Domain-Dependent Single-Agent Search Enhancements

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

Abstract

AI research has developed an extensive collection of methods to solve state-space problems. Using the challenging domain of Sokoban, this paper studies the effect of search enhancements on program performance. We show that the current state of the art in AI generally requires a large programming and research effort into domain-dependent methods to solve even moderately complex problems in such difficult domains. The application of domain-specific knowledge to exploit properties of the search space can result in large reductions in the size of the search tree, often several orders of magnitude per search enhancement. Understanding the effect of these enhancements on the search leads to a new taxonomy of search enhancements, and a new framework for developing single-agent search applications. This is used to illustrate the large gap between what is portrayed in the literature versus what is needed in practice.

Keywords: single-agent search, IDA*, Sokoban

1 Introduction

The AI research community has developed an impressive suite of techniques for solving state-space problems. These techniques range from general-purpose domain-independent methods such as A*, to domain-specific enhancements. There is a strong movement toward developing domain-independent methods to solve problems. While these approaches require minimal effort to specify a problem to be solved, the performance of these solvers is often limited, exceeding available resources on even simple problem instances. This requires the development of domain-dependent methods that exploit additional knowledge about the search space. These methods can greatly improve the efficiency of a search-based program, as measured in the size of the search tree needed to solve a problem instance.

This paper presents a study on solving challenging single-agent search problems for the domain of Sokoban. Sokoban is a one-player game and is of general interest as an instance of a robot motion planning problem [Dor and Zwick, 1995]. Sokoban is analogous to the problem of having a robot in a warehouse move specified goods from their current location to their final destination, subject to the topology of the warehouse and any obstacles in the way. Sokoban has been shown to be NP-hard [Culberson, 1997; Dor and Zwick, 1995].

Previously we reported on our attempts to solve Sokoban problems using the standard single-agent search techniques available in the literature [Junghanns and Schaeffer, 1998c]. When these proved inadequate, solving only 10 of a 90-problem test suite, new algorithms had to be developed to improve search efficiency [Junghanns and Schaeffer, 1998b; 1998a]. This allowed 47 problems to be optimally solved, or nearly so. Additional efforts have since increased this number to 52. The results here show the large gains achieved by adding application-dependent knowledge to our program *Rolling Stone*. With each enhancement, reductions of the search tree size by several orders of magnitude are possible.

Analyzing all the additions made to the Sokoban solver reveals that the most valuable search enhancements are based on search (both on-line and off-line) by improving the lower bound. We classify the search enhancements along several dimensions including their generality, computational model, completeness and admissibility. Not surprisingly, the more specific an enhancement is, the greater its impact on search performance.

When presented in the literature, single-agent search (usually IDA*) consists of a few lines of code. Most textbooks do not discuss search enhancements, other than cycle detection. In reality, non-trivial single-agent search problems require more extensive programming (and possibly research) effort. For example, achieving high performance at solving sliding tile puzzles requires enhancements such as cycle detection, pattern databases, move ordering and enhanced lower bound calculations [Culberson and Schaeffer, 1996]. In this paper, we outline a new framework for developing high-performance single-agent search programs.

This paper contains the following contributions:

1. A case study showing the evolution of a Sokoban

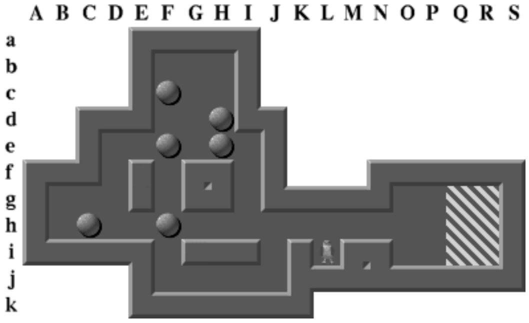

Figure 1: Problem #1 of the Test Set

solver's performance, beginning with a domain-independent solver and ending with a highly-tuned, application-dependent program.

2. A taxonomy of single-agent search enhancements.
3. A new framework for single-agent search, including search enhancements and their control functions.

