Single-Agent Search in the Presence of Deadlocks

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

Abstract

Single-agent search is a powerful tool for solving a variety of applications. Most of the 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. In this paper we address the implications that arise when state transitions can lead to unsolvable (deadlock) states. Deadlock states are partially responsible for the failure of our attempts to solve positions in the game of Sokoban. In this paper, we introduce *pattern search*, a real-time learning algorithm that identifies the minimal conditions (pattern) necessary for a deadlock, and applies that knowledge to eliminate provably irrelevant parts of the search tree. Identification of deadlock patterns is equivalent to correcting the heuristic lower bound of a position to infinity. Generalizing pattern searches to find arbitrary lower bound increases yields a powerful new search enhancement. In the game of Sokoban, pattern searches result in a 15-fold reduction of the cost of each additional IDA* iteration.

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

Introduction

Single-agent search (A*) has been extensively studied in the literature. There are a plethora of enhancements to the basic algorithm that allows one to tailor the algorithms to the problem domains to maximize program performance. The result is an impressive reduction in the search effort required to solve challenging applications (see (Korf 1997) for a recent example). However, the applications used to illustrate the advances in single-agent search efficiency are "easy" in the sense that they have some (or all) of the following properties:

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

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

Sokoban is a popular one-player game. The rules of the game are quite simple. Littered throughout the playing area, consisting of rooms and passageways, 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 (Junghanns & Schaeffer 1998).

Since you cannot pull a stone, 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. However some require extensive analysis to prove their existence; the search trees may be so large that they are essentially unsolvable by traditional search methods.

Sokoban is a difficult domain for many reasons:

1) it has a complex lower-bound estimator ($O(n^3)$, given n goals (Kuhn 1955)),
2) the branching factor is large and variable (potentially over 100),
3) the solution may be very long (some problems require over 500 moves to solve optimally), and
4) some reachable states are unsolvable (deadlock).

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

Our previous attempts to solve Sokoban problems using standard single-agent search techniques are reported in (Junghanns & Schaeffer 1998). There, using our program *Rolling Stone*, we compare the different techniques and their usefulness with respect to the search efficiency when solving Sokoban problems. Even though each of the five standard single-agent search enhancements we investigated resulted in significant improvements (often several orders of magnitude in search-tree size reduction), at the time we were able to only solve 20 problems of a 90-problem test suite (http://xsokoban.lcs.mit.edu/xsokoban.html).

We concluded that the standard techniques are insufficient to make further progress in the domain of Sokoban. Additional search enhancements are needed to enable us to solve significantly more of the problems from the test set. Since large portions of the search are wasted searching problem configurations with deadlocks present, we speculated that the detection of these deadlocks could lead to significant efficiency gains. The techniques suggested in this paper are a direct result of those observations.

In this paper, we introduce a new search enhancement that dynamically finds deadlocks and improved lower bounds. *Pattern search* is a real-time learning algorithm that identifies the minimal conditions necessary for a deadlock, and applies that knowledge to eliminate provably irrelevant parts of the search tree. By devoting a portion of the search effort to learning properties about the search space, the program trades off search-tree size versus acquired knowledge. In the game of Sokoban, the additional knowledge gained by the pattern searches improves the program's search efficiency. The average growth rate of the tree is a factor of 15 times smaller per IDA* (Korf 1985) iteration. This results in 29 solved Sokoban problems, and significant progress towards solving many more.

Pattern Searches

In general, establishing the presence of deadlock can be quite involved. The deadlock may require as few as one and as many as all the stones on the board. Proving that a pattern of stones creates a deadlock will require a search to verify that no possible solution path exists. Ideally, having discovered a deadlock pattern, any state containing that pattern will now be assigned the correct lower bound of infinity.

This section describes our *pattern searches*. We describe how we prove the presence of deadlock by identifying that the properties needed to prevent deadlock are not present. A pattern search results in a minimal pattern which is saved and used throughout the search. In effect, the program learns the deadlock patterns and eliminates any search path leading to a position containing a deadlock pattern.

A deadlock implies a lower bound of infinity. While trying to prove/disprove a deadlock, we may be able to show that our lower bound is too low. Even if a deadlock is not present, we may uncover a pattern that allows the search to improve its admissible lower bound.

