# Sokoban: Enhancing General Single-Agent Search Methods Using Domain Knowledge

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 general search enhancements on program performance. We show that the current state of the art in AI generally requires a large research and programming effort to use domain-dependent knowledge 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. This application-specific knowledge is discovered and applied using application-independent search enhancements. 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, transposition table, pattern search, pattern database, rapid random restart

## 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 enhancements using domain-specific knowledge. 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 puzzle and is of general interest as an instance of robot motion planning problems [4]. 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 and PSPACE-complete [2, 4].

Previously, we reported on our attempts to solve Sokoban problems using the standard single-agent search techniques available in the literature [10]. When these proved inadequate, solving only 10 problems of a 90-problem test suite, new algorithms had to be developed to improve search efficiency [8, 9, 11, 12]. This allowed 47 problems to be solved optimally or near-optimally. Additional efforts have since increased this number to 57. The results reported here document the large gains achieved by adding application-dependent knowledge to our program, *Rolling Stone*. Many of the search enhancements added to *Rolling Stone* result in the search-tree size being reduced by several orders of magnitude.

Analyzing all the additions made to the Sokoban solver reveals that the most valuable enhancements are based on search (both on-line and off-line). We classify the search enhancements along several dimensions including 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 much more extensive programming (and often research) effort. For example, achieving high performance for solving sliding tile puzzles requires enhancements such as cycle detection, pattern databases, move ordering and enhanced lower-bound calculations [3]. In this paper, we outline a new framework for developing high-performance single-agent search programs.

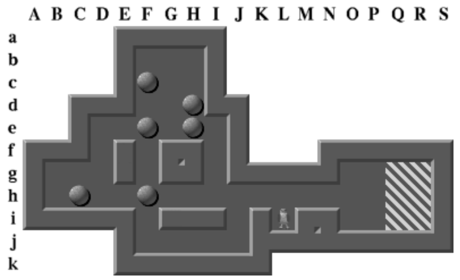
This paper contains the following contributions:

1. A case study showing the evolution of a Sokoban solver's performance, beginning with a domain-independent solver and ending with a highly-tuned, application-dependent program.

2. *Pattern searches* are a new proof procedure for improving a lower bound. They attempt to show that the lower bound for part of a state configuration can be increased.

3. *Relevance cuts* are a new way to add locality to a global search.

4. A taxonomy of single-agent search enhancements.




*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

5. A new framework for single-agent search, including search enhancements and their control functions.

In this paper, the term *domain-dependent* refers to knowledge about the (Sokoban) search space that is used by a search enhancement. The search enhancements discussed are otherwise generally applicable to application domains that have necessary search-space prerequisites (e.g. directed versus undirected graphs, or tree- versus graph-structure of the search space). Many of the techniques described in this paper have been successfully applied to other single-agent search domains (as well as for other classes of search problems). Some of the techniques that were initially conceived for Sokoban (such as pattern searches) have been used in other domains (the 15-puzzle and Bricks).

## 2 Sokoban

Figure 1 shows a sample Sokoban problem, the first and easiest of a 90 problem test suite available at http://xsokoban.lcs.mit.edu/xsokoban.html. The goal is simple: use the man to push (but *not* pull) all the stones in the maze to the shaded goal squares, abiding by the wall constraints. Only one stone can be pushed at a time. These rather simple rules belie the difficulty of Sokoban problems, especially with respect to computer solutions. The rules of Sokoban give rise to beautiful problems that can be extraordinarily complex.

To refer to squares in a Sokoban problem, we use a coordinate notation. The horizontal axis is labeled from "A" to "T", and the vertical axis from "a" to "t" (assuming the maximum sized 20x20 problem), starting in the upper left corner.

A move consists of pushing a stone from one square to another. For example, in Figure 1 the move *Fh-Eh* moves the stone on *Fh* left one square. We use *Fh-Eh-Dh* to indicate a sequence of pushes of the same stone. A move, of course, is only legal if there is a valid path by which the man can move behind the stone and push it. Thus, although we only indicate stone moves (such as *Fh-Eh*), implicit in this is the man's moves from its current position to the appropriate square to do the push (for *Fh-Eh* the man would have to move from *Li* to *Gh* via the squares *Lh*, *Kh*, *Jh*, *Ih* and *Hh*).

Throughout this paper, only a limited number of the strategic principles intrinsic to Sokoban will be mentioned. The full depth of Sokoban can only be appreciated by a more direct encounter with the game. Nevertheless, we want to mention briefly the challenge of deadlock positions resulting from the restriction of being able to push only one stone at a time. In the simplest case the man could push a stone into a corner, effectively immobilizing it on a non-goal square. Since all stones need to be pushed to a goal, any such fixed stone renders the problem unsolvable. We will call these and similarly unsolvable positions *deadlocked*.

In this paper we attempt to optimally solve Sokoban problems. One definition of optimal is to minimize the number of stone pushes in the solution. Another definition is to minimize the number of man movements. It is uncommon for a single solution to achieve both goals. In this work, optimality is defined as the minimal number of stone pushes.[1]

There are several properties that make Sokoban a challenging domain [10]:

- The combination of long solution lengths (from 97 to 674 stone pushes in the test set) and potentially large branching factors (up to 136) make Sokoban difficult for conventional search algorithms to solve. The size of the search space for $20 \times 20$ Sokoban mazes has been estimated at $10^{98}$ [7].

- Sokoban solutions are inherently sequential; only limited parts of a solution are interchangeable. Subgoals are often interrelated and thus cannot be solved independently. Attempts to decompose problems are also ineffective. For example, removing a single stone from a problem may make it trivial to solve, offering no insights as to how to solve the original problem.

- A simple and 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.

- The underlying structure of Sokoban can be represented by a directed graph, meaning that some moves are not reversible. Consequently, there are deadlock states from which no solution can be reached.

[1] Optimizing man movements may be harder in practice because of the difficulty in finding a good lower-bound function.

| Property | Specifics | 24-Puzzle | Rubik's Cube | Sokoban |
|---|---|---|---|---|
| Branching Factor | Average | 2.37 | 13.35 | 12 |
| | Range | 1-3 | 12-15 | 0-136 |
| Solution Length | Average | 100+ | 18 | 260 |
| | Range | 1-unknown | 1-20 | 97-674 |
| Search-Space Size | Upper Bound | $10^{25}$ | $10^{19}$ | $10^{98}$ |
| Calculation of Lower Bound | Full | $O(n)$ | $O(n)$ | $O(n^3)$ |
| | Incremental | $O(1)$ | $O(1)$ | $O(n^2)$ |
| Underlying Graph | | Undirected | Undirected | Directed |

Table 1: Search-Space Properties of Different Domains

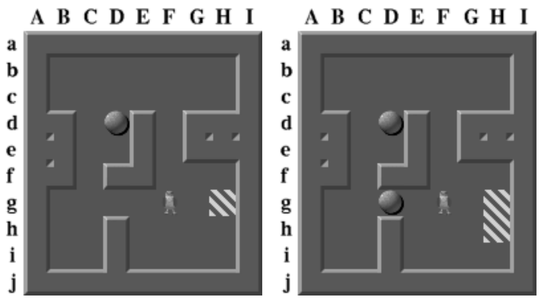



Figure 2: Two Trivial Sokoban Problems

Sokoban exhibits a large number of difficult search-space properties. Traditional domains for the scientific investigation of search methods, such as $N \times N$-puzzles and Rubik's Cube, are usually "easier" with respect to at least one search-space property. Table 1 compares several search-space properties of the above mentioned domains. It is unclear whether the conclusions obtained from these simpler domains will be effective for difficult search domains such as Sokoban, much less "real-world" problems.

# 3 Application-Independent Techniques

Ideally, we would like applications to be specified with minimal effort, and a "generic" solver could be used to compute the solutions. In small domains this is attainable (e.g., by exhaustive enumeration). For more challenging domains, there have been a number of interesting attempts at domain-independent solvers (e.g., *Blackbox* [13]). Before investing a lot of effort in developing a Sokoban-specific program, it is important to understand the capabilities of current AI tools. The comparison reveals a large disparity between what application-independent and application-dependent problem solvers can achieve.

The Sokoban problems in Figure 2 were given to *Blackbox* to solve. *Blackbox* was a winner in the AIPS'98 fastest planner competition. The first problem, containing a single stone, was solved by *Blackbox 3.3* in a few seconds. The second problem, containing two stones, requires 90 seconds to solve. Note that the search space (considering only the stones, not the man) is (43 *choose* 2) = 903 positions. In contrast, the non-trivial six-stone position shown in Figure 1 can be solved in less than a second by *Rolling Stone*. The search space is (52 *choose* 6) $= 293,162,688,000$.

Clearly, generalized planners, like *Blackbox*, have a long way to go if they are to solve even the simplest problem in the test suite (Figure 1). Domain-independent solvers are currently unable to automatically identify the knowledge needed to traverse large search spaces efficiently. Hence, for Sokoban we have no choice but to pursue using application-dependent knowledge in our implementation.

# 4 Application-Dependent Techniques

Iterative deepening A* (IDA*) was the basis for our Sokoban implementation [10]. We gave the algorithm a fixed node limit of 20 million nodes for all experiments (varying from 1 to 3 hours of CPU time on a single 195 MHz processor of an SGI Origin 2000). Over a period of 3 years, numerous enhancements were made to the basic IDA* algorithm. After each enhancement was added, the program's performance was assessed by running *Rolling Stone* on the 90-problem test suite to find out how many problems could be solved, and how much search effort was required to do so. Detailed results of the following experiments can be found in Tables 2 and 3 (pages 20 and 21). Starting with the basic IDA* and a simple lower-bound estimator, each version of the program (labeled from R0 to R10, ordered chronologically) adds one enhancement.