2 Sokoban

Figure 1 shows a sample problem of Sokoban. The goal is simple: use the man to push all the stones in the maze to the shaded goal squares. Only one stone can be pushed at a time. These rather simple rules bely the difficulty of Sokoban problems, especially with respect to computer solutions. We identified several reasons why Sokoban is so difficult [Junghanns and Schaeffer, 1998c]:

- The graph underlying Sokoban problems is directed; some moves are not reversible. Consequently, there are *deadlock* states from which no solution is reachable. Deadlocks represent a challenge for anytime algorithms: when committing to a move, how can we make sure that no deadlock is introduced?
- The combination of long solution lengths (up to 674 stone pushes in the test set) and potentially large branching factors make Sokoban difficult for conventional search algorithms to solve. 20×20 Sokoban offers the challenge of a large search space ($\approx 10^{98}$).
- Sokoban solutions are inherently sequential; only limited parts of a solution are interchangeable. Subgoals are often interrelated and thus cannot be solved independently.
- A "simple", effective lower bound on the solution length of a Sokoban problem remains elusive. The best lower bound estimator is expensive to calculate, and is often ineffective.

None of the above obstacles are found in the "standard" single-agent test domains, such as $N \times N$-puzzles and Rubik's Cube.

3 Application-Independent Techniques

Ideally, applications should be specified with minimal effort and a "generic" solver would be used to compute the solutions. In small domains this is attainable (e.g., if it is easily enumerable). For more challenging domains, there have recently been a number of interesting attempts at

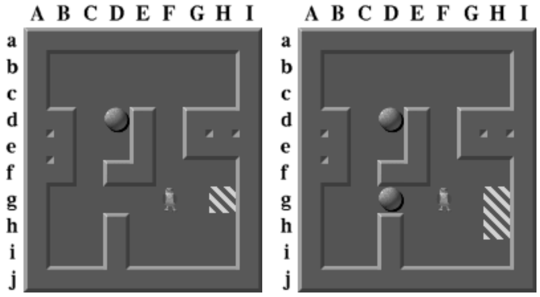

Figure 2: Two Simple Sokoban Problems

domain-independent solvers (e.g., *blackbox* [Kautz and Selman, 1996]). Before investing a lot of effort in developing a Sokoban-specific program, it is important to understand the capabilities of current AI tools. Hence, we include this information to illustrate the disparity between what application-independent problem solvers can achieve, compared to application-dependent techniques.

The Sokoban problems in Figure 2 [McDermott, 1998] were given to the program *blackbox* to solve. *Blackbox* was the winner of the AIPS'98 fastest planner competition. The first problem was solved within a few seconds and the second problem was solved in over an hour.

Clearly, domain-independent planners, like *blackbox*, have a long way to go if they are to solve the even simplest problem in the test suite (Figure 1). Hence, for this application domain we have no choice but to pursue an application-dependent implementation.

4 Application-Dependent Techniques

As reported in [Junghanns and Schaeffer, 1998c], we implemented IDA* for Sokoban. We gave the algorithm a fixed node limit of 1 billion nodes for all experiments (varying from 1 to 3 hours of CPU time on a single 195 MHz processor of an SGI Origin 2000). After adding an enhancement, *Rolling Stone* was run on 90 test problems (http://xsokoban.lcs.mit.edu/xsokoban.html) to find out how many could be solved and how much search effort was required to do so.

Figure 3 presents the experimental results for different versions of *Rolling Stone*. Version *R0* is the program using only IDA* with the lower bound; RA contains all the search enhancements. The logarithmic vertical axis shows the number of search nodes needed to solve a problem. The horizontal axis shows how many problems can be solved (out of 90), ordering the problems by search tree size. The performance lines in the figure are sorted from left to right with an increasing number of search enhancements.

Lower Bound (0 solved): To obtain an admissible estimate of the distance of a position to a goal, a minimum-cost, perfect bipartite matching algorithm is used. The matching assigns each stone to a goal and returns the total (minimum) distance of all stones to their goals. The algorithm is $O(N^3)$ in the number of stones N. IDA* with this lower bound cannot solve any of the test problems within one billion search nodes.