Basic Idea

By definition, a deadlock is a configuration of stones such that not all of the stones can reach a goal. If we make a move A-B, we might introduce a deadlock. If this deadlock was not present before the move, then the moved stone, now on square B, must be part of that pattern. This is the initial stone used for the pattern search. The pattern search will perform small searches with a subset of stones on the board to determine if a deadlock was introduced.

```
PatternSearch( From, To ) {
  clear TestMaze;
  set StonePath = {To};
  for( i=1; i <= MAX_PATTERN_SIZE
        AND NOT EffortLimit(); i++ ) {
    if( stone s on a square in StonePath )
      add closest s to TestMaze
    else if( stone s on a square in ManPath )
      add closest s to TestMaze
    else break;
    solution = PIDA*( TestMaze, SolLength,
          ManPath, StonePath );
    /* Test for a deadlock */
    if( solution == NO AND NOT EffortLimit() ) {
      GeneralizeAndAddPattern( TestMaze, infty );
      break;
    }
    /* Test for a lower bound increase */
    if( solution == YES ) {
      lb = LowerBound( TestMaze );
      if( SolLength > lb )
        GeneralizeAndAddPattern( TestMaze,
              SolLength - lb );
    }
  }
}
```

Figure 1: Pseudo Code for Pattern Searches

In the following, we will refer to two different mazes: the *original maze*, which is the maze with all the stones at the position after the move, and the *test maze* which will be used for the pattern searches.

A pattern search iterates on the number of stones in the test maze. We start by putting only the stone that was moved to B in the test maze. PIDA* (see below) is called to solve this test maze. It either returns in failure (no solution, hence deadlock), or it finds a solution. In the latter case, we are interested in the set of squares that are used by the stones and the man to effect the solution: the squares occupied by the stones(s) on their path to the goal(s) (StonePath), and the squares touched by the man while pushing the stone(s) to a goal(s) (ManPath). In effect, these sets of squares are preconditions for the solution to work.

The ManPath and StonePath are used to determine which stone from the original maze to include next in the test maze. A stone in the original maze on a square that is on one of the squares in ManPath or StonePath conflicts with the solution. The stone in StonePath closest to square B (the square the stone was moved to in the original maze) is included next. If such a stone does not exist, the stone on ManPath closest[1] to square A is used. If none of those exists, the pattern search returns without finding a deadlock.

After including the next stone, PIDA* is called again,

[1] *Closest* is always with respect to the distance of either the stone or the man to the conflicting stone. These distance measures are possibly different due to the more restricted movement of the stones.

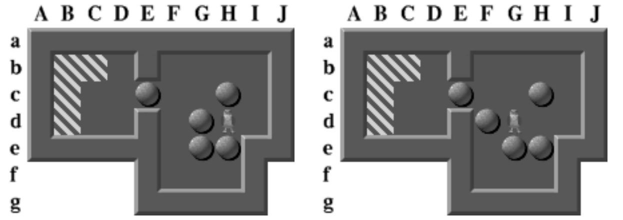

Figure 2: Deadlock example

returning with a solution determination and the two conflict sets. If deadlock has not been found, then the conflict sets are used to add another stone to the test maze. If any of the returning searches indicates a longer solution length than the lower bound estimate of the position, the current pattern is stored with a corresponding lower-bound increase. Figure 1 shows the pseudo code.

The notion of bit (stone) patterns is similar to the Method of Analogies (Adelson-Velskiy, Arlazarov, & Donskoy 1975). Pattern searches are a conflict-driven top-down proof of correctness, while the Method of Analogies is a bottom-up heuristic approximation.

Example

Figure 2 shows a simple position, before and after the move *Gd-Fd*. The question is whether this move introduces a deadlock. Figure 3 shows how the test maze is built. Since the last move ended up on square *Fd*, the test maze is initialized with this single stone (Figure 3a). A PIDA* search reveals a 5-move solution (*Fd-Fc-Ec-Dc-Cc-Bc*), and sets ManPath to the squares needed by the man (*Gd-Ge-Fe-Fd-Gd-Gc-Fc-Ec-Dc-Cc*), and StonePath to the squares used by the stone (*Fd-Fc-Ec-Dc-Cc-Bc*). Since there is a solution, we continue the pattern search.