Although this section is called "application-dependent techniques", in reality all the techniques can be described in an application-independent way. However, their effectiveness depends on domain-specific knowledge.

The following sections describe each of the enhancements in *Rolling Stone*. For well-known ideas, only a brief description is given here. Full details are provided in the Appendices.

## 4.1 Simple Lower Bound (0 problems solved):

IDA* with a simple lower bound has no hope of finding a solution to any of the problems in our test suite. An obvious lower bound is the distance of each stone to its closest goal, a Manhattan distance for Sokoban. However, the gap between the lower-bound value and the actual solution length for any non-trivial problem is so large that the number of IDA* iterations, and thus their respective tree sizes, make solving these problems effectively impossible. Improving the lower bound is the key to better performance. Application-dependent knowledge is needed to produce the best possible bound.

## 4.2 Minimum Matching Lower Bound (R0, 0 solved):

To obtain a better admissible estimate for 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 minimum cost augmentation algorithm is $O(N^3)$, where $N$ is the number of stones [18]. During the search the lower bound only needs to be updated, which requires finding negative-cost cycles [14], and is less expensive to compute. Other optimizations are possible and reduce the computational cost. Nevertheless, maintaining the lower bound dominates the execution time of our program. More details can be found in Appendix A.1.

For the test suite, minimum matching improves the simple lower bound by an average of 30 pushes. Given that minimum matching preserves the solution parity,[2] this represents a decrease of 15 iterations for the IDA* search. The heuristic branching factor for Sokoban is more than 10, so this represents a decrease in the size of the search tree by a factor in excess of $10^{15}$! Nevertheless, IDA* with minimum matching alone cannot solve any of the test problems within the 20 million node search limit. The search limit was increased to one billion nodes, but still no problems could be solved. In the experiments, this version of the program is referred to as "R0".

## 4.3 Transposition Table (R1, 5 solved):

Even though search spaces are generally graphs, most search algorithms treat them as trees. If a state can have several predecessors, this can lead to duplicate work. The search could revisit nodes and even entire subtrees several times. These "transpositions" or cycles are detected using a *transposition table* in which useful information about previously visited nodes is stored [22]. Before expanding a node, the transposition table is consulted, and if valid information is found, it is used to potentially curtail the search. Further details can be found in Appendix A.2.

Adding transposition tables with $2^{18}$ entries to IDA* allows the search to solve 5 problems in our test suite within the 20 million node limit. Figure 3 shows the effort needed to solve those problems, ordered by search-tree size on a linear and a logarithmic scale. The vertical axis shows the number of nodes searched to solve the problems. The horizontal axis shows the number of problems solved. We will use this kind of graph throughout the paper and refer to them as *effort graphs*. The keys of the effort graphs refer to different versions of *Rolling Stone*. In Figure 3, "R1" is a version of *Rolling Stone* that adds transposition tables to version "R0".

[2] If the minimum-matching function returns an odd (even) number, then the correct solution length will also be odd (even). This can easily be verified by imposing a checker-board coloring of the squares and realizing that pushing a stone between differently colored squares requires an odd number of pushes, otherwise even. Furthermore, the difference in the number of black/white stones and goals determines the odd- or evenness of the solution length, regardless of stone-goal assignments and detours necessary because of stone interdependencies.

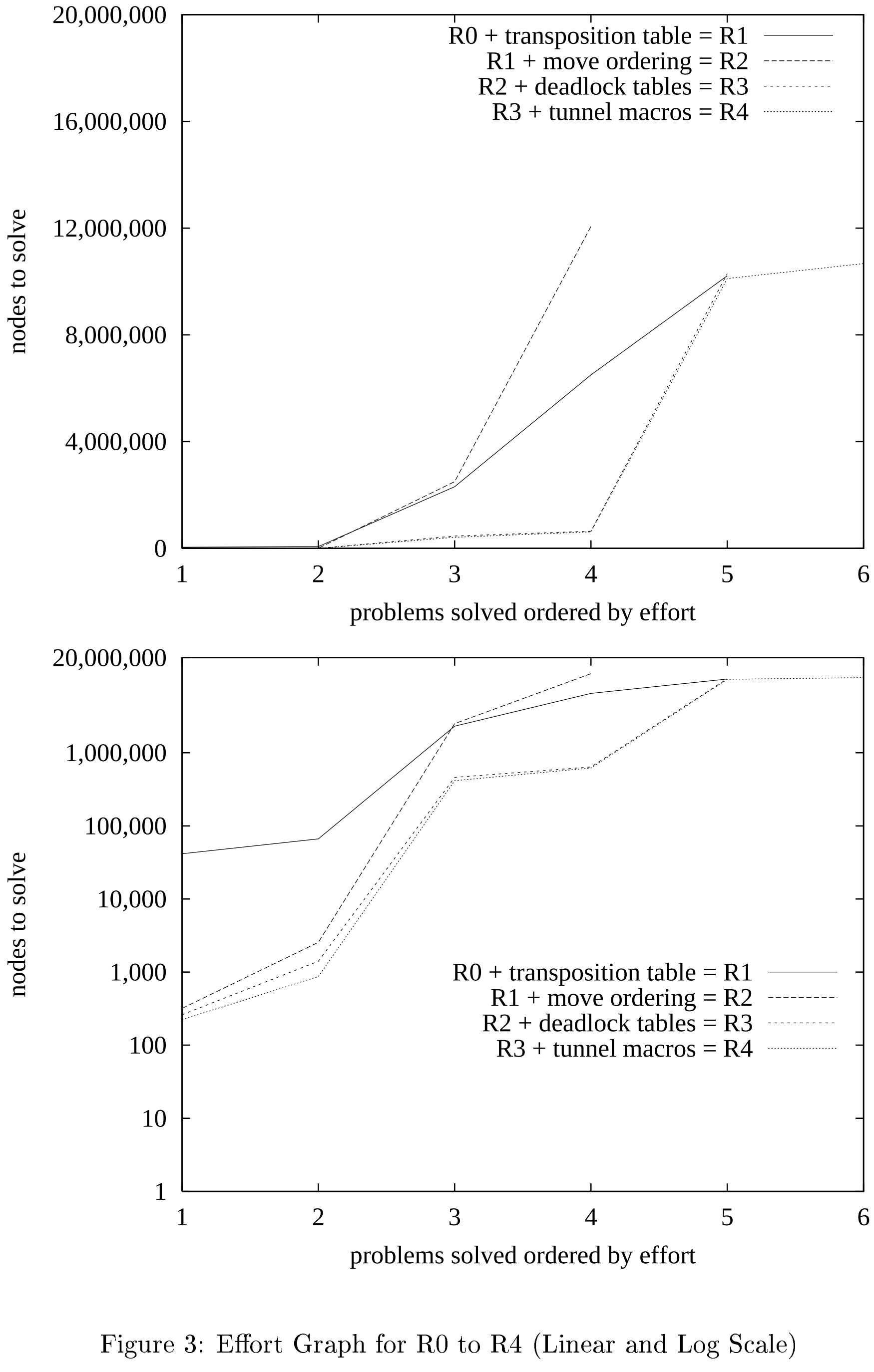



Figure 3: Effort Graph for R0 to R4 (Linear and Log Scale)

## 4.4 Move Ordering (R2, 4 solved):

Instead of visiting successors of a position in an arbitrary order, one can try to look at "good" successors first. Move (or successor) ordering is not used in best-first searches; the algorithm itself provides for a global ordering of the alternatives. In depth-first and breadth-first searches, move ordering can lead to efficiency gains because goals are found earlier (left in the tree) rather than later (right in the tree). For IDA*, ordering moves at interior nodes makes no difference to the search, except for the final iteration. Since the final iteration is aborted once a solution is found, finding a solution early in this iteration can significantly improve the performance [21].

The scheme used in *Rolling Stone*, *inertia*, does an excellent job of placing the best moves near the beginning of the move list (see Appendix A.3). Figure 3 shows the effect of adding move ordering to a program with the minimum matching lower bound and transposition tables (R2). Surprisingly, one problem can no longer be solved (in 20 million nodes) and two others require more nodes. This result is not favorable for move ordering. However, this appears to be bad luck for this small set of problems. After other features are added, move ordering shows up as a valuable contribution (as shown in Section 5 page 18).

## 4.5 Deadlock Table (R3, 5 solved):

The pattern database is a recent idea that has been successfully used in the sliding-tile puzzles [3] and Rubik's Cube [17]. An off-line search is used to enumerate all possible stone/wall placements in a $4 \times 5$ region to determine if a deadlock is present. These results are stored in *deadlock tables*. During the 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.

In the IDA* search, before making a move, the program queries the deadlock table to see if the move would result in a known deadlock. If so, the move is not considered further. On average, deadlock tables reduce the branching factor by 20% (see Appendix A.4). Given that the search is exponential in depth ($b^d$ where $b$ is the branching factor and $d$ is the average search depth) this represents an enormous reduction in the search space considered ($(.8 \times b)^d$).