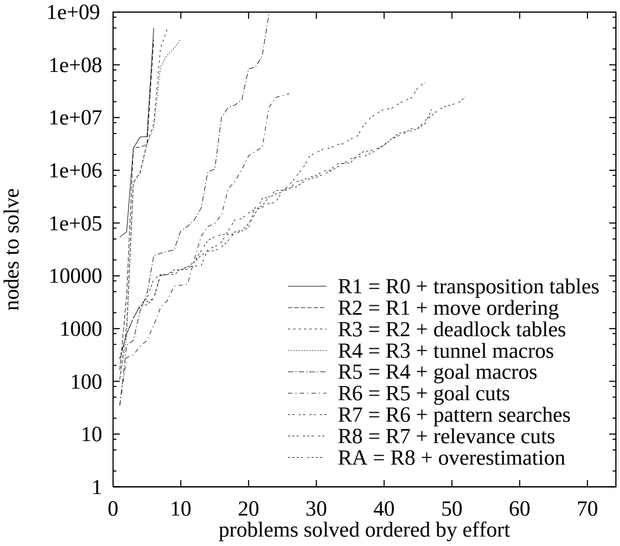

Figure 3: Program Performance

Transposition Table (6 solved): The search space of Sokoban is a graph, rather than a tree, so repeated positions and cycles are possible. A *transposition table* was implemented to avoid duplicate search effort. Positions that have the same stone locations and equivalent man locations (taking man reachability into account) are treated as the same position. Transposition tables reduces the search tree size by several orders of magnitude, allowing *Rolling Stone* to solve 6 problems.

Move Ordering (6 solved): Children of a node are ordered based on their likelihood of leading to a solution. *Move ordering* only helps in the last iteration. Even though move ordering results in no additional problems being solved, less search effort is used to solve each problem.

Deadlock Table (8 solved): The pattern database is a recent idea that has been successfully used in the $N \times N$-puzzles [Culberson and Schaeffer, 1996] and Rubik's Cube [Korf, 1997]. An off-line search enumerated all possible stone/wall placements in a 4×5 region and searched them to determine if deadlock was present. These results are stored in *deadlock tables*. During an IDA* search, the table is queried to see if the current move leads to a local deadlock. Thus, deadlock tables contain search results of partial problem configurations and are general with respect to all Sokoban problems.

Tunnel Macros (10 solved): A Sokoban maze often contains "tunnels" (such as the squares *Kh*, *Lh*, *Mh* and *Nh* in Figure 1). Once a stone is pushed into a tunnel, it must eventually be pushed all the way through. Rather than do this through search, this sequence of moves can be collapsed into a single macro move. By collapsing several moves into one, the height of the search tree is reduced. *Tunnel macros* are identified by pre-processing.

Goal Macros (23 solved): Prior to starting the search, a preliminary search is used to find an appropriate order in which to fill in the goal squares. In many cases this is a non-trivial computation, especially when the goal area(s) has several entrances. A specialized search is used to avoid fill sequences that lead to a deadlock. The knowledge about the goal area is then used to create goal macros, where stones are pushed directly from the goal area entrance(s) to the final goal square avoiding deadlocks. For example, in Figure 1, square *Gh* is defined as the entrance to the goal area; once a stone reaches it, a single macro move is used to push it to the next pre-determined goal square. These macro moves significantly reduce the search depth required to solve problems and can dramatically reduce the search tree size. Whenever a goal macro move is possible, it is the only move considered; all alternatives are *forward pruned.*

Goal Cuts (26 solved): *Goal cuts* effectively push the goal macros further up the search tree. Whenever a stone can be pushed to a goal entrance square, none of the alternative moves are considered. The idea behind these cuts is that if one is confident about using macro moves, one might as well prune alternatives to pushing that stone further up in the search tree.

Pattern Search (46 solved): *Pattern searches* [Junghanns and Schaeffer, 1998b] are an effective way to detect lower bound inefficiencies. Small, localized conflict-driven searches uncover patterns of stones that interact in such a way that the lower bound estimator is off by an arbitrary amount (even infinite, in the case of a deadlock). These patterns are used throughout the search to improve the lower bound. Patterns are specific to a particular problem instance and are discovered on the fly using specialized searches. Patterns represent the knowledge about dynamic stone interactions that lead to poor static lower bounds, and the associated penalties are the corrective measures.

Pattern searches lead to dramatic improvements of the search: many orders of magnitude vanish from the search tree size and 20 more problems can be solved. Note that tree sizes reported include the pattern search nodes.

Relevance Cuts (47 solved): *Relevance cuts* [Junghanns and Schaeffer, 1998a] are an attempt to cut down the branching factor using forward pruning. If moves are "inconsistent" to the previous move history, they are pruned. This heuristic is unsafe, since it has the potential to prune solution paths. However, it does decrease search tree sizes, and can be a beneficial trade-off.

Overestimation (52 solved): Given the difficulty of solving Sokoban problems, *any* solution, even a non-optimal one, is welcome. The patterns that *Rolling Stone* discovers indicate when potentially "difficult" situations arise. To ensure admissibility, some patterns that match are not always used to increase the lower bound. Overestimation allows every pattern to add to the lower bound. In principle, this can be interpreted as the program "avoiding" difficult situations. We prefer to describe it as a knowledge-driven postponement of search: the additional penalty only postpones when the search will explore a certain part of the tree, it will not cut branches indefinitely. In this respect, this method preserves completeness, but not solution optimality.