The original maze has a stone on one of the squares that the stone moved over (square *Ec*) which now gets included in the test maze (Figure 3b). PIDA* will solve the test maze with the two stones and again find a solution. The ManPath is (*Gd-Gc-Fc-Ec-Dc-Dd-Cd-Cc-Dc-Ec-Fc-Gc-Gd-Ge-Fe-Fd-Gd-Gc-Fc-Ec-Dc-Cc*) and the StonePath is (*Ec-Dc-Cc-Cb Fd-Fc-Ec-Dc-Cc-Bc*). This time there are no stones in conflict with StonePath. However, there is a conflict with the ManPath, square *Ge*. This stone is added to the test maze (Figure 3c) and another search is commenced. A solution will be found, requiring a fourth stone to be added (Figure 3d).

The fourth call to PIDA* will return no solution and announce a deadlock with this pattern of four stones. Identifying the critical stones to examine has been driven by whether they conflict with a potential solution. The irrelevant parts of the maze (such as the stone on *Hc*) are ignored.

Generalizing the Patterns

The fewer stones in a deadlock pattern, the more likely it will match an arbitrary position and be used to eliminate futile branches of the search. A *minimal deadlock pattern* is a deadlock pattern from which no stone can be removed without making the remaining pattern solvable. The attentive reader will have noticed that only three stones are needed to guarantee deadlock in Figure 3; the stone on *Ec* is unnecessary. Before saving the deadlock pattern, our program will attempt to minimize the number of stones in it.

The deadlock set minimization routine takes an *N*-stone pattern and considers each of the possible *N-1*-stone sub-patterns. Each of the *N-1*-stone sub-patterns is searched to verify whether removing that stone preserves the deadlock. If the deadlock still exists, then the removed stone was not part of the minimal deadlock set and is removed from the deadlock pattern.

Customizing IDA* for Pattern Searches

If the pattern searches used the same IDA* procedure and lower bound estimator as in *Rolling Stone*, the search would be prohibitively large and slow. Instead, we use a special version of IDA* (PIDA*) that is customized for pattern searches, allowing for additional optimizations that dramatically improve the search efficiency. By relaxing the rules of Sokoban and introducing new goal criteria, the resulting search will be more efficient and will still return an admissible lower bound on the solution.

One optimization is to remove stones from the test maze once they reach a goal square or a man-reachable square. This comes from the observation that most deadlocks result in a number of stones getting *crowded* together. Hence, if a stone "breaks free", we assume we no longer need to consider it in that search sub-tree. Another optimization is to relax what we consider a goal state. Now, goal states are also positions where the man can reach all squares and at least one conflict with the current StonePath was found already.

These shortcuts simplify the search leading to large savings in the cost of a pattern search (from thousands of nodes to an average of 50). However, this comes at the cost of possibly missing a deadlock. In practice, the reduced search effort more than compensates for the few missed opportunities.

Since stones get removed from the board when they reach a goal square, the best lower bound heuristic is not appropriate (see (Junghanns & Schaeffer 1998)). A cheaper heuristic can be used: the sum of the shortest distances of each stone to its closest goal. When a stone moves, this lower bound is easily updated. This results in large savings in the cost per node compared to the original $O(n^3)$ lower bound. Since the number of stones is small in a pattern search, most search-related routines are fast, because their cost depends on the number of stones in the maze.

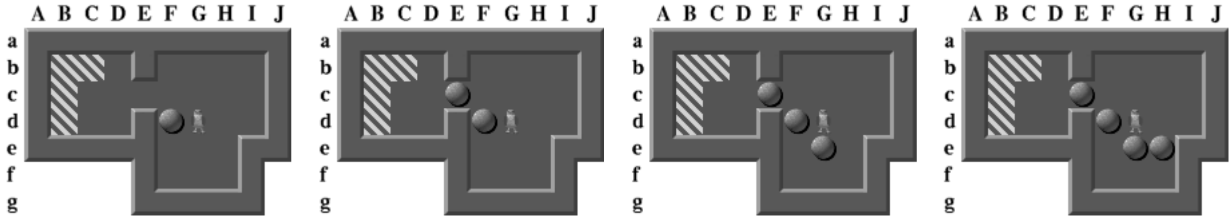

Figure 3: Sequence of test mazes as passed to PIDA* (a, b, c and d)

