Sokoban: Evaluating Standard Single-Agent Search Techniques in the Presence of Deadlock*

Andreas Junghanns, Jonathan Schaeffer
University of Alberta
Dept. of Computing Science
Edmonton, Alberta
CANADA T6G 2H1
Email: {andreas, jonathan}@cs.ualberta.ca

No Institute Given

Abstract. Single-agent search is a powerful tool for solving a variety of applications. Most of the academic application domains used to explore single-agent search techniques have the property that if you start with a solvable state, at no time in the search can you reach a state that is unsolvable (it may, however, not be minimal). In this paper we address the implications that arise when states in the search are unsolvable. These so-called deadlock states are largely responsible for the failure of our attempts to solve positions in the game of Sokoban.

Keywords: single agent search, heuristic search, Sokoban, deadlocks, IDA*

1 Introduction

Single-agent search (A*) has been extensively studied in the literature. There are a plethora of enhancements to the basic algorithm – a programmer's tool box – that allows one to tailor the solution to the problem domain to maximize program performance (such as iterative deepening [Kor85a], transposition tables [RM94] and pattern databases [CS96]). The result has been some impressive reductions in the search effort required to solve challenging applications (see [Kor97] for a recent example).

Many of the academic applications used to illustrate single-agent search (such as sliding-tile puzzles and Rubik's Cube) are "easy" in the sense that they have some (or all) of the following properties: simple and effective lower-bound estimators, small branching factors and moderate solution depths. These domains also have the nice property that given a solvable starting state, every move preserves the solvability. This means that one can construct a real-time search (anytime algorithm) that can be guaranteed to find a (non-optimal) solution [Kor90].

* This is a revised and updated version of a paper presented at the IJCAI workshop on Using Games as an Experimental Testbed for Artificial Intelligence Research, Nagoya, August, 1997.

In the real world, one often has to make irrevocable decisions. Similarly, search applications (particularly if they are real-time) have to deal with this complexity. In competitive games programs (such as for chess), the irreversible moves mean that one may play a move that loses. In the context of single-agent search, an irreversible move means that one may move from a solvable state to an unsolvable one. A short-term decision (within the constraints of real time) may lead to a long-term disaster (problem cannot be solved). How to deal with this is a difficult problem.

Sokoban is a popular one-player game. Given a topology of rooms and passageways, the object is to push a number of stones from their current locations to goal locations. Of interest is that since you can only push, never pull, a single move can transform the problem from being solvable to being unsolvable. Many of these so-called deadlock states are trivial to identify (and avoid in the search), but some require extensive analysis to prove their existence. For example, one can easily construct an unsolvable Sokoban position that will require a massive search tree that is hundreds of moves deep to verify the deadlock.

Sokoban is a difficult search application for many reasons:

1. it has a complex lower-bound estimator,
2. the branching factor is large and variable (potentially over 100),
3. the solution may be very deep in the search tree (some problems require over 500 moves to solve optimally), and
4. some states are provably unsolvable (deadlock).

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

In this paper, we evaluate the standard single-agent search techniques while trying to optimally solve Sokoban problems. We identify the existence of deadlock as a new property of the search space that has not been addressed in previous research. We argue that even though standard search techniques have a dramatic impact on the size of the search, they are insufficient to solve most of the standard Sokoban problems.

We have constructed an IDA*-based program to solve Sokoban problems (*Rolling Stone*). In addition to using the standard single-agent search enhancements (such as transposition tables and move ordering) we introduce a good lower-bound estimator, deadlock tables, the inertia heuristic and macro moves that preserve the optimality of the solution. Despite these enhancements chopping orders of magnitude from the search tree size, we can solve only 16 of the 90 benchmark problems. Although this sounds rather poor, it is the best reported result to date. We believe our techniques can be extended and more problems will be solved. However, we also conclude that some of the Sokoban problems are so difficult as to be effectively unsolvable using standard single-agent search techniques.

2 Sokoban

Sokoban is a popular one-player game. The game apparently 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 difficult problems.