Figure 3 shows the effect of adding deadlock tables (R3). Once again 5 problems can be solved, regaining the one lost with move ordering. For some problems, the search-tree size has been reduced by several orders of magnitude (see Tables 2 and 3, pages 20 and 21). It is illuminating to discover that such an impressive reduction in the branching factor does not allow more problems to be solved.

## 4.6 Tunnel Macros (R4, 6 solved):

The search algorithms discussed so far treat all moves equally. After making a move, all legal moves are considered as successors. These algorithms are therefore treating all moves as if they were unrelated. The method of macro moves

[15] is an attempt to group related atomic actions into higher level composite actions: macros. This can result in impressive search-space reductions. However, special attention must be paid to the side-effects that macros can have. They might influence the correctness and/or the completeness of the search, as well as the ability of the algorithm to find optimal solutions.

A *tunnel* is the part of a maze where the maneuverability of the man is restricted to a width of one. Since there can be at most one stone in a tunnel without creating an immediate deadlock, we can complete the remaining tunnel moves without loss of generality or optimality. If a tunnel is composed of articulation squares[3], we call the tunnel a *one-way tunnel*. Whenever the move generator creates a move into a one-way tunnel, the move is substituted with the macro pushing the stone all the way through the tunnel. This eliminates all the interleavings with other legal moves. More details are provided in Appendix A.5.

Tunnel macros result in one additional problem being solved, for a new total of 6 (Figure 3, page 8, version R4). However, the significant reduction in the size of the search tree contributes to the solvability of many future problems.

## 4.7 Goal Macros (R5, 17 solved):

Many of the Sokoban problems have all the goal squares grouped together in rooms. These *goal areas* are usually accessible through only a few squares which we call *entrances*. One can decompose the problem of solving a maze into:

- how to get each stone to one of the entrances, and
- how to pack stones into the goal areas.

Often these subgoals can be solved independently, thus reducing the search space enormously. Problem #1 is a good example. As soon as a stone reaches the entrance to the goal area at the right side of the maze (e.g. square *Mh*), the stone can be pushed directly to its final destination.

This is achieved by defining a goal area, marking its entrances, and precomputing the order in which goal squares are filled without introducing deadlock in the goal area. During the search, if a move is generated that pushes a stone onto the entrance square of a goal area, that move is replaced with a *goal macro* that generates a sequence of moves to push the stone directly to an appropriate goal square (in Figure 1, underlined sequences of moves are goal macros and are treated as a single move). Depending on the precomputation, there could be one or more goal-macro moves. All other moves are deleted from the move list; only the goal-macro moves are considered. If a stone can be pushed to its final destination, nothing else should matter at the moment, since completion of this task will reduce the complexity of the remaining problem. This differs from tunnel macros, where alternative moves are still searched. By removing other moves when a goal macro is present, the effect on the search-tree size is more dramatic than for tunnel macros. More details are provided in Appendix A.6.

[3] Squares dividing the maze into otherwise disconnected parts.

Figure 4 shows the dramatic effects of goal macros. Instead of solving 6 problems, *Rolling Stone* can now solve 17. The savings for individual problems are again several orders of magnitude. For example, the number of search nodes for problem #55 drops from over 20 million down to a mere 333 (see Table 2, page 20) – almost 5 orders of magnitude! On average, the searches are smaller by a factor of 20 with the goal macros. This is a conservative estimate, since unsuccessful searches are stopped at 20 million nodes. However, it is important to mention that goal cuts are unsafe and therefore incomplete.

## 4.8 Goal Cuts (R6, 24 solved):

The goal-macro heuristic eliminates all alternative moves from consideration when a goal macro is present. The reason for this is that if we can push a stone to its final destination, it will not affect other moves and they can be ignored. The same reasoning can be applied to the previous move: the move that pushed the stone to the square from which it will be "macro"-pushed to the goal square. *Goal cuts* do exactly that recursively further up the tree: if a stone is pushed to a goal with a goal macro at the end without interleaving other stone pushes, all alternatives to pushing that stone are deleted from the move list. More details are in Appendix A.7.

Figure 4 shows savings of approximately one to two orders of magnitude in search-tree size for the version using goal cuts (R6). Now 24 problems can be solved. Problem #65 was not solved without goal cuts; now it is solved with just over 600 nodes – the search tree is over 4 orders of magnitude smaller. For solved problems, the median search tree is a factor of 6 smaller.

## 4.9 Pattern Search (R7, 48 solved):

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. Ideally, having discovered a subset of a state that causes a deadlock (a *pattern* of stones), any state containing that pattern should be assigned the lower bound of $\infty$.

*Pattern searches* find patterns of stones that prove that the lower bound is in error. The errors could be small, improving the lower bound by as little as 2, or as much as $\infty$ in the case of a deadlock. All discovered patterns are saved and used throughout the search. If a pattern matches a subset of stones in a position, then the penalty associated with that pattern is added to the lower-bound estimate for the position. In effect, the program learns lower-bound penalty patterns and uses them to dynamically improve the lower-bound function.

In the following, we will refer to two different mazes: the *original maze*, the data structure used by the IDA* search, 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. 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

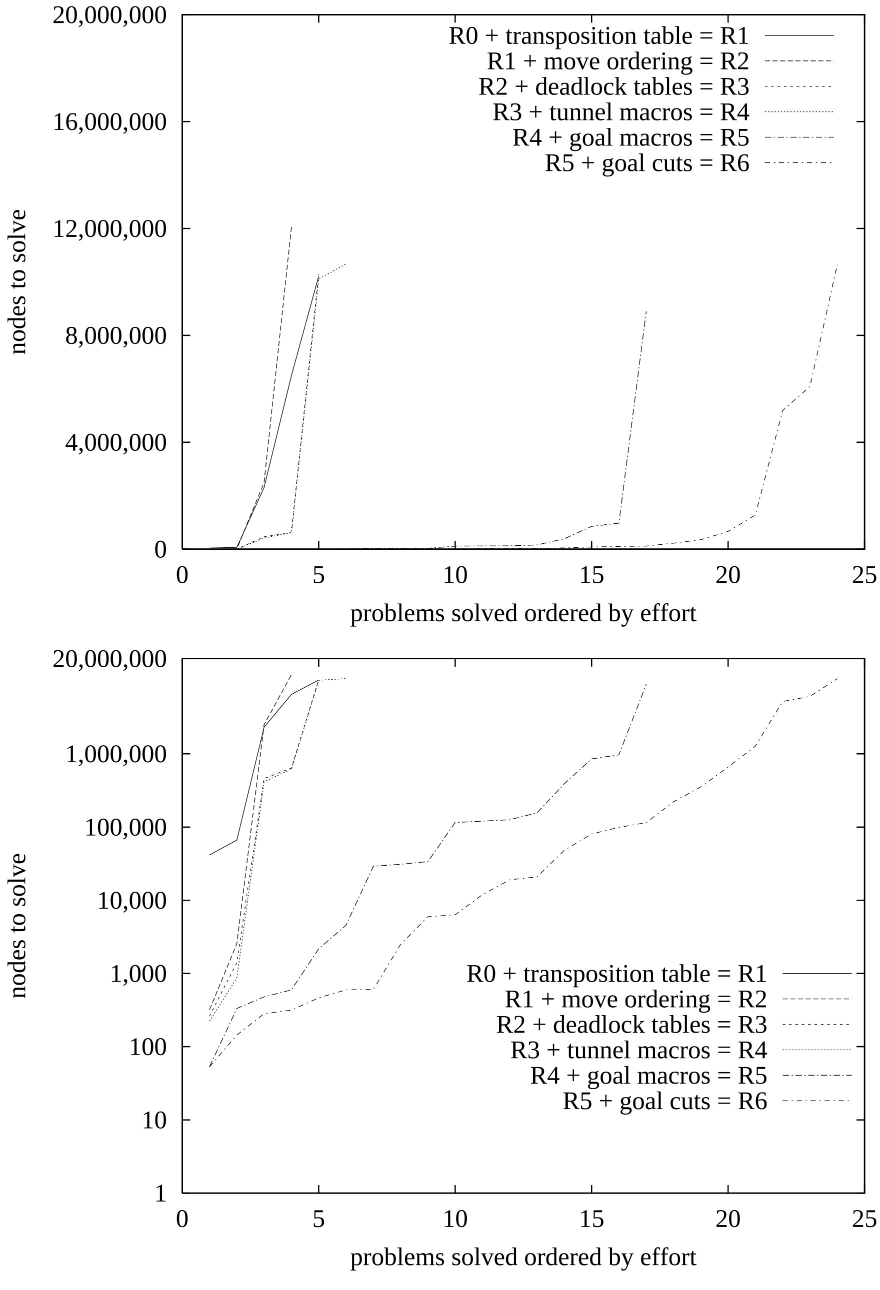



Figure 4: Effort Graph for R0 to R6 (Linear and Log Scale)

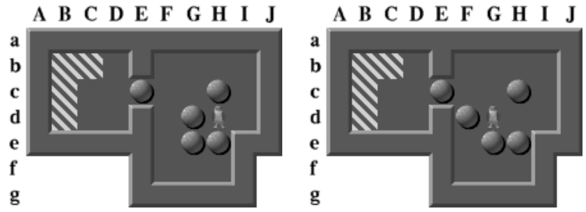



Figure 5: Deadlock Example

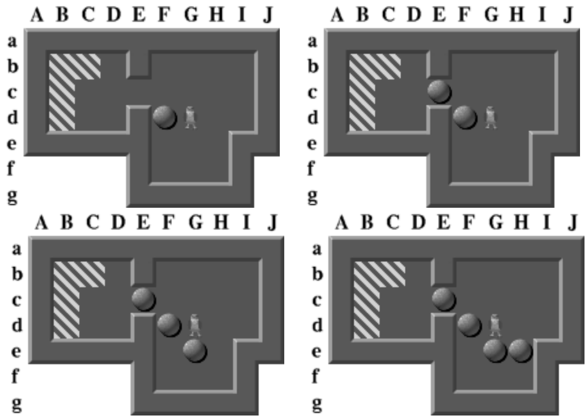



Figure 6: Sequence of Test Mazes as Passed to PIDA* (a, b, c and d)

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 included into the test maze for the pattern search. PIDA*, a version of IDA* tailored to be efficient at pattern searching, is called to solve this test maze (see Appendix A.8). It either returns in failure (no solution, hence deadlock), or it finds a solution. In the latter case, the number of pushes in the solution may disagree with that of our minimum matching lower bound. If so, then we know that the lower bound function is in error and can be improved.