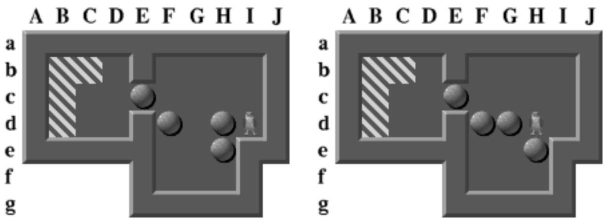

Figure 4: Penalty Example

Tradeoffs

Pattern searches can be costly. There are three main factors involved in their cost: the frequency of the pattern searches, a bound on the size of a pattern search, and a bound on the deadlock pattern size.

Frequency of Pattern Searches: We cannot afford to do a pattern search at every node in the IDA* search. We use some simple heuristics for deciding when to invest in a deadlock search.

The pattern search is always done for a node for which a deadlock search has not been previously done before (as retrieved from the transposition table) and the amount of effort spent below that node on a previous iteration exceeds a threshold. For our experiments, we use a threshold of 50 nodes, a number that reflects the size of a typical pattern search. Furthermore, if a stone is pushed onto an articulation point of the underlying graph structure of the maze, if the stone blocks an area for the man that was previously accessible, or if the stone pushed has no more legal moves, the pattern search is executed. These heuristics ensure the execution of pattern searches where the possibility of erroneous lower bounds is high.

Size of the Pattern Search: Pattern searches are restricted to a maximum effort of 1000 nodes. If the threshold is reached, the search is aborted.

Pattern Size: Deadlock patterns are restricted to 8 stones. This is an artificial limitation, but we have not fully explored the tradeoffs of finding larger deadlock patterns, versus the effort required to find them.

Generalizing Pattern Searches

The presence of a deadlock pattern in a position means the lower bound increases to infinity. Can we find patterns that allow us to increase the lower bound by an arbitrary amount, not just infinity?

Assume there are three stones in the test maze and PIDA* starts its first iteration but fails to find a solution. Hence PIDA* proved that this pattern cannot be solved with the number of moves that the heuristic lower bound indicated. In other words, the lower bound is wrong.

Some of the shortcuts used in the pattern searches are not appropriate when searching for a lower bound increase. Thus a second search routine is used to look for patterns that allow for arbitrary lower bound increases. If the first pattern search fails (looking for a deadlock), the second pattern search is executed, possibly finding a pattern that allows us to improve the lower bound.

In Figure 4, after the move *Hd-Gd* a pattern search looking for a deadlock will fail. A pattern search looking to improve the lower bound will uncover that the solution requires the non-optimal moves *Fd-Fe* and *Gd-Hd*, proving that the lower bound is off by four moves.

Storage and Retrieval of Patterns

To incorporate the deadlock and penalty patterns into the search, we need to save the patterns found and use them to match positions in the search. The pattern matching is complicated by the fact that you need to match not only the stones, but also the man position. With each pattern of stones the squares are stored which the man in the test maze cannot reach. To increase the usefulness of the information found by the pattern searches, we use the *multi-insert* technique. Instead of the root node only, the top two ply of the pattern search nodes are stored.

To match a pattern, the test maze must have the stones in the same places as the pattern and the man must not be able to reach any of the non-reachable squares stored together with the pattern. Since several patterns might match, stones that were used to match a pattern are not used when looking for a second pattern to match, to avoid double penalization. To maximize the penalties, the pattern matching starts with the highest penalty patterns.

This is similar to Ginsberg's Partition Search idea (Ginsberg 1996) where the entries of a hash table were generalized to hold information about sets of problem states. In *Rolling Stone* a pattern is the information about the lower-bound increase of the set of problem states in which this pattern is present.