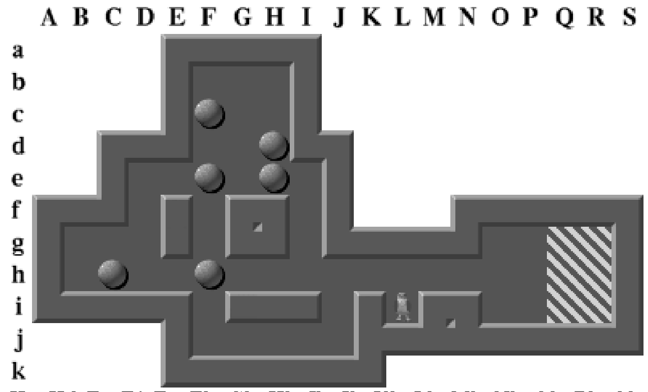
The rules of the game are quite simple. Figure 1 shows a sample Sokoban problem (problem 1 of the standard 90-problem suite available at http://xsokoban.lcs.mit.edu/xsokoban.html). The playing area consists of rooms and passageways, laid out on a rectangular grid of size 20x20 or less. Littered throughout 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 nodes 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" (assuming the maximum sized 20x20 problem), 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*).

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 that gives the best solution achieved to date (at the above www address). Thus solving a problem is only part of the satisfaction; improving on one's solution is equally important.

Note that there are two definitions of an optimal solution to a Sokoban problem: the number of stone pushes and the number of man movements. For a few problems there is one solution that optimizes both; in general they conflict. In this paper, we have chosen to optimize the number of stone pushes. Both optimization problems are computationally equivalent. Using a single-agent search algorithm, such as IDA* [Kor85a], one stone push decreases the solution length by at most one, but may increase it by an arbitrary amount. Optimizing the man movements involves using non-unitary changes to the lower bound (the number of man movements it takes to position the man behind a stone to do the push).

Sokoban has been shown to be NP-hard [Cul97,DZ95]. [DZ95] show that the game is an instance of a motion planning problem, and compare the game to other motion planning problems in the literature. 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 [Wil88].



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

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.

3 Why is Sokoban so Interesting?

Although the authors are well-versed in single-agent search, it quickly became obvious that Sokoban is not an ordinary single-agent search problem. Much of the single-agent search literature concentrates on "simple" problems, such as the sliding tile puzzles or Rubik's Cube. The following is a listing of problems encountered with Sokoban that make it difficult and essentially cause a program based solely on the standard single agent-search techniques to fail to solve more than a handful of problems.

3.1 Lower Bound

In general, it is hard to get a tight lower bound on the solution length for Sokoban problems. The tighter the bound, the more efficient a single-agent search algorithm can be. The stones can have complex interactions, with long elaborate maneuvers often being required to reposition stones. For example, in problem 50 (see Figure 2), the solution requires moving stones *through* and *away* from the goal squares to make room for other stones. Our best lower bound is 100 stone pushes (see section 4.1), whereas the best human solution required 370 moves – clearly a large gap, and an imposing obstacle to an efficient IDA* search. For some problems, without a deep understanding of the problem and its solution, it is difficult to get a reasonable bound.

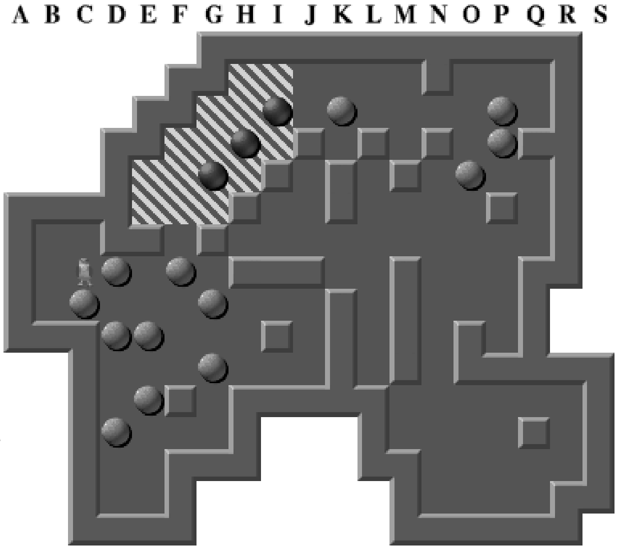


Fig. 2. Sokoban problem 50

3.2 Deadlock

In most of the single-agent search problems studied in the literature, all state transitions preserve the solvability of the problem (but not necessarily the optimality of the solution). This is a consequence of all state transitions (moves) being reversible (there exists a move sequence which can undo a move). Sokoban has irreversible moves (e.g. pushing a stone into a corner), and these moves can lead to states that provably cannot lead to solutions. In effect, a single move can change the lower bound on the solution length to infinity. If the lower bound function does not reflect this, then the search will spend unnecessary effort exploring a sub-tree that has no solution. We call these states *deadlocks* because one or more stones will never be able to reach a goal.

