Sokoban: A Challenging Single-Agent Search Problem

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

Abstract

Sokoban is a challenging single-player game – for both man and machine. The simplicity of the rules belies the complexity of the game. This paper describes our program, *Rolling Stone*, a first attempt to solve Sokoban problems. Adapting and extending the standard single-agent search techniques in the literature, we are able to optimally solve 12 of 90 problems from a standard test suite. This result demonstrates how difficult a game Sokoban really is for computers to solve, and underlines the need for more sophisticated search techniques, including planning.

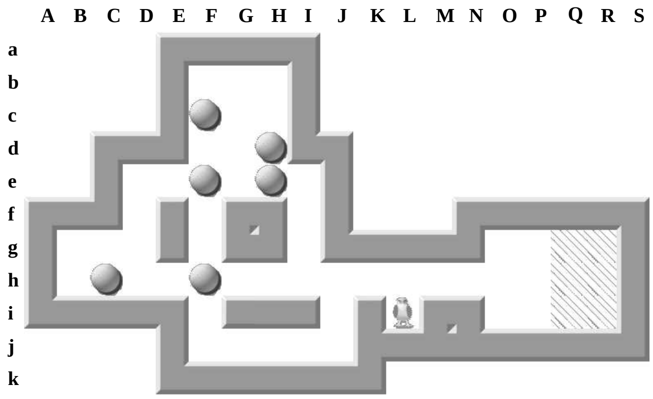
Keywords: single agent search, heuristic search, Sokoban, macro moves, deadlocks

1 Introduction

Sokoban is a popular one-player computer 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 easy 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



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

Figure 1: Sokoban problem 1 with one solution

"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 showing who has solved which problems and how efficient their solution is (also at http://xsokoban.lcs.mit.edu/xsokoban.html). Thus solving a problem is only part of the satisfaction; improving on your solution is equally important.

Sokoban has been shown to be NP-hard [Culberson, 1997; Dor and Zwick, 1995]. [Dor and Zwick, 1995] 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 [Wilfong, 1988]. 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.

In this paper, we adapt and enhance the standard single-agent search techniques to try to solve Sokoban problems. Our program, *Rolling Stone*, is able to optimally solve 12 of the 90 benchmark problems. Although this sounds rather poor, to the best of our knowledge 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 so as to be effectively unsolvable using standard single-agent search techniques.

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* [Korf, 1985a], stone pushes change the lower bound estimate of the solution length by at most one. 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).

2 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. In this section, we argue that the complexity of Sokoban works against the standard search techniques in the literature.

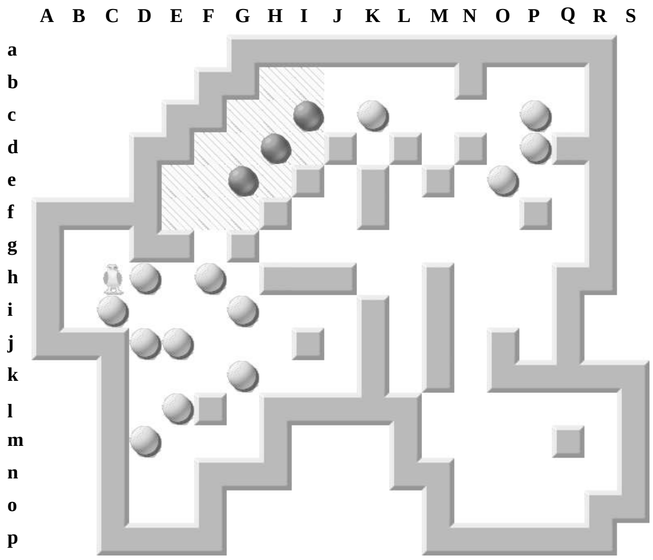


Figure 2: Sokoban problem 50

2.1 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 96 stone pushes (see section 3), whereas the best human solution required 374 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 very difficult to get a reasonable bound.