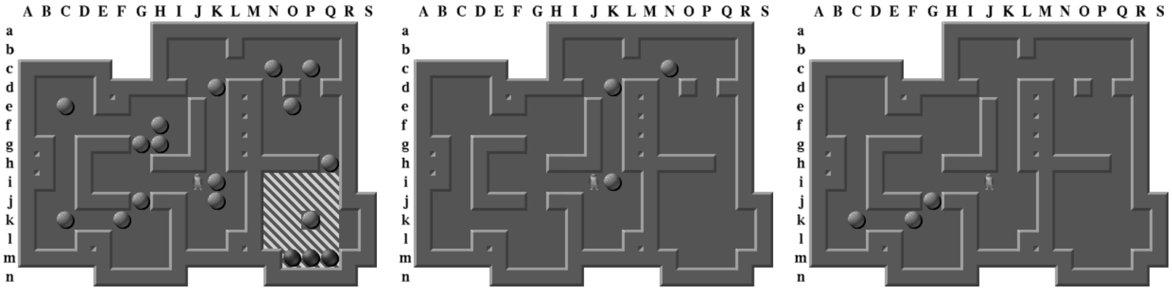

Figure 5: Maze #30 receives a penalty of 38 (24+14) after 2 patterns were matched

Experimental Results

Figure 5 shows maze #30 with a stone configuration that arises during the search. Two penalty patterns were successfully matched, resulting in an increase of 38 (14+28) to the lower bound.

Given 20 million nodes of search effort, our program can currently solve 29 problems of the 90 problem test suite. Without the pattern searches, only 22 problems can be solved[2]. Table 1 shows the results of searching these problems. Each column is labeled according to which of the three features is enabled: penalty searches (pen), deadlock searches (dl) and multi-insert (mi), where + and - mean enabled and disabled, respectively. Two node numbers are given: the *IDA** nodes and the *IDA* + PIDA** nodes.

Except for the small searches, the cost of performing the additional PIDA* searches is offset by the reduction in the IDA* search nodes. Problem 53 is an example. Previously, with 20,000,000 nodes of search, we were unable to solve this problem. Now the search is accomplished with only 177 IDA* nodes and a total of 1,229 nodes. Clearly, the pattern searches dominate the search cost, but the knowledge uncovered allows us to solve the problem where we failed previously.

Analysis of the data shows that the average growth rate of the search tree from iteration to iteration in an IDA* search decreased from 84,669 to 5,559 due to the pattern searches. Although this represents a significant reduction in search effort (a factor of 15 per iteration), it demonstrates how resistant the problem is to search. Decreasing the growth rate of the search tree size generally increases the number of iterations that the main IDA* search can perform in the same time. For example, on 13 of the remaining 61 problems 3 or more additional IDA* iterations were accomplished (the maximum was 9 extra iterations). Since the average increase of the tree size to the next iteration is 5,559, even 3 iterations are significant improvements.

Pattern searches are a gamble: you invest search effort (PIDA* nodes) expecting to find useful knowledge. A failed pattern search costs roughly 50 nodes. A successful pattern search typically costs over 1,000 nodes, because of the additional difficulty of the search and the cost of minimizing the pattern. Only 12% of the pattern searches are successful at discovering something useful. Although this sounds low, the results show the value of the discovered knowledge. Problem #21 is one example of where the gamble does not pay off. Even though the tree size (IDA*) is reduced to about 33%, including the PIDA* nodes quadruples the total number of nodes searched.

The results reported here are not the best numbers that can be achieved. There are numerous parameters in the search, each of which can be tuned for maximal performance. In Table 1, the PIDA* nodes dominate the cost of the search for some problems. Some additional heuristics for deciding when to do pattern searches can result in further improvements in the search efficiency.

Furthermore, examination of the results shows that lower-bound increases are more beneficial then the deadlock patterns. More is to be gained by improving the lower bound than by identifying deadlock states.

Conclusions and Future Work

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

The property of deadlocks in a search space adds considerable complexity to the search. The previously introduced deadlock tables (Junghanns & Schaeffer 1998) are beneficial for local deadlock detection, but inadequate to handle non-trivial situations. Pattern searches can detect global deadlock scenarios and are able to improve the lower bound considerably, resulting in a substantial improvement in search efficiency.

Further work is needed to identify when deadlocks are likely to occur and either avoid them or invest the resources to verify their existence. Detecting deadlocks is critical to any real-time application.