Figure 5 shows a simple position, before and after the move *Gd-Fd*. The question is whether this move introduces a deadlock. Figure 6 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 6a). A PIDA* search finds a trivial solution. However, the search reveals that there is a conflict in the original maze that prevents this solution: the stone on *Ec*. This conflict is resolved by adding the stone to the test maze and trying to solve it (Figure 6b).

PIDA* will search the two-stone maze and again find a solution. This time there are no stone conflicts. However, the man had to move through square *Ge* to get behind the stone on *Fd*, again conflicting with the original maze. This stone is added to the test maze (Figure 6c) and another search is commenced. A solution will be found, requiring a fourth stone to be added (Figure 6d).

The next 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*) have been ignored.

The notion of bit (stone) patterns is similar to the Method of Analogies [1]. Pattern searches are a conflict-driven top-down proof of correctness, while the Method of Analogies is a bottom-up heuristic approximation.

The fewer stones in a penalty pattern, the more likely it will match an arbitrary position and be used to eliminate futile branches of the search. A *minimal penalty pattern* is a pattern from which no stone can be removed without decreasing its penalty. The attentive reader will have noticed that only three stones are needed to guarantee deadlock in Figure 6; the stone on *Ec* is not necessary. Before saving the pattern, our program will attempt to minimize the number of stones in it. The minimization routine takes an *N*-stone pattern and considers each of the possible *N-1*-stone sub-patterns. Each of the sub-patterns is searched to verify whether removing that stone preserves the deadlock or penalty. If the penalty still exists, then the stone was not part of the minimal pattern and is removed.

During an IDA* search, at each node the normal minimum matching lower bound is computed. If this value is insufficient to cause a cutoff, then the collection of penalty patterns is matched against the position. Of the patterns that match, the largest penalty is computed and added to the lower bound. If two or more patterns overlap, only a maximal non-overlapping subset of them is counted towards the position penalty. To prevent excessive pattern matching during the search (utility problem [20]), the number of patterns stored is restricted. The least recently used patterns are removed if necessary.

Figure 8 (page 19) shows the effort graph, now including the version of *Rolling Stone* using pattern searches (R7). The program can now solve 48 problems, 24 more than the previously best version!

In Table 3 (page 21), the search-tree size for R7 is broken down into two categories. The "total nodes" column reflects all positions visited in the search. The "IDA*" column gives the number of positions that the IDA* search visits. The difference is the number of pattern search nodes (PIDA*).

Except for the small searches (<20,000 nodes), the cost of performing the additional PIDA* searches is offset by the reduction in the IDA* search nodes. Problem #53 is an example. The savings for the IDA* tree are dramatic. Previously, the search was unable to solve this problem given 20,000,000 nodes. Now the search succeeds with only 159 IDA* nodes and a total of 22,310 nodes. Clearly, the pattern searches dominate the search cost, but the knowledge uncovered allows the program to solve the problem where it failed previously. In this example, *Rolling Stone* searches fewer IDA* nodes than the length of the solution! The search backtracks a mere 13 times for a solution of 186 pushes.

Pattern searches are a gamble: we invest search effort (PIDA* nodes) expecting to find useful knowledge. Problem #78 is one example of where the gamble does not pay off. Even though the tree size (IDA*) is reduced about 50 fold, including the PIDA* nodes triples the total number of nodes searched.

The results reported here are not the best numbers that can be achieved. The PIDA* nodes dominate the cost of the search for some problems. Some additional heuristics for deciding when to execute pattern searches could result in further improvements in the overall search efficiency. There are numerous parameters in the search, each of which can be tuned for maximal performance [7, 11].

Pattern searches have also been applied to sliding-tile puzzles [7]. The program dynamically learns penalty patterns, such as linear conflicts [6]. The cost of the pattern searches is small compared to the large reductions in the IDA* search tree.

Deadlock tables (or pattern databases) are another way to store pattern information. However, the patterns in such databases are necessarily smaller, because precomputing these patterns requires considerable computing resources and the resulting data needs to be stored, often exhaustively for fast hashing. Pattern searches avoid both these problems, because they are demand driven and only patterns that actually appear in the search are explored.

### 4.10 Relevance Cuts (R8, 50 solved):

Analysis of the trees built by an IDA* search quickly reveals that the search algorithm considers move sequences that no human would ever consider. Even completely unrelated moves are tested in every legal combination—all in an effort to prove that there is no solution for the current threshold. How can a program mimic an "understanding" of relevance? We suggest that a reasonable approximation of relevance is *influence*. If two moves do not influence each other, then it is unlikely that they are relevant to each other. If a program had a good "sense" of influence, it could assume that in a given position all previous moves belong to a (unknown) plan of which a continuation can only be a move that is relevant—in our approximation, is influencing whatever was played previously. *Relevance cuts* eliminate moves from that search that appear to be irrelevant to the preceding sequence of moves.

A move is considered relevant only if the previous $m$ moves influence it. The search is only allowed to make relevant moves with respect to previous moves, and only a few exceptions are permitted. With these restrictions in place, the search is forced to spend its effort locally, since random jumps within the search area are discouraged. Forcing the program to consider local moves is making it adopt a pseudo-plan; an exception corresponds to a decision to change plans. Of course, restricting the number considered for a node will result in the possibility of optimal solutions being found.

An influence metric can be achieved in different, domain-specific ways. Appendix A.9 gives an overview of our implementation. Even though the specifics aren't necessarily applicable to other domains, the basic philosophy of the approach is. We approximate the influence of two moves on each other by the influence between their *from* squares. Influence is determined using the notion

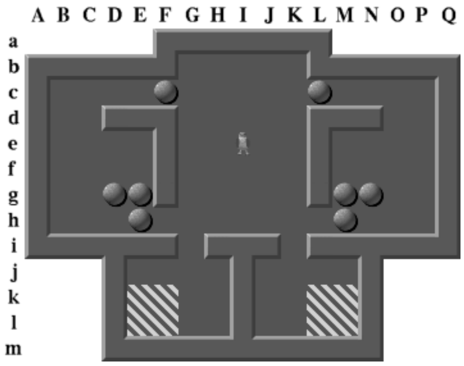



Figure 7: Example Maze With Locality

of a "most influential path" between the squares. Small off-line searches are used to statically precompute an $InfluenceTable$ containing the influence values between any pair of $from$ squares. For each pair of squares, a breadth-first search is used to find the path(s) with the largest influence. The algorithm is similar to a shortest-path finding algorithm, except that we use influence here and not geographic distance.

Figure 7 shows an example where humans immediately identify that solving this problem involves considering two separate sub-problems. The solution to the left and right sides of the problem are completely independent of each other. An optimal solution needs 82 moves; *Rolling Stone*'s lower-bound estimator returns a value of 70. Standard IDA* will need 7 iterations to find a solution (our lower-bound estimator preserves the odd/even parity of the solution length, meaning that it iterates by 2 at a time). IDA* will try every possible (legal) move combination, intermixing moves from both sides of the problem. Clearly, this is unnecessary and inefficient. Solving one of the sub-problems requires only 4 iterations, since the lower bound is off by only 6. Considering this position as two separate problems will result in an enormous reduction in the search complexity.

Our implementation of influence considers all moves on the left side as distant from those on the right, and *vice versa*. This way only a limited number of switches is considered during the search. Our parameter settings allow for only one non-local move per 9-move sequence. For this contrived problem, relevance cuts decrease the number of nodes searched from 32,803 to 24,748 while still returning an optimal solution (the pattern searches were turned off for simplicity). The savings (25%) appear relatively small because the transposition table catches repeated positions (many of which may be the result of irrelevant moves) and eliminates them from the search. Although the relevance cuts provide a welcome reduction in the search effort required, it is only a small step towards achieving all the possible savings. For example, each of the sub-problems can be solved by itself in only 329 nodes! The difference between $329 \times 2$ and

32,803 illustrates why IDA* in its current form is inadequate for solving large, non-trivial real-world problems; the algorithm is incapable of taking advantage of exploitable structural properties of the domain. Clearly, more sophisticated methods are needed. Further refinement of the relevance cut parameters can likely make a big difference in performance.

The overhead of the relevance cuts is negligible; the influence of two moves can be established by a simple table lookup. This is in stark contrast to our pattern searches, where the overhead dominates the cost of the search for most problems. The addition of relevance cuts increases the number of solved problems to 50. Figure 8 shows that the benefits of relevance cuts are only discernible on the largest searches. This is not a negative comment on the effectiveness of relevance cuts; it only reflects the observation that most of the solved problems already have very efficient searches.