2.2 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). In Sokoban, moves can lead to states that provably cannot lead to solutions. We call these states *deadlocks* because one or more stones will never be able to progress to the goal. Deadlocks can be trivial as, for example, moving a stone into a corner (in Figure 1, moving *Hc-Hb*[1] creates a deadlock state; the man can never get behind the stone to push it out). Others involve interaction with other stones (in Figure 1, moving *Fe-Fd*[1] creates a deadlock). Some deadlocks can be

[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.

wide ranging and quite subtle, involving complex interactions of stones over a large portion of the playing area. Any programming solution to Sokoban must be able to detect deadlock states so that unnecessary search can be curtailed.

2.3 Large Branching Factor

Sliding tile puzzles have a maximum branching factor of four. Rubik's Cube has a branching factor of 18. In contrast, each stone in Sokoban has a maximum of four possible moves. In the set of 90 problems, the number of stones ranges from 6 to 34. With 34 stones, the maximum branching factor is 136, although undoubtedly in most positions the effective branching factor would be somewhat less than that. Nevertheless, the large branching factor severely limits the search depth that can be reached.

The large branching factor gives rise to a surprising result. Consider problem 51 (Figure 3). Our program computes the lower bound as 118 moves. Since human players have solved it in 118 moves, we can conclude that the optimal solution requires exactly 118 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 51 is very difficult to solve because of the large branching factor. Although IDA* will never make a non-optimal move, 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, the program builds an exponentially large tree. Our program currently cannot solve problem 51.

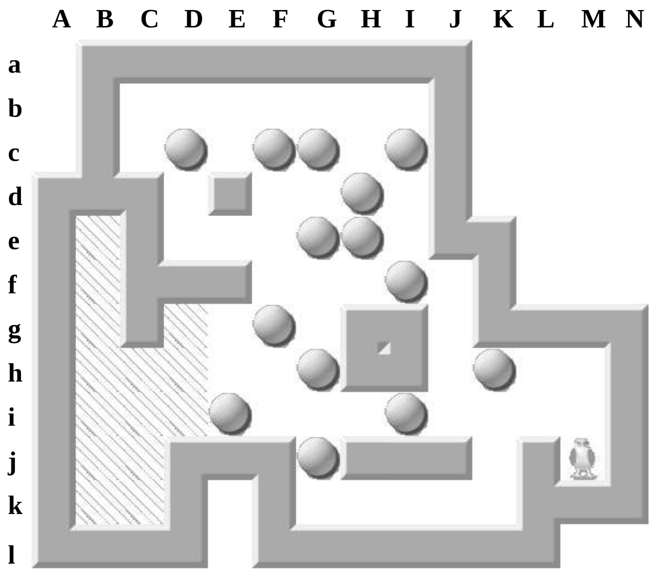


Figure 3: Sokoban problem 51

3 Towards Solving Sokoban

Although we believe that standard search algorithms, such as IDA*, will be inadequate to 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.

3.1 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, 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, then, 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 [Kuhn, 1955]. Given n nodes in a graph and m edges, then 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 only one stone has changed its weight to the goals. This requires finding a negative cost cycle [Klein, 1967] 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

