-agent search in the presence of deadlock, in: AAAI, Madison/WI, USA, July 1998, pp. 419–424.
[7] A. Junghanns, J. Schaeffer, Sokoban: evaluating standard single-agent search techniques in the presence of deadlock, in: R. Mercer, E. Neufeld (Eds.), AI'98 Advances in Artificial Intelligence, Springer, Berlin, 1998, pp. 1–15.
[8] A. Junghanns, J. Schaeffer, Domain-dependent single-agent search enhancements, IJCAI, 1999, pp. 570–575.
[9] A. Junghanns, J. Schaeffer, Relevance cuts: localizing the search, in: H.J. van den Herik, H. Iida (Eds.), Computers and Games, Lecture Notes in Computer Science, Vol. 1558, Springer, Berlin, 1999, pp. 1–14.
[10] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artificial Intelligence 27 (1) (1985) 97–109.
[11] R.E. Korf, Macro-operators: a weak method for learning, Artifical Intelligence 26 (1) (1985) 35–77.
[12] R.E. Korf, Finding optimal solutions to Rubik's Cube using pattern database, AAAI, 1997, pp. 700–705.
[13] H.W. Kuhn, Extensive games and the problem of information, in: H.W. Kuhn, A.W. Tucker (Eds.), Contributions to the Theory of Games, Vol. 2, Princton University Press, Princeton, 1953, pp. 193–296.
[14] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin, Best-first fixed-depth minimax algorithms, Artificial Intelligence 87 (1–2) (1996) 255–293.
[15] J. Schaeffer, Experiments in search and knowledge, Ph.D. Thesis, University of Waterloo, Canada, 1986.
[16] G. Wilfong, Motion planning in the presence of movable obstacles, Fourth ACM Symp. on Computational Geometry, 1988, p. 278–288.
[17] A. Junghanns, New Developments in Single-Agent Search, Ph.D. Thesis, University of Alberta, 1999.
[18] H.W. Kuhn, The Hungarian method for the assignment problem, Naval Res. Logist. Quartz., (1995) 83–98.