Deadlocks can be as trivial as, for example, moving a stone into a corner (in Figure 1, moving *Ch-Bh*[1] creates a deadlock state; the man can never get

[1] This is in fact an illegal move in that position, since the man can't reach the stone. We assume here, that the stone on Fh was not in the maze.

behind the stone to push it out). Some deadlocks can be wide ranging and quite subtle, involving complex interactions of stones over a large portion of the maze (in Figure 1, moving *Fh-Fg* creates a deadlock). Any programming solution to Sokoban must be able to detect deadlock states so that unnecessary search can be curtailed.

The presence of deadlock states in a search space creates a serious dilemma for a real-time search applications (anytime applications). If we have to commit to a move (because of resource constraints) we may move to a deadlock state, guaranteeing insolvability. Since many of these deadlock scenarios cannot be determined without search, the real-time algorithm will have a difficult time allocating resources to guarantee that a solution will be found.

3.3 Size of Search Space

Sliding tile puzzles have a branching factor of less than 4 and the maximum solution length for the 15-puzzle is 80. Rubik's Cube has a branching factor of 18 and a maximum solution length of 20 [Kor97].

The large size of the search space for Sokoban is due to potentially large branching factors and long solution lengths, compared to the previously studied problem domains. The number of stones ranges from 6 to 34 in the standard problem set. With 4 potential moves per stone, the branching factor could be well over 100! The solution lengths range from nearly 100 to over 650 pushes. The trees are bushier and deeper than previously studied problems, resulting in a search space that is considerably larger.

The large size of the search space for Sokoban gives rise to a surprising result. Consider problem 48 (Figure 3). Our program computes the lower bound as 200 moves. Since human players have solved it in 200 moves, we can conclude that the optimal solution requires exactly 200 moves. Knowing the solution length is only part of the answer – one has to find the sequence of moves to solve the problem. In fact, problem 48 is difficult to solve because of the large branching factor. Although IDA* will never make a non-optimal move according to its heuristic estimate of the distance to the goal, it has no idea what order to consider the moves in. An incorrect sequence of moves can lead eventually to deadlock. For this problem, to IDA* one optimal move is as good as another. The program builds a huge tree, trying all the optimal moves in all possible orders. Hence, even though we have the right lower bound, our program builds an exponentially large tree and fails to solve problem 48.

4 Towards Solving Sokoban

Although we believe that standard search algorithms, such as IDA*, will be inadequate for solving all 90 Sokoban problems, as a first step we decided to invest our efforts in pushing the IDA* technology as far as possible. Our goal is to eventually demonstrate the inadequacy of single-agent search techniques for this puzzle. This section discusses our work with IDA* and the problems we are encountering.

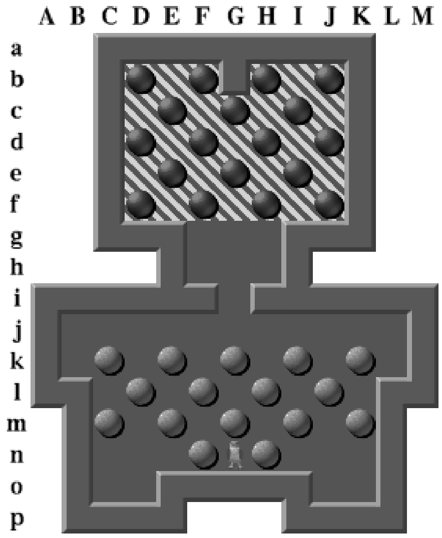


Fig. 3. Sokoban problem 48

4.1 Lower Bound

A naive but computationally inexpensive lower bound is the sum over the distances of all the stones to their respective closest goal. It is clear however that only one stone can go to any one goal in any solution. Since there are as many stones as there are goals and every stone has to be assigned to a goal, we are trying to find a *minimum cost* (distance) *perfect matching* on a complete bipartite graph. Edges between stones and goals are weighted by the distance between them, and assigned infinity if the stone cannot reach a goal.

Essentially the problem can be summarized as follows. There are n stones and n goals. For each stone, there is a minimum number of moves that is required to maneuver that stone to each goal. For each stone and for each goal, there is a distance (cost) of achieving that goal. The problem then is to find the assignment of goals to stones that minimizes the sum of the costs.