The pattern searches were based on demonstrating whether there existed a scenario by which all the stones in a position subset could reach their goals, and in how

[2] We previously reported 20 problems solved. Increasing the utility of the goal macros allowed the standard version to solve two more problems.

#	-dl -pen -mi	+dl -pen -mi		+dl +pen -mi		+dl +pen +mi	
	IDA*	IDA*	IDA*+PIDA*	IDA*	IDA*+PIDA*	IDA*	IDA*+PIDA*
1	53	53	125	50	410	50	412
2	224	210	1,137	149	3,090	149	3,329
3	393	188	3,734	97	7,231	97	11,753
4	394	310	3,488	187	1,878	187	1,879
5	1,768,356	16,071	110,709	2,813	61,608	2,664	66,658
6	207	168	427	139	2,234	139	3,008
7	30,118	20,833	138,685	2,818	61,879	2,743	61,017
9	198,667	121,868	207,833	7,619	162,999	8,647	160,833
17	10,108	1,676	9,512	1,103	23,771	821	22,588
21	374,843	330,273	2,112,792	145,939	1,668,636	109,913	1,231,217
38	312,017	104,255	128,451	75,244	117,770	75,244	118,835
43	> 20,000,000	> 13,431,042	> 20,000,000	13,834	2,223,154	10,661	2,337,712
51	50,675	97,299	99,754	133	2,564	133	6,042
53	> 20,000,000	177	368	177	1,184	177	1,229
55	> 20,000,000	> 13,198,858	> 20,000,000	4,527,930	12,482,548	4,462,920	12,301,510
57	3,078,112	215,541	521,533	176,413	780,398	79,792	608,261
62	127,434	821,533	872,317	2,036	21,395	440	18,711
63	3,137,313	415,401	2,293,790	25,200	1,086,424	18,024	742,490
65	1,333	1,333	2,272	980	3,848	1,355	4,242
68	> 20,000,000	> 19,519,926	> 20,000,000	203,966	8,214,930	145,528	6,260,529
70	> 20,000,000	> 9,189,442	> 20,000,000	> 619,739	> 20,000,000	185,820	3,639,142
72	> 20,000,000	> 13,142,686	> 20,000,000	49,279	145,276	1,701	49,962
73	> 20,000,000	> 19,011,026	> 20,000,000	29,586	45,309	29,586	45,318
78	75	75	267	75	882	75	882
79	4,474	3,799	5,504	1,970	13,163	1,957	10,555
80	2,430	109	2,981	98	9,555	98	10,534
81	305,185	16,655	50,344	1,991	28,502	1,908	21,333
82	162,517	151,720	717,656	230	32,395	471	49,323
83	1,198	90	2,883	90	7,204	90	7,271
$\sum$	>149,566,126	> 89,812,617	>127,286,562	> 5,889,885	> 47,210,237	5,141,390	27,796,575

Table 1: Experimental Data

many moves. There are other proof conditions that can be tried. One promising avenue for proving that deadlock is not being introduced is the reversible move. For example, assume that in position P move A-B is made. It may be easy to verify that there is a sequence of moves that effectively unmakes the move A-B (possibly as simple as B-A) resulting in all the stones and the man being back as they were in position P. If this can be shown, then this is a proof that deadlock was not introduced by A-B. This property can be verified with a search where the goal conditions are changed.

Although pattern searches can be enhanced to make them more efficient, it appears they are inadequate to successfully solve all 90 Sokoban test positions. This is the subject of ongoing research.

Acknowledgements

The authors would like to thank the German Academic Exchange Service, the Killam Foundation and the Natural Sciences and Engineering Research Council of Canada for their support. This paper benefited from interactions with Yngvi Bjornsson, Russ Greiner, Peter van Beek and from the referees' comments.

References

Adelson-Velskiy, G.; Arlazarov, V.; and Donskoy, M. 1975. Some methods of controlling the tree search in chess programs. *Artificial Intelligence* 6(4):361–371.

Ginsberg, M. 1996. Partition search. In *AAAI-96*, 228–233.

Junghanns, A., and Schaeffer, J. 1998. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In *Proceedings AI-98*. To appear in Springer-Verlag's *Lecture Notes in Computer Science* series.

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

Korf, R. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI–97*, 700–705.

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