### 4.11 Overestimation (R9, 54 solved):

To ensure optimality of solutions produced by A*-based algorithms, the heuristic has to be admissible. This limits the choice of knowledge that can be used. Even if some knowledge correlates well with the distance to the goal, but there is a chance that it overestimates, it cannot be used because the solution optimality would not be guaranteed. This shows that optimality has its price. Instead of fitting the heuristic distance to a solution $h$ as closely as possible to the actual distance $h^*$, we are restricted to creating a lower bound. The error of such a lower-bound function is often larger than a function that is allowed to occasionally overestimate. The larger the error of the lower-bound function, the less efficient the search.

We have seen in previous sections that an aggressive treatment of the search space is needed to make significant progress. The examples of the goal macros and relevance cuts have shown the benefits that are achievable when the small risk of losing optimality and completeness is taken. Therefore, it seems logical to question the admissibility constraint for the heuristic function. The hope is to achieve a closer fit of $h$ to $h^*$, albeit at the cost of non-optimal solutions.

Our *overestimation* technique combines the penalties for all pattern-search patterns that match in a position. Further details are in Appendix A.10.

Figure 8 shows that 3 additional problems can now be solved. There is roughly one order of magnitude savings in search-tree size (see Table 3). With overestimation, almost all solved problems, except #49, have smaller or insignificantly larger number of nodes. Problem #26, for example, drops from over 20 million nodes to just under 123,000. While some searches with overestimation use more iterations to find a goal, the search for problem #26 uses less; the initial position is overestimated enough to allow the search to find a solution in fewer iterations. On average, the IDA* and total nodes are reduced by roughly half.

### 4.12 Rapid Random Restart (R10, 57 solved):

Some problem classes exhibit the property of *heavy tails*. Heavy tails refer to the high likelihood of problem instances being very hard to solve with a certain algorithm, its heuristics and (random) parameters used. Rapid Random Restart (RRR) assumes that by varying parameters to the solution algorithm (here search), it is possible to reduce the solution time dramatically [5]. Therefore, instead of using all the available time with one parameter setting, RRR repeatedly aborts the search after a given effort limit and restarts it with different (random) parameters.

In *Rolling Stone*, RRR is used to interrupt an iteration and restart it with a different move ordering tie-breaking scheme (see Appendix A.11). Now 57 of the 90 problems can be solved, as shown in Figure 8.

## 5 Single-Agent Search Enhancements

The performance gap between the first and last versions of *Rolling Stone* in Figure 8 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 (57). $10^{50}$ (not a typo!) seems to be a reasonable lower bound on the difference in search-tree sizes.

For each of the unsolved problems, an additional search to 200 million nodes was performed. This resulted in two more problems being solved (numbers 25 and 28), bringing the total number of solved problems to 90. It is discouraging to see an order of magnitude more computing power translating into such a small improvement, clearly an indication of the difficulty of solving Sokoban problems. For some problems (notably number 50), the IDA* search threshold is so far from the best known human solution, that there is no hope of ever solving this problem with our current techniques.

The ordering of the preceding sub-sections closely corresponds to the order in which enhancements were initially added to *Rolling Stone* (although most enhancements have been continually refined). Figure 9 shows how these results were achieved over the 3-year development time. The development effort equates to a full-time PhD student, a part-time professor, one summer student, and valuable 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 8 may misrepresent the importance of each feature. Consider removing a single enhancement from *Rolling Stone*. 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. Turning off goal macros reduces the number

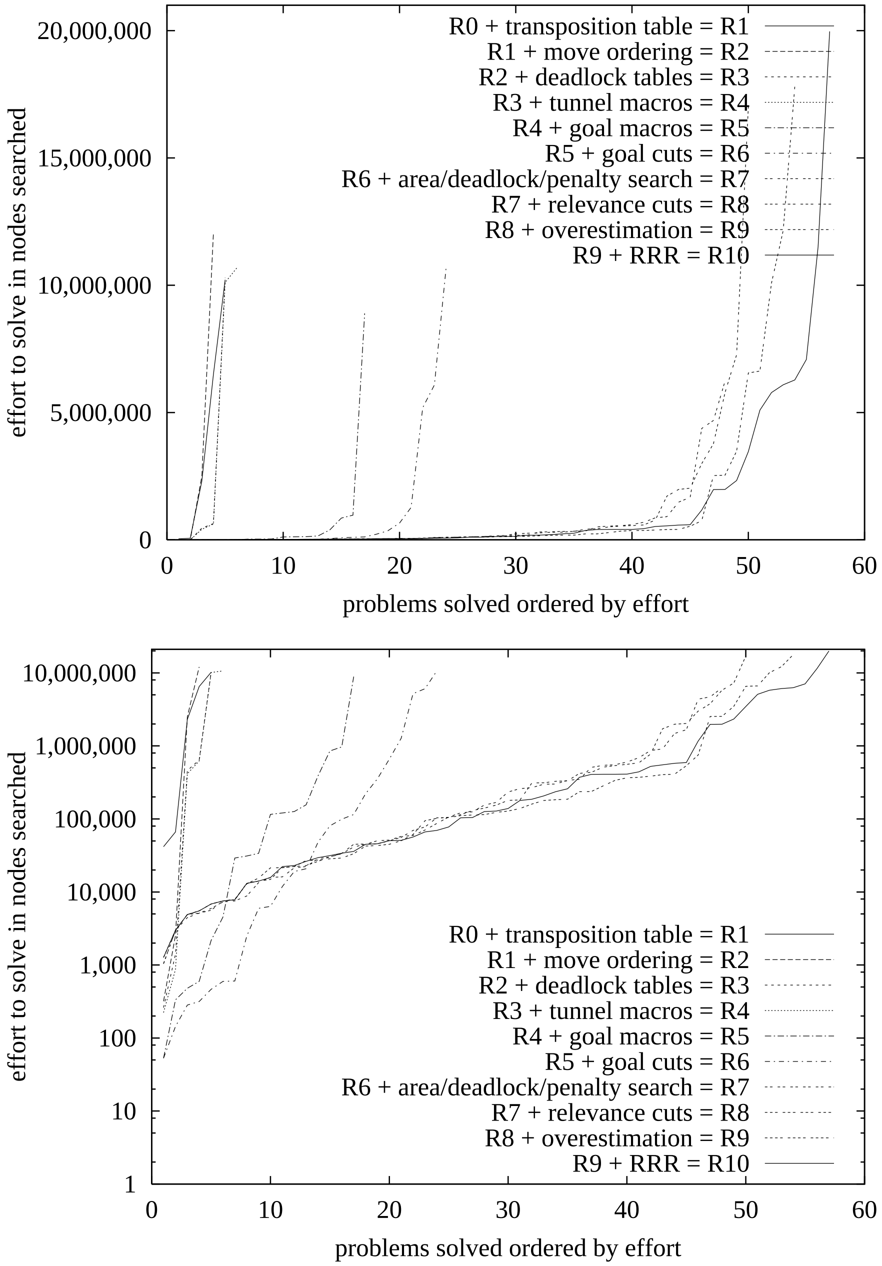



Figure 8: Effort Graph for R0 to R10 (Linear and Log Scale)

| # | R1 = R0 + Transposition Table | R2 = R1 + Move ordering | R3 = R2 + Deadlock Tables | R4 = R3 + Tunnel Macros | R5 = R4 + Goal Macros | R6 = R5 + Goal Cuts |
|---|---|---|---|---|---|---|
| | IDA* nodes | IDA* nodes | IDA* nodes | IDA* nodes | IDA* nodes | IDA* nodes |
| 1 | 41,640 | 319 | 261 | 223 | 53 | 53 |
| 2 | > 20,000,000 | > 20,000,000 | 640,680 | 620,030 | 2,176 | 316 |
| 3 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 29,148 | 2,493 |
| 4 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 597 | 597 |
| 5 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 1,275,146 |
| 6 | 10,214,381 | 12,061,182 | 10,294,734 | 10,107,621 | 4,546 | 283 |
| 7 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 126,023 | 48,209 |
| 8 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 9 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 659,972 |
| 10 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 11 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 12 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 17 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 10,672,805 | 120,747 | 11,910 |
| 19 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 21 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 10,643,971 |
| 23 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 25 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 26 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 30 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 33 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 34 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 36 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 38 | 2,311,000 | 2,500,678 | 460,089 | 415,485 | 33,812 | 19,083 |
| 40 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 43 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 6,084,369 |
| 45 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 49 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 8,895,883 | 5,189,494 |
| 51 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 390,690 | 80,504 |
| 53 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 54 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 55 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 333 | 144 |
| 56 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 57 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 58 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 59 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 60 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 61 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 62 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 6,337 |
| 63 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 64 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 65 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 604 |
| 67 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 68 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 70 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 71 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 72 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 73 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 75 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 76 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 77 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 |
| 78 | 66,309 | 2,555 | 1,408 | 871 | 480 | 465 |
| 79 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 156,203 | 5,964 |
| 80 | 6,500,890 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 115,574 | 114,930 |
| 81 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 853,607 | 221,690 |
| 82 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 971,093 | 99,236 |
| 83 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 31,096 | 20,847 |
| 84 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | > 20,000,000 | 354,295 |
| | >1,059,134,220 | >1,074,564,734 | >1,051,397,172 | >1,041,817,035 | >811,732,061 | >684,840,912 |