Minimum cost perfect matching for a bipartite graph can be solved using minimum cost augmentation [Kuh55]. Given a graph with n nodes and m edges, the cost of computing the minimal cost matching is $O(n * m * log_{(2+m/n)}n)$. Since we have a complete bipartite graph, $m = n^2/4$ and the complexity is $O(n^3 * log_{(2+n/4)}n)$. Clearly this is an expensive computation, especially if it has to be computed for every node in the search. However, there are several optimizations that can reduce the overall cost. First, during the search we only need to update the matching, since each move results in a single stone changing its distance to the goals. This requires finding a negative cost cycle [Kle67] involving the stone moved. Second, we are looking for a perfect matching, which considerably reduces the number of possible cycles to check. Even with these optimizations, the cost of maintaining the lower bound dominates the execution time of our program. Most of the lower bounds used in single-agent search in the literature, such as the Manhattan distance used for sliding tile puzzles, are trivial in comparison.

One advantage of the minimum matching lower bound is that it correctly returns the parity of the solution length (Manhattan distance in sliding tile puzzles also has this property). Thus, if the lower bound is an odd number, the solution length must also be odd. Using IDA*, this property allows us to iterate by two at a time.

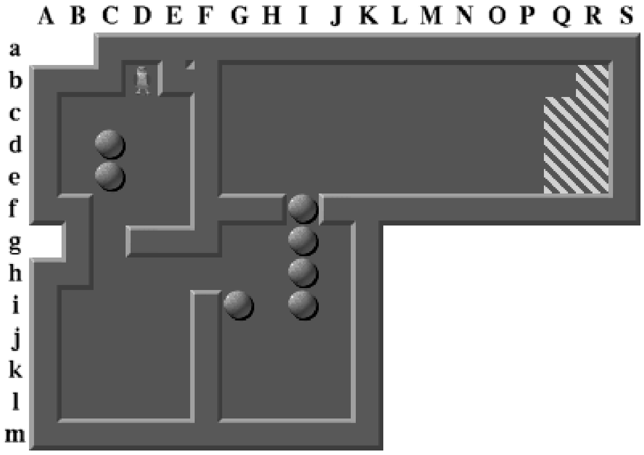


Fig. 4. Illustrating lower bound calculations

There are a number of ways to improve the minimum matching lower bound. Here we introduce two useful enhancements. First, if two adjacent stones are in each other's way towards reaching their goal, then we can penalize this position by increasing the lower bound appropriately. We call this enhancement *linear conflicts* because of its similarity to the linear conflicts enhancement in sliding-tile puzzles [HMY92]. Figure 4 shows an obvious example. The four stones on *If*, *Ig*, *Ih* and *Ii* are obstructing each others' optimal path to the goals[2]. We have to move two stones off their optimal paths to be able to solve this problem (for example, *Ig*-*Hg* to allow the man to push *If*, and *Ii*-*Hi* to move the stone on *Ih*). In each case, two additional moves are required. In addition, the stones on *Cd* and *Ce* have a linear conflict. Hence, in this example, the lower bound will be increased by six.

The second enhancement notes that sometimes stones on walls have to be *backed out* of a room, and then pushed back in just to re-orient the position of

[2] The optimal path is defined as the route a stone would take if no other stone was in the maze obstructing its movements.

the man. In Figure 4, the stone on *Gi* has a *backout conflict*. Consider this stone while pretending there are no other stones on the board. The man must move the stone to a room entrance (*Gi*-*Gh*), push it out of the room (*Gh*-*Fh*-*Eh*-*Dh*), and then push it back into the room it came from (*Dh*-*Eh*-*Gh*-*Hh*). This elaborate maneuver is required because the man has to be on the left side of the stone to be able to push it off the wall. In this problem, there is only one way to get to the left of the stone – by backing it out and then back into the room. This conflict increases the lower bound by six[3].

Table 1 shows the effectiveness of our lower bound estimate. The table shows the lower bound achieved by minimum matching (MM), inclusion of the linear conflicts enhancement (+LC), inclusion of the backout enhancement (+BO), and the combination of all three features (ALL). The upper bound (UB) is obtained from the global Sokoban score file. Since this file represents the best that human players have been able to achieve, it is an upper bound on the solution length. The table is sorted according to the last column (Diff), which shows the difference between the lower and upper bound. Clearly for some problems (notably problem 50) there is a huge gap. Note that the real gap might be smaller, as it is likely that some of the hard problems have been non-optimally solved by the human players.