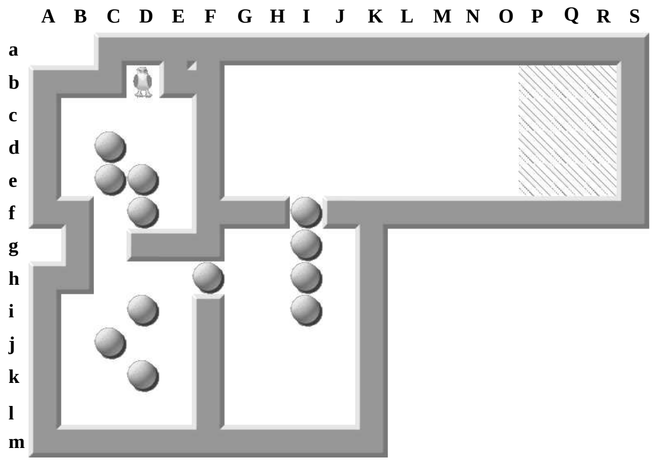


Figure 4: Sokoban problem 80

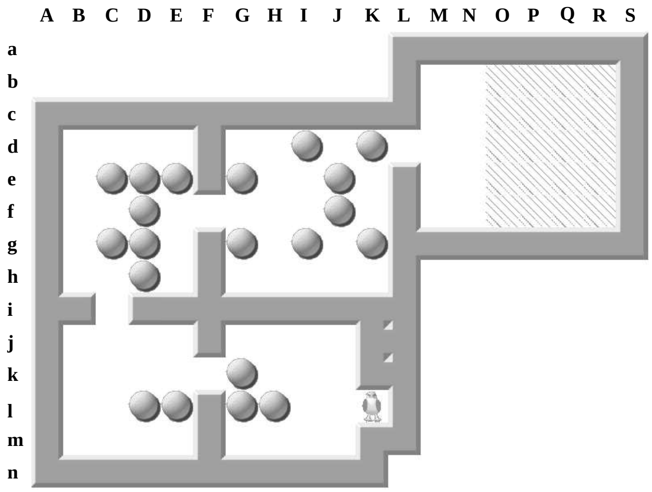


Figure 5: Sokoban problem 4

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.

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 [Hansson *et al.*, 1992]. Problem 80 (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, an additional two 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 the man. This is illustrated by problem 4 in Figure 5. The stones on *Ge*, *Gg*, *Dh* and *El* have a *backout conflict*. Consider the stone on *Gg*, while pretending there are no other stones on the board. The man must move the stone to a room entrance (*Gg-Gf*), push it out of the room (*Gf-Ff-Ef-Df*), and then push it back into the room it came from (*Df-Ef-Gf-Hf*). 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.

Table 1 shows the effectiveness of our lower bound estimate. The table shows the lower bound achieved by minimum matching, inclusion of the linear conflicts enhancement, inclusion of the backout enhancement, and the combination of all three features. The upper bound 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. The table is sorted according to the last column, 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.

3.2 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 [Slate and Atkin, 1977]. 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 [Reinefeld and Marsland, 1994].

One subtlety of Sokoban is that saving exact positions in the transposition 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