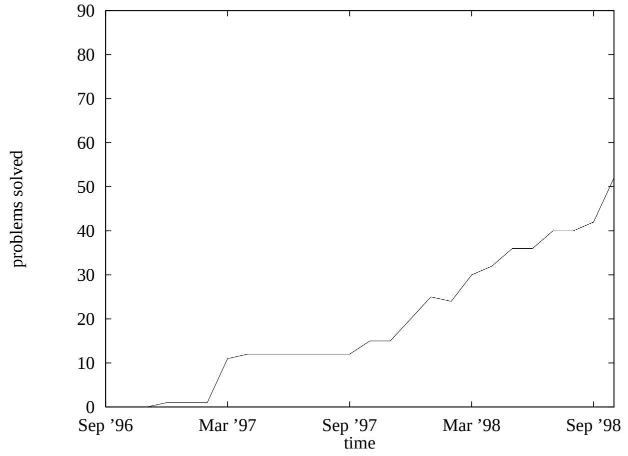

Figure 4: Number of Problems Solved Over Time

The performance gap between the first and last versions of *Rolling Stone* in Figure 3 is astounding. For example, consider extrapolating the performance of *Rolling Stone* with transposition tables so that it can solve the same number of problems as the complete program (52). 10^{50} (not a typo!) seems to be a reasonable lower bound on the difference in search tree sizes.

The preceding discussion closely corresponds to the order in which enhancements were initially added to *Rolling Stone* (although most enhancements have been continually refined). Figure 4 shows how these results were achieved over the 2-year development time. The development effort equates to a full-time PhD, a part time professor, a full-time summer student, and feedback from many people. Additionally, a large number of machine cycles were used for tuning and debugging. It is interesting to note the occasional *decrease* in the number of problems solved, the result of (favorable) bugs being fixed. The long, slow, steady increase is indicative of the reality of building a large system. Progress is incremental and often painfully slow.

The results in Figure 3 may misrepresent the importance of each feature. Figure 5 shows the results of taking the full version of *Rolling Stone* and disabling single search enhancements. In the absence of a particular method, other search enhancements can compensate to allow a solution to be found. Most notably, while the lower bound function alone cannot solve a single problem, neither can the complete system solve a single problem without the lower bound function.

Figure 5 shows that turning off goal macros reduces the number of problems solved by 35, more than 66%! Turning off transposition tables loses 23 problems. Turning off pattern searches reduces the number of solved problems by 16. Other than the lower bound function, these three methods are the most important for *Rolling Stone*; losing any one of them dramatically reduces the performance. While other enhancements don't have as dramatic an effect, turning any one of them off loses at least one problem.

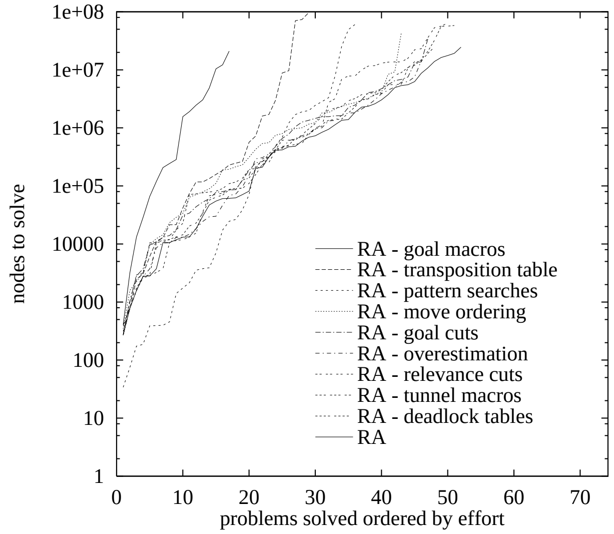

Figure 5: Effort Graphs For Methods Turned Off

5 Knowledge Taxonomy

In looking at the domain-specific knowledge used to solve Sokoban problems, we can identify several different ways of classifying the knowledge:

Generality. Classify based on how general the knowledge is: *domain* (e.g., Sokoban), *instance* (a particular Sokoban problem), and *subtree* (within a Sokoban search).

Computation. Differentiate how the knowledge was obtained: *static* (such as advice from a human expert) and *dynamic* (gleaned from a search).