4.2 Transposition Table

The search tree is really a graph. Two different sequences of moves can reach the same position. The search effort can be considerably reduced by eliminating duplicate nodes from the search. A common technique is to use a large hash table, called the transposition table, to maintain a history of nodes visited [SA77]. Each entry in the table includes a position and information on the parameters that the position was searched with. Transposition tables have been used for a variety of single-agent search problems [RM94].

One subtlety of Sokoban is that saving exact positions in the transposition table misses many transpositions. While the exact positions of the stones is critical, the exact position of the man is not. Two positions, A and B, are identical if both positions have stones on the same squares and if the man in A can move to the location of the man in B. Thus, when finding a match in the transposition table, a computation must be performed to determine the reachability of the man. In this way, the table can be made more useful, by allowing a table entry to match a class of positions.

4.3 Deadlock Tables

Our initial attempt at avoiding deadlock was to hand-code a set of tests for simple deadlock patterns into *Rolling Stone*. This quickly proved to be of limited value,

[3] Another increase by 4 is achieved by observing that stones moving into the goal room and targeted at the two lower goals need two extra pushes each to allow the man into the room, before pushing it to the right.

#	MM	+LC	+BO	ALL	UB	Diff	#	MM	+LC	+BO	ALL	UB	Diff
51	118	118	118	118	118	0	34	152	154	152	154	168	14
55	118	118	120	120	120	0	71	290	294	290	294	308	14
78	134	134	136	136	136	0	40	310	310	310	310	324	14
53	186	186	186	186	186	0	35	362	364	362	364	378	14
83	190	194	190	194	194	0	36	501	507	501	507	521	14
48	200	200	200	200	200	0	41	201	203	219	221	237	16
80	219	225	225	231	231	0	45	274	276	282	284	300	16
4	331	331	355	355	355	0	19	278	280	282	286	302	16
1	95	95	95	95	97	2	22	306	308	306	308	324	16
2	119	119	129	129	131	2	20	302	304	444	446	462	16
3	128	132	128	132	134	2	18	90	90	106	106	124	18
58	189	189	197	197	199	2	21	123	127	127	131	149	18
6	104	106	104	106	110	4	13	220	220	220	220	238	18
5	135	137	137	139	143	4	31	228	232	228	232	250	18
60	148	148	148	148	152	4	64	331	331	367	367	385	18
70	329	329	329	329	333	4	25	326	330	364	368	386	18
63	425	427	425	427	431	4	90	436	436	442	442	460	18
73	433	433	437	437	441	4	49	96	96	104	104	124	20
84	147	147	149	149	155	6	42	208	208	208	208	228	20
81	167	167	167	167	173	6	61	241	243	241	243	263	20
10	494	496	506	506	512	6	28	284	286	284	286	308	22
38	73	73	73	73	81	8	68	317	319	319	321	343	22
7	80	80	80	80	88	8	39	650	652	650	652	674	22
82	131	135	131	135	143	8	46	219	219	223	223	247	24
79	164	166	164	166	174	8	67	367	369	375	377	401	24
65	181	185	199	203	211	8	23	286	286	424	424	448	24
12	206	206	206	206	214	8	32	111	113	111	113	139	26
57	215	217	215	217	225	8	16	160	162	160	162	188	26
9	215	217	227	229	237	8	85	303	303	303	303	329	26
14	231	231	231	231	239	8	89	345	349	349	353	379	26
62	235	235	237	237	245	8	24	442	442	516	518	544	26
72	284	288	284	288	296	8	15	94	96	94	96	124	28
77	360	360	360	360	368	8	33	140	140	150	150	180	30
54	177	177	177	177	187	10	26	149	149	163	163	195	32
56	191	191	193	193	203	10	11	197	201	201	207	241	34
76	192	194	192	194	204	10	75	261	263	261	263	297	34
47	197	199	197	199	209	10	29	124	124	122	122	164	42
8	220	220	220	220	230	10	74	158	158	172	172	214	42
27	351	353	351	353	363	10	37	220	220	242	242	290	48
86	122	122	122	122	134	12	88	306	308	334	336	390	54
44	167	167	167	167	179	12	52	365	367	365	367	423	56
17	121	121	201	201	213	12	30	357	359	357	359	465	106
59	218	218	218	218	230	12	66	185	187	185	187	325	138
87	221	221	221	221	233	12	69	207	209	217	219	443	224
43	132	132	132	132	146	14	50	96	96	96	100	370	270

Table 1. Lower bounds

since it missed many frequently occurring patterns, and the cost of computing the deadlock test grew as each test was added. Instead, we opted for a more "brute-force" approach.