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

#	MM	+LC	+BO	ALL	UB	Diff
51	118	118	118	118	118	0
55	118	118	120	120	120	0
78	134	134	136	136	136	0
53	186	186	186	186	186	0
83	190	194	190	194	194	0
48	200	200	200	200	200	0
80	219	225	225	231	231	0
4	331	331	355	355	355	0
1	95	95	95	95	97	2
2	119	119	129	129	131	2
3	128	132	128	132	134	2
58	189	189	197	197	199	2
6	104	106	104	106	110	4
5	135	137	137	139	143	4
60	148	148	148	148	152	4
70	329	329	329	329	333	4
63	425	427	425	427	431	4
73	433	433	437	437	441	4
84	147	147	149	149	155	6
81	167	167	167	167	173	6
38	73	73	73	73	81	8
7	80	80	80	80	88	8
82	131	135	131	135	143	8
79	164	166	164	166	174	8
65	181	185	199	203	211	8
12	206	206	206	206	214	8
57	215	217	215	217	225	8
9	215	217	227	229	237	8
14	231	231	231	231	239	8
62	235	235	237	237	245	8
72	284	288	284	288	296	8
54	177	177	177	177	187	10
56	191	191	193	193	203	10
47	197	199	197	199	209	10
8	220	220	220	220	230	10
27	351	353	351	353	363	10
86	122	122	122	122	134	12
44	167	167	167	167	179	12
76	192	194	192	194	206	12
17	121	121	201	201	213	12
59	218	218	218	218	230	12
43	132	132	132	132	146	14
87	221	221	221	221	235	14
71	290	294	290	294	308	14
40	310	310	310	310	324	14
77	360	360	360	360	374	14
35	362	364	362	364	378	14
36	501	507	501	507	521	14
34	152	154	152	154	170	16
41	201	203	219	221	237	16
45	274	276	282	284	300	16
19	278	280	282	286	302	16
22	306	308	306	308	324	16
20	302	304	444	446	462	16
18	90	90	106	106	124	18
21	123	127	127	131	149	18
13	220	220	220	220	238	18
31	228	232	228	232	250	18
64	331	331	367	367	385	18
25	326	330	364	368	386	18
90	436	436	442	442	460	18
10	494	496	494	494	512	18
49	96	96	104	104	124	20
42	208	208	208	208	228	20
61	241	243	241	243	263	20
28	284	286	284	286	308	22
68	317	319	319	321	343	22
39	650	652	650	652	674	22
46	219	219	223	223	247	24
67	367	369	375	377	401	24
23	286	286	424	424	448	24
32	111	113	111	113	139	26
16	160	162	160	162	188	26
85	303	303	303	303	329	26
24	442	442	516	518	544	26
15	94	96	94	96	124	28
33	140	140	150	150	180	30
89	345	349	349	353	383	30
11	197	201	201	207	241	34
75	261	263	261	263	297	34
26	149	149	157	157	195	38
29	124	124	122	122	166	44
74	158	160	168	170	214	44
37	220	220	242	242	294	52
52	365	367	365	367	429	62
88	306	308	314	316	390	74
30	357	359	357	359	465	106
66	185	187	185	187	341	154
69	207	209	211	213	443	230
50	96	96	96	96	374	278

Table 1: Lower bounds

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.

3.3 Deadlock Tables

Our initial attempt at avoiding deadlock was to hand code into *Rolling Stone* a set of tests for simple deadlock patterns. This quickly proved to be of limited value, 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 what 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 very efficient. For our experiments, we built a deadlock table for a region of 5x4 squares (4.6 megabytes in size).

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. Figure 6 shows how the 5x4 deadlock table is mapped on to square Fg, with the stone being pushed in from Fh. Note that the table can be used to cover other regions as well[3]. To maximize the usage of the table, reflections 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

[3] We are currently using a second, smaller table, covering a slightly different region.

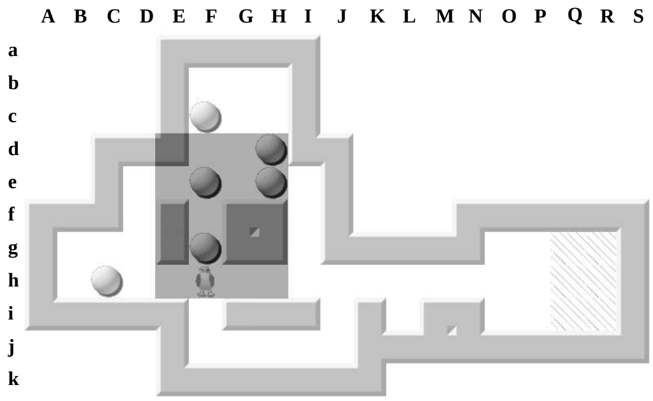


Figure 6: Placement of the 5x4 deadlock table after move *Fh-Fg* in problem 1

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 is a mistake, but if the square is a goal node, then the deadlock created is irrelevant.

3.4 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 ([Reinefeld and Marsland, 1994] comment on the effectiveness of move ordering in single-agent search). One could argue that our inability to solve problem 51 (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 man in the same direction if possible.

3.5 Macro Moves

Macro moves have been described in the literature [Korf, 1985b]. Although they are typically associated with non-optimal problem solving, we have chosen to investigate a series of macro moves that preserve solution optimality. We implemented the following two macros in *Rolling Stone*.