Admissibility/Completeness. Knowledge can be: *admissible* (preserve optimality in a solution) or *non-admissible*. Non-admissible knowledge can either preserve *completeness* of the algorithm or render it *incomplete*. Admissible knowledge is necessarily complete.

Figure 6 summarizes the search enhancements used in *Rolling Stone*. Other enhancements from the literature could easily be added into spaces that are still blank, e.g. perimeter databases [Manzini, 1995] (dynamic, admissible, instance). Note that some of the enhancement classifications are fixed by the type of the enhancement. For example, any type of forward pruning is incomplete by definition, and move ordering always preserves admissibility. For some enhancements, the properties depend on the implementation. For example, overestimation techniques can be static or dynamic; goal macros can be admissible or non-admissible; pattern databases can be domain-based or instance-based.

It is interesting to note that, apart from the lower bound function itself, the three most important program enhancements in terms of program performance (Figure 5) are all dynamic (search-based) and instance/subtree specific. The static enhancements, while of value, turn out to be of less importance. Static knowledge is usually rigid and does not include the myriad of exceptions that search-based methods can uncover and react to.

Classification		Domain	Instance	Subtree
Static	admissible	lower bound	tunnel macros	move ordering
	complete			
	incomplete		relevance cuts	goal cuts
Dynamic	admissible	deadlock tables		pattern searches
				transposition table
	complete			overestimation
	incomplete		goal macros	

Figure 6: Taxonomy of Search Enhancements in Sokoban

6 Control Functions

There is another type of application-dependent knowledge that is critical to performance, but receives scant attention in the literature. *Control functions* are intrinsic parts of efficient search programs, controlling when to use or not use a search enhancement. In *Rolling Stone* numerous control functions are used to improve the search efficiency. Some examples include:

Transposition Table: Control knowledge is needed to decide when new information is worth replacing older information in the table. Also, when reading from the table, control information can decide whether the benefits of the lookup justify the cost.

Goal Macros: If a goal area has too few goal squares, then goal macros are disabled. With a small number of goals or too many entrances, the search will likely not need macro moves, and the potential savings are not worth the risk of eliminating possible solutions.

Pattern Searches: Pattern searches are executed only when a non-trivial heuristic function indicates the likelihood of a penalty being present. Executing a pattern search is expensive, so this overhead should be introduced only when it is likely to be cost effective. Control functions are also used to stop a pattern search when success appears unlikely.

Implementing a search enhancement is often only one part of the programming effort. Implementing and tuning its control function(s) can be significantly more time consuming and more critical to performance. We estimate that whereas the search enhancements take about 90% of the coding effort and the control functions only 10%, the reverse distribution applies to the amount of tuning effort needed and machine cycles consumed.

A clear separation between the search enhancements and their respective control functions can help the tuning effort. For example, while the goal macro creation only considers which order the stones should be placed into the goal area, the control function can determine if goal macros should be created at all. Both tuning efforts have very different objectives: one is search efficiency, the other risk minimization. Separating the two seems natural and convenient.

```
IDA*() {
  /** Compute the best possible lower bound **/
  lb = ComputeLowerBound();
  lb += UsePatterns();     /** Match Patterns **/
  lb += UseDeadlockTable();
  lb += UseOverestimate( CntrlOverestimate() );
  if( cutoff ) return;
  /** Preprocess **/
  lb += ReadTransTable();
  if( cutoff ) return;
  PatternSearch( CntrlPatternSearch() );
  lb += UsePatterns();
  if( cutoff ) return;
  /** Generate searchable moves **/
  movelist = GenerateMoves();
  RemoveDeadMoves( movelist );
  IdentifyMacros( movelist );
  OrderMoves( movelist );
  for( each move ) {
    if( Irrelevant( move, CntrlIrrelevent() )) next;
    solution = IDA*();
    if( solution) return;
    if( GoalCut() ) break;
    UpdateLowerBound();  /** Use New Patterns **/
    if( cutoff ) return;
  }
  /** Post-process **/
  SaveTransTable( CntrlTransTable() );
  return;
}
```

Figure 7: Enhanced IDA*

7 Single-Agent Search Framework

As presented in the literature, single-agent search consists of a few lines of code (usually IDA*). Most textbooks do not discuss search enhancements, other than cycle detection. In reality, non-trivial single-agent search problems require a more extensive programming (and possibly research) effort.