Rolling Stone includes a pattern database [CS96] that we call *deadlock tables*. An off-line search is used to enumerate all possible combinations of walls, stones and empty squares for a fixed-size region. For each combination of squares and square contents, a small search is performed to determine if deadlock is present or not. This information is stored in a tree data structure. There are many optimizations that make the computation of the tree efficient. For our experiments, we built two differently shaped deadlock tables for regions of roughly 5x4 squares (containing approximately 22 million entries).

When a move *Xx*-*Yy* is made, the destination square *Yy* is used as a base square in the deadlock table and the direction of the stone move is used to rotate the region, such that it is oriented correctly. In Figure 4 if the move *Gi*-*Gh* is made, then a deadlock table could cover the 5x4 region bounded by the squares *Eh*, *Ee*, *Ie* and *Ih*. Note that the table can be used to cover other regions as well. To maximize the usage of the tables, reflections of asymmetric patterns along the direction the stone was moved in are considered.

Although a 5x4 region may sound like a significant portion of the 20x20 playing area, in fact many deadlocks encountered in the test suite extend well beyond the area covered by our deadlock tables. Unfortunately, it is not practical to build larger tables.

Note that if a deadlock table pattern covers a portion of the board containing a goal node, most of the effectiveness of the deadlock table is lost. Once a stone is on a goal square, it need never move again. Hence, the normal conditions for deadlock do not apply. Usually moving a stone into a corner creates a deadlock, but if the square is a goal node, then the position is not necessarily a deadlock.

4.4 Macro Moves

Macro moves have been described in the literature [Kor85b]. Although they are typically associated with non-optimal problem solving, we have chosen to investigate a series of macro moves that preserve the solution's optimality. We implemented the following two macros in *Rolling Stone*.

Tunnel Macros In Figure 1, consider the man pushing a stone from *Jh* to *Kh*. The man can never get to the other side of the stone, meaning the stone can only be pushed to the right. Eventually, the stone on *Kh must* be moved further: *Jh*-*Kh*-*Lh*-*Mh*-*Nh*-*Oh*-*Ph*. Once the commitment is made (*Jh*-*Kh*), there is no point in delaying a sequence of moves that must eventually be made. Hence, we generate a macro move that moves the stone from *Jh* to *Ph* in a single move.

The above example is an instance of our *tunnel macro*. If a stone is pushed into a one-way tunnel (a tunnel consisting of articulation points[4] of the underlying graph of the maze), then the man has to push it all the way through to the

[4] Squares that divide the graph into two disjoint pieces.

other end. Hence this sequence of moves is collapsed into a single macro move. Note that this implies that macro moves have a non-unitary impact on the lower bound estimate.

Goal Macros As soon as a stone is pushed into a room that contains goals, then the single-square move is substituted with a macro move to move the stone directly to a goal node. Unlike with the tunnel macro, if a goal macro is present, *it is the only move generated.* This is illustrated using Figure 1. If a stone is pushed onto the room containing the goal squares (such as square *Oh*), then this move is substituted with the goal macro move. This pushes the stone all the way to the next highest priority empty goal square (*Rg* or *Ri* if it is the first stone into the goal area). The goals are prioritized in a pre-search phase. This is necessary to guarantee that stones are moved to goals in an order that precludes deadlock and preserves optimality of the solution.

In Figure 1 a special case can be observed: the end of the tunnel macro overlaps with the beginning of the goal macro. The macro substitution routine will discover the overlap and chain both macros together. The effect is that one longer macro move is executed. In the solution given in Figure 1, the macro moves are underlined (an underlined move should be treated as a single move).