Tunnel Macros: In Figure 1, consider the consequences of 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 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 all the goals (extended as far back as the entrance to the room is one square wide), 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 *Mh* from the left, 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. 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.

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).

Figure 7 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 3 moves, *a*, *b*, *c* for the first stone, and *d*, *e*, *f* for the second stone. The top tree in Figure 7 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.

3.6 Goal Ordering

Since our deadlock tables exclude goal squares, special care has to be taken to avoid deadlocks in goal areas.

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

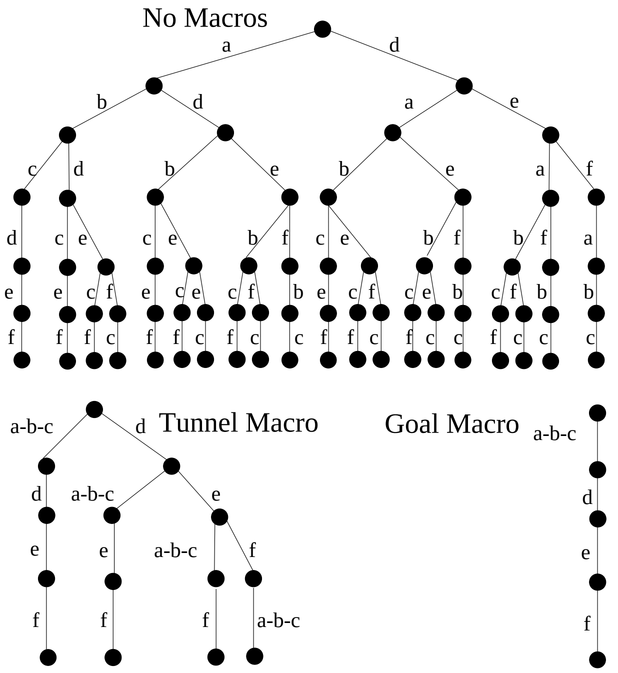


Figure 7: The impact of macro moves

Currently, we implemented a goal prioritization that ensures that the goal macros are moving the stones to the right goal squares at the right moment. We do not have a general implementation yet and can only handle the special case where all the goals are in one room with a single entrance. A generalization of this is discussed in Section 5.

4 Experimental Results

Table 2 shows the results of running *Rolling Stone* on the 90-position test suite. The table shows the minimum matching lower bound with all enhancements (Best LB), the last iteration of IDA* when the search was stopped (Final Iter.), the upper bound of the solution length as given by humans (Upper Bound), and the difference between final iteration and upper bound (Difference) as a rough indicator of how close we are to solving each problem. All problems were searched for up to 10,000,000 nodes.

The last column (Result) gives the results of the search. A search is either *open* (the default), the search has narrowed the range of possible solution values, *length solved* (shown by an asterisk), the optimal solution length has been found, or *solved*, where the solution length and solution moves have been found (the size of the search tree is given). Of the 90 problems, 12 have been solved and 17 have their optimal solution length computed.

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. Looking at the results, one can see that some problems (such as problem 50) aren't going to be solved by our current implementation.

Table 3 attempts 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. The search was stopped if it reached 20 million nodes without finding a solution. Only the 12 problems that *Rolling Stone* can solve are included.

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!
2. The program can find very deep solutions with nominal depth. For example, the solution to problem 4 is 355 moves, and yet it is found by building a tree that is only 185 moves deep!
3. 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 at least a factor of 5,650!

Each of our enhancements has 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.

5 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.