Table 2: Adding Enhancements (I)

| # | R7 = R6 + Pattern Search | | R8 = R7 + Relevance Cuts | | R9 = R8 + Overestimation | | R10 = R9 + Rapid Restart | |
|---|---|---|---|---|---|---|---|---|
| | IDA* nodes | total nodes | IDA* nodes | total nodes | IDA* nodes | total nodes | IDA* nodes | total nodes |
| 1 | 50 | 1,042 | 50 | 1,042 | 55 | 1,267 | 55 | 1,267 |
| 2 | 82 | 7,532 | 80 | 7,530 | 80 | 7,530 | 80 | 7,530 |
| 3 | 94 | 13,445 | 87 | 12,902 | 94 | 14,095 | 94 | 14,095 |
| 4 | 187 | 50,369 | 187 | 50,369 | 187 | 50,369 | 187 | 50,369 |
| 5 | 436 | 59,249 | 202 | 43,298 | 153 | 33,755 | 239 | 35,974 |
| 6 | 85 | 5,119 | 84 | 5,118 | 84 | 5,503 | 84 | 5,503 |
| 7 | 1,704 | 28,561 | 1,392 | 28,460 | 338 | 14,832 | 237 | 15,790 |
| 8 | 317 | 339,255 | 291 | 311,609 | 315 | 409,714 | 315 | 409,714 |
| 9 | 704 | 168,412 | 1,884 | 435,388 | 1,591 | 385,084 | 1,734 | 407,103 |
| 10 | 1,909 | 1,480,115 | 1,810 | 1,713,429 | 2,920 | 2,539,524 | 25,034 | 19,967,875 |
| 11 | 14,048 | 4,691,929 | 5,679 | 2,994,297 | 4,058 | 2,527,286 | 3,902 | 2,331,950 |
| 12 | 162,129 | 4,373,802 | 4,912 | 559,184 | 951 | 372,264 | 951 | 372,264 |
| 17 | 2,473 | 30,111 | 2,038 | 29,116 | 2,158 | 30,242 | 2,336 | 33,901 |
| 19 | 59,433 | > 20,000,000 | 16,606 | 7,269,595 | 14,178 | 6,631,475 | 12,801 | 6,089,182 |
| 21 | 1,853 | 154,593 | 1,177 | 179,734 | 573 | 113,042 | 1,774 | 258,852 |
| 23 | 87,744 | > 20,000,000 | 59,498 | > 20,000,000 | 23,337 | 6,555,398 | 23,679 | 7,082,584 |
| 25 | 1,239 | 553,900 | 21,536 | 5,784,086 | 683 | 366,035 | 1,231 | 592,585 |
| 26 | 2,606,167 | > 20,000,000 | 2,125,116 | > 20,000,000 | 380 | 122,997 | 496 | 126,379 |
| 30 | 14,297 | > 20,000,000 | 14,124 | > 20,000,000 | 27,731 | 17,795,114 | 3,595 | 3,467,260 |
| 33 | 5,035 | 866,085 | 2,765 | 586,684 | 604 | 283,926 | 1,865 | 551,406 |
| 34 | 542 | 298,674 | 11,431 | 1,981,993 | 9,746 | 749,787 | 731 | 442,025 |
| 36 | 78,325 | > 20,000,000 | 23,467 | > 20,000,000 | 18,338 | 12,150,606 | 10,196 | 5,785,290 |
| 38 | 2,539 | 51,276 | 7,011 | 154,969 | 10,473 | 160,176 | 2,340 | 56,563 |
| 40 | 41,131 | > 20,000,000 | 23,274 | 17,004,253 | 16,725 | 10,086,547 | 19,125 | 11,505,836 |
| 43 | 5,308 | 690,426 | 1,729 | 421,483 | 2,225 | 535,148 | 2,332 | 523,907 |
| 45 | 1,685 | 508,124 | 339 | 181,566 | 602 | 404,217 | 588 | 410,134 |
| 49 | 375,293 | 1,670,236 | 53,113 | 327,643 | 441,638 | 3,486,905 | 136,700 | 1,168,194 |
| 51 | 137 | 8,825 | 256 | 21,491 | 256 | 21,491 | 306 | 29,569 |
| 53 | 159 | 22,310 | 157 | 22,308 | 157 | 22,308 | 157 | 22,308 |
| 54 | 106,663 | 910,532 | 163,757 | 2,031,577 | 269 | 45,332 | 872 | 66,306 |
| 55 | 97 | 2,993 | 97 | 2,993 | 97 | 2,993 | 97 | 2,993 |
| 56 | 353 | 57,785 | 377 | 61,189 | 911 | 55,865 | 605 | 50,924 |
| 57 | 256 | 121,384 | 234 | 114,416 | 209 | 128,282 | 209 | 128,282 |
| 58 | 426 | 268,713 | 211 | 130,474 | 231 | 138,838 | 231 | 138,838 |
| 59 | 795 | 348,214 | 1,420 | 775,753 | 602 | 337,905 | 1,437 | 409,470 |
| 60 | 223 | 41,310 | 160 | 27,386 | 18,100 | 114,642 | 304 | 31,413 |
| 61 | 314 | 106,206 | 309 | 105,411 | 299 | 77,555 | 299 | 77,555 |
| 62 | 211 | 70,478 | 195 | 101,934 | 180 | 69,728 | 180 | 69,728 |
| 63 | 567 | 259,537 | 703 | 312,546 | 473 | 237,196 | 1,371 | 578,066 |
| 64 | 378 | 300,684 | 405 | 332,402 | 193 | 186,508 | 193 | 186,508 |
| 65 | 196 | 21,442 | 196 | 21,442 | 165 | 23,004 | 165 | 23,004 |
| 67 | 18,107 | 601,178 | 12,669 | 512,488 | 298 | 104,356 | 298 | 104,356 |
| 68 | 2,278 | 541,080 | 1,953 | 538,509 | 324 | 236,157 | 324 | 236,157 |
| 70 | 412 | 125,454 | 431 | 140,765 | 446 | 178,657 | 446 | 178,657 |
| 71 | 1,432,332 | > 20,000,000 | 8,234,574 | > 20,000,000 | 1,132,180 | > 20,000,000 | 183,170 | 1,973,352 |
| 72 | 134 | 44,908 | 134 | 44,908 | 123 | 45,735 | 123 | 45,735 |
| 73 | 201 | 87,019 | 214 | 94,568 | 225 | 103,494 | 225 | 103,494 |
| 75 | 61,973 | > 20,000,000 | 55,274 | > 20,000,000 | 259,971 | > 20,000,000 | 12,786 | 5,095,054 |
| 76 | 185,633 | 6,236,656 | 74,315 | 3,775,394 | 251 | 183,656 | 4,123 | 1,980,094 |
| 77 | 1,092,369 | > 20,000,000 | 1,019,702 | > 20,000,000 | 1,108,195 | > 20,000,000 | 251,768 | 6,277,715 |
| 78 | 64 | 4,451 | 64 | 4,913 | 64 | 4,913 | 64 | 4,913 |
| 79 | 125 | 15,833 | 122 | 15,527 | 127 | 13,114 | 127 | 13,114 |
| 80 | 100 | 16,114 | 165 | 26,943 | 176 | 26,309 | 176 | 26,309 |
| 81 | 21,501 | 234,235 | 2,662 | 42,445 | 875 | 111,033 | 2,651 | 206,423 |
| 82 | 86 | 33,445 | 86 | 33,445 | 117 | 45,014 | 117 | 45,014 |
| 83 | 91 | 7,294 | 80 | 5,631 | 108 | 6,856 | 108 | 6,856 |
| 84 | 94 | 5,960 | 106 | 7,938 | 108 | 7,818 | 108 | 7,818 |
| | 6,391,084 | >206,536,295 | 11,950,910 | >189,388,544 | 3,105,947 | >128,361,597 | 715,741 | 79,833,557 |

Table 3: Adding Enhancements (II)

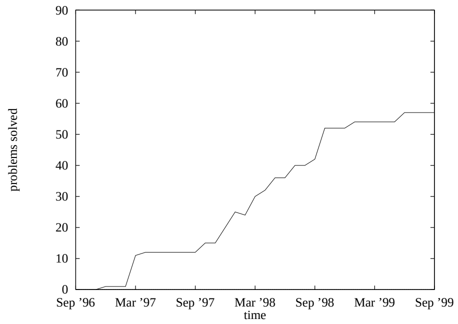



Figure 9: Number of Problems Solved Over Time

of problems solved by 33, almost 60%! Turning off pattern searches reduces the number of solved problems by 22, while disabling transposition tables loses 19 problems. 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.

## 5.1 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).

**Knowledge Source.** 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 10 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 [19] (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

| 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 10: Taxonomy of Search Enhancements in Sokoban

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 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.

## 5.2 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.

## 5.3 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 11 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 11, 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 information 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 12 shows where the pre-search processing occurs at the domain and

```
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 moves to consider **/
    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 11: Enhanced IDA*

```
for( each domain ) {
    /** Preprocess **/
    BuildDeadlockTable( CntrlDeadlockTable() );

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

Figure 12: Preprocessing Hierarchy

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.