Figure 7 illustrates the basic IDA* routine, with our enhancements included (in *italics*). This routine is specific to *Rolling Stone*, but could be written in more general terms. It does not include a number of well-known single-agent search enhancements available in the literature. Control functions are indicated by parameters to search enhancement routines. In practice, some of these functions are implemented as simple *if* statements controlling access to the enhancement code.

Examining the code in Figure 7, one realizes that there are really only three types of search enhancements:

1. Modifying the lower bound (as indicated by the updates to *lb*). This can take two forms: optimally increasing the bound (e.g. using patterns) which reduces the distance to search, or non-optimally (using overestimation) which redistributes where the search effort is concentrated.
2. Removing branches unlikely to add additional infor-

```
for( each domain ) {
  /** Preprocess **/
  BuildDeadlockTable( CntrlDeadlockTable() );
  for( each instance ) {
    /** Preprocess **/
    FindTunnelMacros();
    FindGoalMacros( CntrlGoalMacros() );
    while( not solved ) {
      SetSearchParamaters();
      IDA*();
    }
    /** Postprocess **/
    SavePatterns( CntrlSavingPatterns() );
  }
}
```

Figure 8: Preprocessing Hierarchy

mation to the search (the *next* and *break* statements in the *for* loop). This forward pruning can result in large reductions in the search tree, at the expense of possibly affecting the completeness.

3. Collapsing the tree height by replacing a sequence of moves with one move (for example, macros).

Some of the search enhancements involve computations outside of the search. Figure 8 shows where the pre-search processing occurs at the domain and instance levels. Off-line computation of pattern databases or pre-processing of problem instances are powerful techniques that receive scant attention in the literature (chess endgame databases are a notable exception). Yet these techniques are an important step towards the automation of knowledge discovery and machine learning. Preprocessing is involved in many of the most valuable enhancements that are used in *Rolling Stone*.

Similar issues occur with other search algorithms. For example, although it takes only a few lines to specify the alpha-beta algorithm, the *Deep Blue* chess program's search procedure includes numerous enhancements (many similar in spirit to those used in *Rolling Stone*) that cumulatively reduce the search tree size by several orders of magnitude. If nothing else, the *Deep Blue* result demonstrated the degree of engineering required to build high-performance search-based systems.

8 Conclusions

This paper described our experiences working with a challenging single-agent search domain. In contrast to the simplicity of the basic IDA* formulation, building a high-performance single-agent searcher can be a complex task that combines both research and engineering. Application-dependent knowledge, specifically that obtained using search, can result in an orders-of-magnitude improvement in search efficiency. This can be achieved through a judicious combination of several search enhancements. Control functions are overlooked in the literature, yet are critical to performance. They represent a significant portion of the program development time and most of the program experimentation resources.

Domain-independent tools offer a quick programming solution when compared to the effort required to develop domain-dependent applications. However, with current AI tools, performance is commensurate with effort. Domain-dependent solutions can be vastly superior in performance. The trade-off between programming effort and performance is the critical design decision that needs to be made.

9 Acknowledgements

This research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada. Computational resources were provided by MACI. This paper benefited from interactions with Yngvi Björnsson, Afzal Upal and Rob Holte.

References

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

[Culberson, 1997] J. Culberson. Sokoban is PSPACE-complete. Technical Report TR97–02, Dept. of Computing Science, University of Alberta, 1997. ftp.cs.ualberta.ca/pub/TechReports/1997/TR97–02.

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

[Junghanns and Schaeffer, 1998a] A. Junghanns and J. Schaeffer. Relevance cuts: Localizing the search. In *The First International Conference on Computers and Games*, pages 1–13, 1998. To appear in: *Lecture Notes in Computing Science*, Springer Verlag.

[Junghanns and Schaeffer, 1998b] A. Junghanns and J. Schaeffer. Single-agent search in the presence of deadlock. In *AAAI*, pages 419–424, 1998.

[Junghanns and Schaeffer, 1998c] A. Junghanns and J. Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In R. Mercer and E. Neufeld, editors, *Advances in Artificial Intelligence*, pages 1–15. Springer Verlag, 1998.

[Kautz and Selman, 1996] H. Kautz and B. Selman. Pushing the envelope: planning, propositional logic and stochastic search. In *AAAI*, pages 1194–1201, 1996.

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

[Manzini, 1995] G. Manzini. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75:347–360, 1995.

[McDermott, 1998] Drew McDermott. Using regression-match graphs to control search in planning, 1998. Unpublished manuscript.