- While the deadlock tables have been beneficial to *Rolling Stone*'s performance, the 5x4 area is insufficient to detect many frequently occurring deadlocks. Although it is possible to build a 5X5 table (given sufficient computing resources), it is unreasonable to expect a larger table. The current table contains all possible combinations of stones, walls and empty squares. However, the Sokoban problems are not randomly created – the positions have structure. To help detect larger deadlocks, we propose having a deadlock table of rooms. The table

#	Best LB	Final Iter.	Upper Bound	Differ- ence	Result
1	95	97	97	0	73
2	129	131	131	0	1,646
3	132	134	134	0	877
4	355	355	355	0	230,747
5	139	141	143	2	
6	106	110	110	0	3541
7	80	88	88	0	4,635,479
8	220	224	230	6	
9	229	235	237	2	
10	494	494	512	18	
11	207	221	241	20	
12	206	210	214	4	
13	220	220	238	18	
14	231	233	239	6	
15	96	102	124	22	
16	162	168	188	20	
17	201	213	213	0	6,052,056
18	106	112	124	12	
19	286	290	302	12	
20	446	450	462	12	
21	131	143	149	6	
22	308	310	324	14	
23	424	424	448	24	
24	518	520	544	24	
25	368	370	386	16	
26	157	171	195	24	
27	353	357	363	6	
28	286	290	308	18	
29	122	130	166	36	
30	359	381	465	84	
31	232	236	250	14	
32	113	129	139	10	
33	150	156	180	24	
34	154	162	170	8	
35	364	368	378	10	
36	507	507	521	14	
37	242	244	294	50	
38	73	81	81	0	102,667
39	652	660	674	14	
40	310	312	324	12	
41	221	223	237	14	
42	208	208	228	20	
43	132	138	146	8	
44	167	169	179	10	
45	284	284	300	16	

#	Best LB	Final Iter.	Upper Bound	Differ- ence	Result
46	223	233	247	14	
47	199	203	209	6	
48	200	200	200	0	*
49	104	122	124	2	
50	96	98	374	276	
51	118	118	118	0	*
52	367	385	429	44	
53	186	186	186	0	*
54	177	177	187	10	
55	120	120	120	0	*
56	193	193	203	10	
57	217	223	225	2	
58	197	197	199	2	
59	218	220	230	10	
60	148	148	152	4	
61	243	247	263	16	
62	237	239	245	6	
63	427	427	431	4	
64	367	373	385	12	
65	203	209	211	2	
66	187	189	341	152	
67	377	387	401	14	
68	321	325	343	18	
69	213	213	443	230	
70	329	329	333	4	
71	294	294	308	14	
72	288	290	296	6	
73	437	439	441	2	
74	170	182	214	32	
75	263	277	297	20	
76	194	196	206	10	
77	360	362	374	12	
78	136	136	136	0	154
79	166	172	174	2	
80	231	231	231	0	476
81	167	173	173	0	*
82	135	143	143	0	3,690,687
83	194	194	194	0	388
84	149	151	155	4	
85	303	305	329	24	
86	122	128	134	6	
87	221	225	235	10	
88	316	318	390	72	
89	353	355	383	28	
90	442	450	460	10	

Table 2: Experimental Results

Problem	ALL	ALL-TT	ALL-DT	ALL-MM	ALL-CB	ALL-IN
1	73	73	82	116	73	312
2	1,646	1,036,503	6,868	14,247,731	>20,000,000	1,798
3	877	42,885	15,297	16,319	27,243	1,471
4	230,747	>20,000,000	>20,000,000	>20,000,000	>20,000,000	891,180
6	3,541	>20,000,000	4,520	>20,000,000	4,558	3,573
7	4,635,479	>20,000,000	>20,000,000	4,635,479	>20,000,000	5,274,788
17	6,052,056	>20,000,000	13,827,703	>20,000,000	>20,000,000	14,935,980
38	102,667	>20,000,000	8,784,788	102,667	734,592	>20,000,000
78	154	154	154	154	1,397	67,729
80	476	476	476	677	>20,000,000	67,808
82	3,690,687	>20,000,000	8,902,706	>20,000,000	14,002,015	3,681,313
83	388	1,727	412	764	23,117	696

Table 3: Experimental Data

would support a collection of commonly occurring room topologies and for each room type (including its shape, enclosed obstacles, number of room entrances, and the entrance positions) enumerate all possible deadlock scenarios. Although this does not solve the deadlock problem in general, it will allow us to find some deadlocks that we currently cannot.