Figure 13 shows a different perspective on the problem of knowledge levels. The diagram shows a hierarchy of problems, solvers, and the corresponding knowledge used. The application domain is located at the core; the basic solver is strictly concerned with this core application and exclusively uses core-application-specific knowledge. The next higher level in the hierarchy treats this entire process as the application. It supplies the core-application knowledge to the core-solver. The knowledge required at this level is control knowledge—controlling the core-knowledge gathering. When viewed in this context, it is clear why pattern searches and goal macros are such important enhancements. This diagram also shows an important direction of future research: automation of the next higher levels of the knowledge hierarchy.

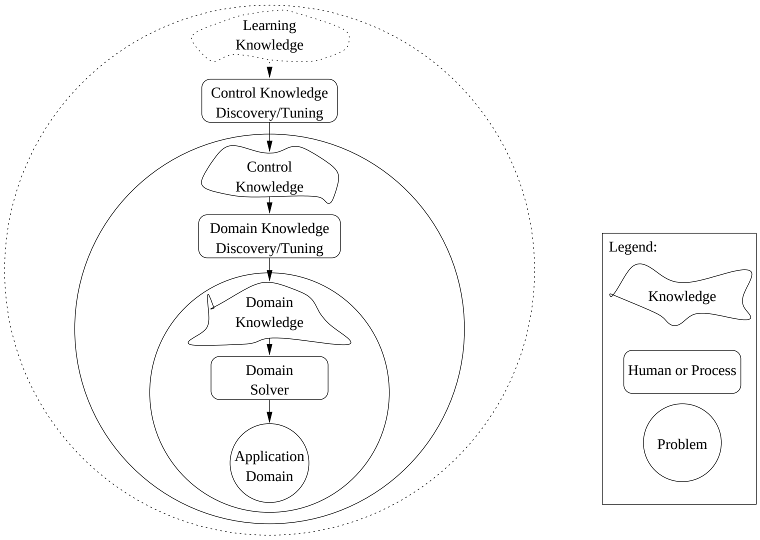



Figure 13: Framework

# 6 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.

# 7 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 research benefited from interactions with Don Beal, Darse Billings, Yngvi Björnsson, John Buchanan, Neil Burch, Joe Culberson, Matt Ginsberg, Carla Gomes, Rob Holte, Henry Kautz and Richard Korf.

## References


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

[2] J. Culberson. Sokoban is PSPACE-complete. In *Informatics 4, Fun With Algorithms*, pages 65–76. Carleton Scientific, 1999.

[3] J. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

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

[5] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI*, pages 431–437, 1998.

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

[7] A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search.* PhD thesis, University of Alberta, 1999.

[8] A. Junghanns and J. Schaeffer. Relevance cuts: Localizing the search. In *The First International Conference on Computers and Games*, pages 1–13, 1998. Also in: *Computers and Games*, J. van den Herik and H. Iida (editors), pages 1-14, 1999, Springer Verlag.

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

[10] 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.

[11] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In *IJCAI*, pages 570–575, 1999.

[12] A. Junghanns and J. Schaeffer. Sokoban: Improving the search with relevance cuts. *Journal of Theoretical Computing Science*, 1999. To appear.

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

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

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

[16] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

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

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

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

[20] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *AAAI*, pages 564–569, 1988.

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

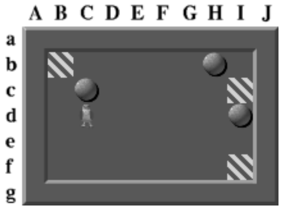
[22] D. Slate and L. Atkin. Chess 4.5 — The Northwestern University chess program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

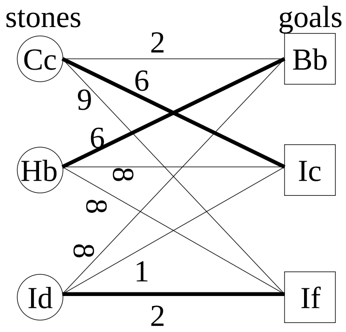
# A Appendices

## A.1 Minimum Matching

Figure 14 shows an example of the lower-bound calculation. The table lists the distances from the three stones to each of the three goals in the maze. The bold entries represent a minimum cost matching. It is important to note here that the minimum matching algorithm solves one important problem. Even though the stone on *Cc* and the stone on *Id* both have goals close by, they have to be pushed to a goal further away. While counting how many stones are off a goal square would return a lower bound of 3, and summing the distances of all stones to their closest goal squares would return 5, the minimum matching lower bound returns 14. This higher heuristic bound allows the search algorithm to eliminate a large fraction of the total search space that is irrelevant to an optimal solution.

The distance of a stone to a goal can depend on the location of the man. Consider Figure 15. While the stone is only 3 squares away from the goal, 7 pushes are required to move the stone to the goal (two pushes away from the goal are needed for the man to reach the opposite side of the stone). Therefore,




| stones on | goals on *Bb* | *Ic* | *If* |
|---|---|---|---|
| *Cc* | 2 | **6** | 9 |
| *Hb* | **6** | $\infty$ | $\infty$ |
| *Id* | $\infty$ | 1 | **2** |

Figure 14: Minimum Matching Example


Figure 15: Distance Depends on the Position of the Man

the stone distances used for minimum matching take the current position of the man into account.

## A.2 Transposition Tables

Transposition tables are usually implemented as (large) hash tables. The hash keys we use incorporate only the exact stone positions. To match an entry, the keys must be identical. Since the position of the man is important, a second test is performed. The locations of the man in both positions must be connected by a legal man path. Thus multiple positions that differ only in the man's location may map to identical transposition entries. This simplification is possible because we only optimize stone pushes.

Transposition tables can handle cycle detection. The table entry for a position can be flagged before doing a search from that position, and the flag removed after the search completes. If a search ever reaches a flagged state then a cycle has occurred.

## A.3 Move Ordering

The information used to order moves can come from different sources, but is usually domain-dependent knowledge. Sometimes knowledge gathered during the search (*e.g.*, tree sizes or tree depths) can be useful. In the case of iterative

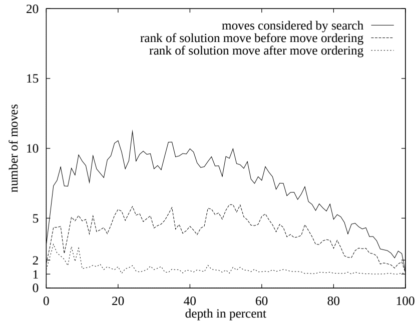



Figure 16: The Effect of Move Ordering

deepening, move ordering information is passed from one iteration to the next by means of the transposition table.

*Rolling Stone* uses a move-ordering scheme that we call *inertia*. Analysis of solution sequences shows long runs where the same stone is repeatedly pushed. Hence, moves are ordered to preserve the *inertia* of the previous move in the following way:

1. Inertia moves are considered first.
2. Then all the moves that decrease the lower bound (optimal moves) are tried, sorted by the distance from the stone pushed to the goal it is targeted to, with close stones first.
3. Then all the "non-optimal" moves are tried, sorted similarly.

Figure 16 shows the effect of move ordering.[4] The vertical axis shows the number of moves considered. The horizontal axis shows the depth of the node in the tree in percent. The upper curve indicates the average number of moves considered by the program.[5] The middle curve shows where the actual solution move is located in the list returned by the move generator. Not surprisingly, the solution move is in the middle of the move list on average. The lower curve shows that inertia ordering results in solution moves being placed closer to the front of the move list. Move ordering becomes more accurate with decreasing distance to the goal. In fact, after reaching a depth of about 20% of the solution length, the move ordering becomes close to perfect. At the start of a Sokoban problem, with many complications in the maze, seemingly good moves might actually lead to deadlocks. Many of the problems in the test suite are designed in such a way that an initial "knot" has to be freed up. This can usually be

[4]The data was compiled from all the positions on the solution paths for the 57 problems that *Rolling Stone* can solve.

[5]Some of the legal moves are discarded immediately because they lead to trivially provable deadlocks. These moves are not included in the graph. See Appendix A.4 for more detail.

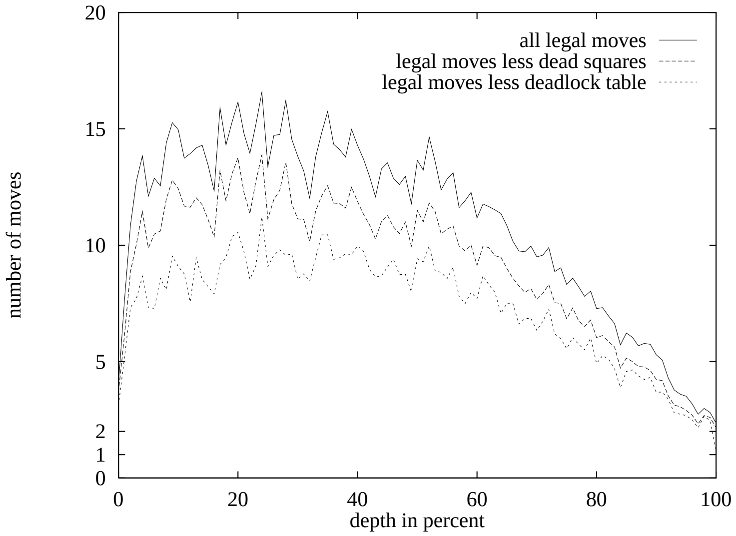



Figure 17: Effect of Deadlock Tables

achieved only with moves that the lower-bound estimator views as being non-optimal. After the knot is untangled, a "mop-up phase" commences during which stones are simply pushed to the goals. This is where our heuristic excels.