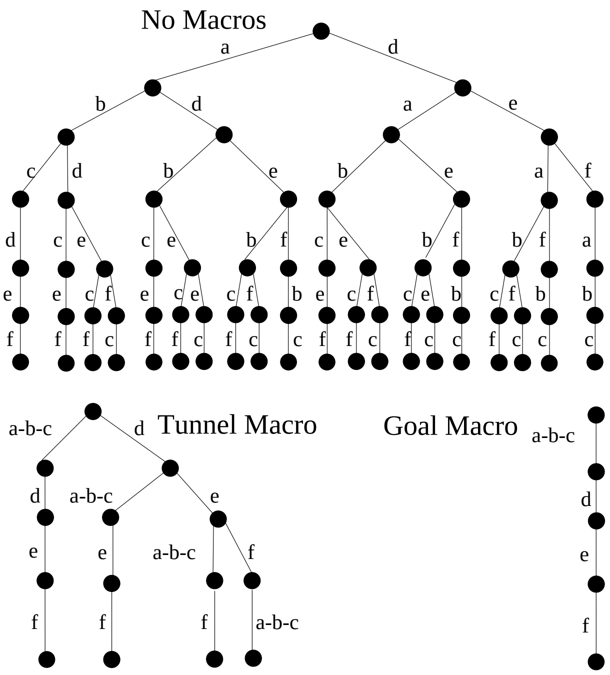


Fig. 5. The impact of macro moves

Figure 5 shows the dramatic impact this has on the search. At each node in the figure, the individual moves of the stone are considered. There are two stones that can each make a sequence of 3 moves, a-b-c for the first stone, and d-e-f for the second stone. The top tree in Figure 5 shows the search tree with no macro moves; essentially all moves are tried, whenever possible, in all possible variations. The left lower tree shows the search tree, if a-b-c is a tunnel macro and the lower right tree if a-b-c is goal macro.

4.5 Move Ordering

We have experimented with ordering the moves at interior nodes of the search. For IDA*, move ordering makes no difference to the search, except for the last iteration. Since the last iteration is aborted once the solution is found, it can make a big difference in performance if the solution is found earlier rather than later ([RM94] comment on the effectiveness of move ordering in single-agent search). One could argue that our inability to solve problem 48 (Figure 3) is solely a problem of move ordering. For this problem, we have the correct lower bound – it is just a matter of finding the right sequence of moves.

We are currently using a move ordering schema that we call *inertia*. Looking at the solution for problem 1 (Figure 1), one observes that there are long runs where the same stone is repeatedly pushed. Hence, moves are ordered to preserve the *inertia* of the previous move – move the same stone in the same direction if possible.

5 Experimental Results

Given 20 million nodes of search effort, our program can currently solve 16 problems. Table 2 shows these problems and contains in the second column the number nodes needed to find a solution (all search enhancements enabled).

These results illustrate just how difficult Sokoban really is. Even with a good lower bound heuristic and many enhancements to dramatically reduce the search cost, most problems are still too difficult to solve.

The later columns in table 2 attempt to quantify the benefits of the various enhancements made to IDA*. The table shows the results for IDA* using minimum matching enhanced with: a transposition table (128k entries – TT), deadlock table (5x4 region – DT), macro moves (goal and tunnel macros – MM), linear conflicts and backout enhancements (CB), and inertia move ordering (IN). The ALL column is the number of nodes searched by *Rolling Stone* with all the above features enabled. The columns thereafter show the tree size when one of these features is disabled.

These experiments highlight several interesting points:

1. Because of macro moves, the size of the search tree for problem 1 is *smaller* than the solution path length!

Problem	ALL	ALL-TT	ALL-DT	ALL-MM	ALL-CB	ALL-IN
1	61	61	70	116	61	160
2	1,646	1,036,503	6,855	370,213	>20,000,000	1,798
3	876	42,732	15,225	16,289	26,940	1,470
4	213,670	>20,000,000	>20,000,000	>20,000,000	>20,000,000	743,639
6	10,004	>20,000,000	10,846	>20,000,000	11,393	10,003
7	88,890	>20,000,000	6,166,124	744,912	253,404	77,539
17	126,121	>20,000,000	155,506	>20,000,000	>20,000,000	133,439
38	1,063,178	>20,000,000	2,958,995	8,999,852	1,267,181	450,017
51	125,413	>20,000,000	182,125	19,817,875	>20,000,000	103,021
63	256,835	>20,000,000	415,011	>20,000,000	>20,000,000	6,233,537
65	4,203,390	>20,000,000	6,438,584	13,561,416	>20,000,000	>20,000,000
78	76	76	76	154	1,195	1,902
80	237	237	237	627	>20,000,000	2,524
81	11,887,844	>20,000,000	>20,000,000	>20,000,000	>20,000,000	7,864,587
82	1,167,457	>20,000,000	1,989,577	>20,000,000	1,359,703	1,101,122
83	200	232	200	764	13,003	318

Table 2. Experimental Data

2. The program can find deep solutions with nominal depth. For example, the solution to problem 78 is 136 moves, and yet it is found by building a tree that is only 64 levels deep!
3. Problem 63 has a solution length of 431 moves and yet it is found with a search of only 257,000 nodes.
4. Transpositions tables are much more effective than seen in other single-agent and two-player games. For example, removing transposition tables for problem 6 increases the search by more than a factor of 2000!
5. Without the improvements of the lower-bound estimator (linear and backout conflict), the search tree for problem 80 increases by over a factor of 84,000! All the other improvements on the other hand have no or only minor effect on the search tree size.
6. Each of the enhancements can have a dramatic impact on the search tree size (depending on the problem).