- The goal macro is currently restricted to having all the goals in a single room. Most problems have goal nodes scattered throughout the puzzle or goal areas have several entrances, meaning the goal macros cannot be used. We have designed a new algorithm that allows us to handle all goal nodes, regardless of the location. It involves two pre-search computations. First, each collection of goal nodes must have a perimeter defined around them. When a stone moves to that perimeter, then the goal macro is invoked. The trick is to extend the perimeter back as far from the goals as possible. Second, many small searches are needed to identify any constraints on the ordering in which stones can be placed onto goal nodes. We believe that successfully implementing this algorithm will more than double the number of problems we can solve.

  One should also note, however, that although goal macros are a powerful tool that can be used to solve most of the Sokoban problems, there are some insidious problems (such as problem 50) where goal macros will effectively exclude the solution from the search. Clearly, more work needs to be done to identify the conditions under which goal macros do not apply.

- Our version of IDA* considers all legal moves in a position (modulo goal macros). For many problems, local *regional* searches make more sense. Typically, a man rearranges some stones in a region. Once done, then it 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 [Ginsberg, 1996]. 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 3 the stone on *Kh* cannot move to a goal until the stones on *Ei* and *Ii* are out of the way. Knowing that this is a pre-requisite for *Kh* to move, there is no point in even considering legal moves for that stone until the right opportunity.

- The preceding, of course, suggests that planning will be an essential component in any program that can successfully solve all 90 problems. Currently, we believe brute-force search is incapable of solving problem 50. A goal-driven search based on a plan is mandatory.

- 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. One thing we are investigating is 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 than 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.

6 Other Sokoban Programs

We know of several attempts to build a program to solve Sokoban problems, however Andrew Myers' program is the only one that appears to be successful (it has solved nine problems).

Myers writes [Myers, 1997] that his: "...program uses a breadth-first A* search, with a simpler heuristic to select the next state to examine. A compact transposition table stores the states. When the solver runs out of memory, it discards some states below the 10th percentile in moves made. This feature allows the program to handle levels like level 51. The solver tries to minimize both moves and pushes. It does not support macro moves.

"The heuristic estimates both the number of stone pushes and the number of man movements needed to complete the puzzle. The number of pushes is estimated more quickly but less accurately, taking advantage of the usual clustering of the goal spaces in one area of the board. The estimate has two parts: the number of moves and pushes needed to push a ball to the nearest goal square, and the number of pushes needed to push a ball to each goal square from the nearest *non-goal* square. In addition, the estimator compensates for the ball that is optimal for the man to push next. The estimate is summed quickly, using approximately 700K of pre-computed tables. The estimate does not consider linear conflicts, which would probably help. The heuristic is not monotonic; a conservative, monotonic estimate is used to discard suboptimal states.

"Deadlocks are automatically identified for 3x3 regions, and also for certain goal locations that can never be filled. A goal location can only be filled if in one of the four directions, the two immediately adjacent squares can be made empty. If a immovable ball is placed in either square, the state is deadlocked. An optional deadlock table allows easy specification of complex deadlock conditions by hand. However, the program does not attempt to automatically fill the deadlock table."

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. Currently we intend to push the established search techniques to their limit, before turning to other approaches.

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

[Culberson, 1997] Joe 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.

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

[Ginsberg, 1996] M. Ginsberg. Partition search. In *AAAI*, pages 228–233, 1996.

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

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

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

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

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

[Myers, 1997] A. Myers. March 19. *personal communication*, 1997.

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

[Slate and Atkin, 1977] 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.

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