## A.4 Deadlock Table

Figure 17 shows the number of moves in the move list versus the depth of the tree. Only positions on paths to solutions were used to generate the data for the figure to avoid pathological cases. The top curve shows how many legal moves those positions have, averaged over all test positions. The second curve shows how many legal moves exist that do not directly push stones onto dead squares (squares from which no goal is reachable, such as moving a stone into a corner). Note that this simple test reduces the effective branching factor by about 20%. The third curve shows how many moves are actually considered after screening moves with the deadlock tables. The savings are similar to the simple dead-square checking, almost an additional 20%.

## A.5 Tunnel Macros

Figure 18 illustrates the impact of the move sequence $a$-$b$-$c$ being treated as a tunnel macro. Instead of exploring every possible interchanging combination of moves $a$, $b$, $c$ of one stone, and $d$, $e$, $f$ of another stone, most of the search tree can be eliminated by treating the sequence $a$-$b$-$c$ as a single move. The macro also has the effect of reducing the depth of the tree.

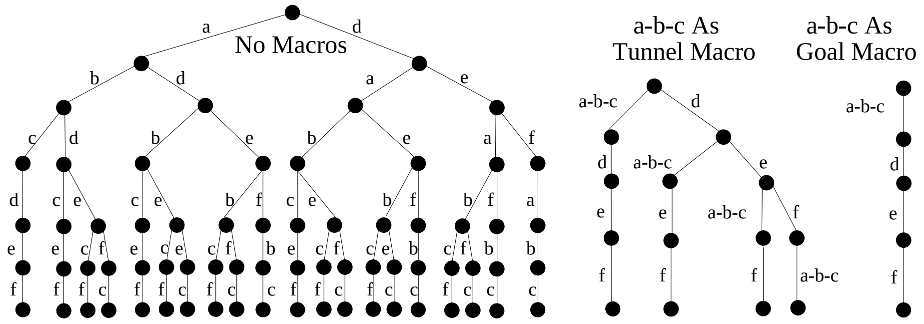



Figure 18: Impact of Macro Moves

## A.6 Goal Macros

Figure 18 illustrates the impact of goal macros on the search.

The goal macros in the current implementation have limitations. One underlying assumption is that no stone will leave the goal area once inside. Problem #50, for example, cannot be solved without pushing stones through the goal area. A second, even stricter assumption is that once a stone is inside the goal area, it will never move again. This does not allow for *parking* inside goal areas. Sometimes it is necessary to leave a stone in a key position inside the goal area until later in the solution, when it can finally be pushed to its goal square. Another limitation is that a goal area containing several entrances is often a travel area for the man; certain parts of the maze need to remain unblocked to allow the man to push stones in a certain way outside the goal area. Problem #38 is an example where the careless packing of stones in the goal area can obstruct the man from other areas of the maze.

These problems show that goal macro creation is still far from being solved satisfactorily. Interactions between the goal area and the outside parts of the maze make it difficult to create good goal macros. However, their positive impact in the problems where they work is so large that any high-performance Sokoban program needs to use this type of knowledge in one form or another.

## A.7 Goal Cuts

We implemented a scheme that will cut moves only after a stone push towards its macro move is explored. The search backs up the cut information, instead of statically trying to deduce that such a move exists in a certain position. This could potentially lead to missed opportunities for additional cuts if other moves are explored before the one that leads to the goal cut. Since ordering puts moves that are close to goals towards the front of the move list, lead-off moves to goal macros are likely to be considered early in the list.

## A.8 Pattern Searches

Figure 19 shows the pseudo code for pattern searches. 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 (i.e. add a stone that violates one of the preconditions). 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 to square $A$ is used.[6] If none of those exists, the pattern search returns without finding a deadlock.

After including the next stone, PIDA* is called again, 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 than the lower-bound estimate of the position, the current pattern is stored with a corresponding lower-bound increase.

Figure 5 (page 13) shows a simple position, before and after the move *Gd-Fd*. The question is whether this move introduces a deadlock. Figure 6 (page 13) 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 6a). A PIDA* search reveals a 5-move solution (*Fd-Fc-Ec-Dc-Cc-Bc*) which is also the StonePath, and sets ManPath to the squares needed by the man (*Gd-Ge-Fe-Fd-Gd-Gc-Fc-Ec-Dc-Cc*). 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 6b). PIDA* will solve the two-stone maze 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 6c) and another search is commenced. A solution will be found, requiring a fourth stone to be added (Figure 6d).

The fourth call to PIDA* will return no solution and announce a deadlock with this pattern of four stones.

## A.9 Relevance Cuts

When judging how two squares in a Sokoban maze influence each other, Euclidean distance is not adequate. Taking the structure of the maze into account would lead to a simple geographic distance which is not proportional to influence

[6] *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.

```
PatternSearch( From, To ) {
  clear TestMaze;
  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;
    /* Call to PIDA* modifies SolLength, ManPath and StonePath */
    solution = PIDA*( TestMaze, SolLength, ManPath, StonePath );
    /* Test for a deadlock */
    if( solution == NO AND NOT EffortLimit() ) {
      GeneralizeAndAddPattern( TestMaze, infinity );
      break;
    }
    /* Test for a lower-bound increase */
    if( solution == YES ) {
      lb = LowerBound( TestMaze );
      if( SolLength > lb )
        GeneralizeAndAddPattern( TestMaze, SolLength - lb );
    }
  }
}
```

Figure 19: Pseudo Code for Pattern Searches

either. For example, consider two squares connected by a tunnel; the squares are equally influencing each other, no matter how long the tunnel is. Elongating the tunnel without changing the general topology of the problem would change the geographic distance, but not the influence.

The influence measure should reflect the following properties:

**Alternatives:** The more alternatives that exist on a path between two squares, the less the squares influence each other. That is, squares in the middle of a room, where stones can go in all 4 directions, should decrease influence more than squares in a tunnel, where no alternatives exist.

**Goal-Skew:** For a given square $sq$, any squares on the optimal path from $sq$ to a goal should have stronger influence than squares off the optimal path.

**Connection:** Two neighboring squares connected such that a stone can move between them should influence each other more than two squares connected such that only the man can move between them.

**Tunnel:** In a tunnel, influence remains equal, regardless of length.

Our implementation of relevance cuts uses small off-line searches to statically precompute an $InfluenceTable$ containing the influence values for each square of a $20 \times 20$ maze to every other square in the maze [7, 12]. Between every pair of squares, a breadth-first search is used to find the path(s) with the largest influence. The algorithm is similar to a shortest-path finding algorithm, except that we use influence here and not geographic distance. The smaller the influence number, the more two squares influence each other. Our approach is quite simple and can undoubtedly be improved. For example, influence is statically computed. A dynamic measure, one that takes into account the current positions of the stones, would undoubtedly be more effective.

## A.10 Overestimation

Since the pattern searches are limited in certain ways to keep them tractable, the correct size of the penalties and shape of the patterns might not be known. Therefore, the patterns represent incomplete knowledge. Furthermore, when patterns are matched, only some of the penalties can be used to preserve admissibility. However, the presence of matching patterns that are not included in the lower-bound calculations suggests that there may be additional complications in the current position. Not using the penalty of such a pattern is equivalent to ignoring available knowledge. The following describes the best of our attempts to use the knowledge contained in the patterns that match a position. It was achieved after a significant effort spent on experimentation and tuning. We call this method *maximum partial penalties*. More straightforward ways of overestimation suggested in the literature were not effective [16].

One simple overestimation idea is to sum the penalties for all the patterns that match in a position (a worst-case scenario). Predictably, this does not perform well. Instead of attributing penalties to patterns, they can be assigned to stones in the maze. The penalty of a matched pattern is split equally among all the stones contained in that pattern. For each stone the maximum of these partial penalties is stored. The total penalty of a position is the sum of all the maximum partial penalties for each stone. Thus, every stone involved in a penalty pattern contributes to the total penalty assigned to a stone configuration. To tune the overestimation further, the penalty is scaled by a factor $s$ (currently set to 1.8, as determined by experimentation). A final rounding step assures that the total penalty is an even number to preserve the parity property of the heuristic.

Adding a limited penalty to the heuristic estimation of the distance to the goal will only delay the examination of a node to a later iteration. If no solution can be found, the threshold will increase until the position's lower-bound estimate is not enough to cause a cutoff anymore. The exploration of the node is only *postponed*. This is in stark contrast to forward pruning with fixed rules, such as deterministic relevance cuts, that will prune the same node in every iteration. Because new patterns are added and useless patterns are dropped, the decision to postpone a node change dynamically over the course of a search as new knowledge is found or other knowledge is discarded.

## A.11 Rapid Random Restart

Each iteration of IDA* is in principle a restart with a different parameter setting: the threshold. However, in the classic IDA* the threshold is only increased after an iteration is exhaustively searched. When RRR aborts an iteration early, it is unclear whether to restart with the same iteration or to increase the threshold. *Rolling Stone* uses a “double impatience” approach. If a certain number of restarts in a specific threshold iteration have not produced a solution, the threshold is increased. Furthermore, with each new restart within an iteration the randomization of the move ordering is increased. This can be justified by simply stating that if the move ordering was good the solution would have been found by now. With each restart our confidence in the move ordering shrinks and more randomization is used. When the threshold is increased, the randomization is reduced to 0 again, because it is assumed that no solution existed for the lower threshold.