Rolling Stone spends 90% of its execution time updating the lower bound. Clearly this is an area requiring further attention.

6 Enhancing the Current Program

Our program is still in its infancy and our list of things to experiment with is long. The following details some of the ways we intend to extend our implementation.

- To help detect larger deadlocks, we propose using localized searches that prove that for a certain stone combination no solution exists to push them to goals. It remains to be investigated if the search effort spent in testing for deadlocks will be offset by the savings gained from avoiding sub-trees with no solution.

– Our version of IDA* considers all legal moves in a position (modulo goal macros). For many problems, *local* searches make more sense. Typically, a man rearranges some stones in a region. Once done, it then moves on to another region. It makes sense to do local searches rather than global searches. A challenge here will be to preserve the optimality of solutions.
– The idea of *partition search* may be useful for Sokoban [Gin96]. For example, partition search could be used to discover previously seen deadlock states, where irrelevant stones are in different positions.
– A pre-search analysis of a problem can reveal constraints that can be used throughout the search. For example, in Figure 1 the stone on *Ch* cannot move to a goal until the stone on *Fh* is out of the way. Knowing that this is a prerequisite for *Ch* to move, there is no point in even considering legal moves for that stone until the right opportunity.
– So far, we have constrained our work by requiring an optimal solution. Introducing non-optimality allows us to be more aggressive in the types of macros we might use and in estimating lower bounds.
– Looking at the solution for Figure 1, one quickly discovers that having placed one stone into a goal, other stones follow similar paths. This is a recurring theme in many of the test problems. We are investigating dynamically learning repeated sequences of moves and modifying the search to treat them as macros.
– Sokoban can also be solved using a backward search. The search can start with all the stones on goal nodes. Now the man *pulls* stones instead of pushing them. The backward search may be useful for discovering some properties of the correct order that stones must be placed in the goal area(s) (the inverse of how a backward search can pull them out). This is an interesting approach that needs further consideration.

7 Conclusions and Future Work

Sokoban is a challenging puzzle – for both man and machine. The traditional enhanced single-agent search algorithms seem inadequate to solve the entire 90-problem test suite, even with their dramatic impact on the search tree size.

The property of deadlocks contained in a search space adds considerable complexity to the search. Since deadlock situations are an important consideration in real world applications, the notion of deadlock needs further attention. Deadlock tables are beneficial but inadequate to handle these situations. Further work is needed to identify when deadlocks are likely to occur and either avoid them (if possible) or invest the resources (search) to verify their existence. The problem of deadlocks is critical for any real-time application.

8 Acknowledgments

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. This paper benefited from interactions with Yngvi Bjornsson, John Buchanan, Joe Culberson, Roel van der Goot, Ian Parsons and Aske Plaat.

References

[CS96] J. Culberson and J. Schaeffer. Searching with pattern databases. In G. McCalla, editor, *Advances in Artificial Intelligence*, pages 402–416. Springer-Verlag, 1996. Proceedings of CSCSI'95.

[Cul97] J. Culberson. Sokoban is PSPACE-complete. Technical Report TR 97-02, Dept. of Computing Science, University of Alberta, 1997. Also: http://web.cs.ualberta.ca/˜joe/Preprints/Sokoban.

[DZ95] D. Dor and U. Zwick. SOKOBAN and other motion planing problems, 1995. At: http://www.math.tau.ac.il/˜ddorit.

[Gin96] M. Ginsberg. Partition search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-96)*, pages 228–233, 1996.

[HMY92] O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, 1992.

[Kle67] M Klein. A primal method for minimal cost flows. *Management Science*, 14:205–220, 1967.

[Kor85a] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kor85b] R.E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[Kor90] R.E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2–3):189–211, 1990.

[Kor97] R.E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI National Conference*, pages 700–705, 1997.

[Kuh55] H.W. Kuhn. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, pages 83–98, 1955.

[RM94] A. Reinefeld and T.A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, July 1994.

[SA77] D. Slate and L. Atkin. Chess 4.5 — The Northwestern University chess program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118, New York, 1977. Springer-Verlag.

[Wil88] G. Wilfong. Motion planning in the presence of movable obstacles. In *4th ACM Symposium on Computational Geometry*, pages 279–288, 1988.