Using Abstraction for Planning in Sokoban

Adi Botea, Martin Müller, and Jonathan Schaeffer

Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2E8
{adib, mmueller, jonathan}@cs.ualberta.ca

Abstract. Heuristic search has been successful for games like Chess and Checkers, but seems to be of limited value in games such as Go and Shogi, and puzzles such as Sokoban. Other techniques are necessary to approach the performance that humans achieve in these hard domains. This paper explores using planning as an alternative problem-solving framework for Sokoban. Previous attempts to express Sokoban as a planning application led to poor performance results. Abstract Sokoban is introduced as a new planning formulation of the domain. The approach abstracts a Sokoban problem into rooms and tunnels. This allows for the decomposition of the hard initial problem into several simpler sub-problems, each of which can be solved efficiently. The experimental results show that the abstraction has the potential for an exponential reduction in the size of the search space explored.

1 Introduction

Heuristic search has led to impressive performance in games such as Chess and Checkers. However, for some two-player games like Go and Shogi, or puzzles like Sokoban, approaches based on heuristic search seem to be of limited value. For example, the search effort required to solve Sokoban problems increases exponentially with the difficulty of the problem [6]. Obviously, waiting until computers become 10-fold faster is not the best way to address this problem. New approaches are needed to deal with such hard domains, where humans still perform much better than the best existing programs.

Planning can be a powerful alternative to heuristic search. For example, humans are very good at planning in games, and not quite as good at searching. The last few years have seen major advances in the capabilities of planning systems, in part stimulated by the planning competitions held as part of the AIPS conference [1]. However, there are only a few results in the literature about using planning in a game-playing program [12, 13]. Part of the explanation for this is that the performance-driven aspect of many game-playing research efforts is more conducive to short-term objectives (i.e., what will make an impact in the next tournament), rather than long-term goals. In single-agent search, *macro moves* can be considered as simple plans and are, arguably, the most successful planning idea to make its way into games/puzzle practice. Macro moves are

sequences of moves that are treated as a single, more powerful move. They can dramatically reduce the search tree by collapsing a sub-tree into a single move. The idea has been successfully used in the sliding-tile puzzle [10]. Two of the most effective concepts used in the Sokoban solver *Rolling Stone* are macro moves (tunnel and goal macros) [6].

Sokoban is an excellent test-bed for planning research, as the domain is recognized as being hard not only for humans, but also for artificial intelligence (AI) applications [6]. In Sokoban, a man in a maze has to push stones from their current location to designated goal locations. The problem is difficult for a computer for several reasons including deadlocks (positions from which no goal state can be reached), the large branching factor (can be over 100), long optimal solutions (can be over 600 moves), and an expensive lower-bound heuristic estimator, which limits search speed. Sokoban problems are especially challenging because they are composed to be as difficult as possible. Many problems are combinations of wonderful and subtle ideas, and finding the solution may require substantial resources — both for humans and computers. Sokoban has been shown to be PSPACE-complete [4]. Junghanns' Sokoban solver *Rolling Stone* is able to solve two thirds of the standard 90-problem test suite[1] [6, 9]. There is also a strong Japanese Sokoban community whose best program is written by a researcher who calls himself or herself *deep green* [6].

In this article we introduce a novel planning approach to the game of Sokoban, with the goal of overcoming the limitations of previous approaches based on heuristic search. There are at least two ways to represent Sokoban as a planning domain, and there is a huge difference in terms of efficiency between them. The first, naive approach is to translate all the properties of the domain to a planning representation. We call this the *plain* Sokoban domain representation. For instance, a regular move in Sokoban becomes an action in the planning domain. Previous experiments based on this representation generated poor results [11]. The second approach, which we will use in the paper, is to apply abstraction to the domain, so that the planner need only solve the simpler abstracted problem. We call this the *abstract* Sokoban domain. Our abstract Sokoban representation uses a preprocessing phase to decompose the puzzle into two types of objects: *rooms* and *tunnels*. At the abstract level, the maze is reduced to a graph of rooms linked by tunnels. The rooms are treated as *black boxes*. They have abstract states that model their internal configuration. The planning actions, which are essentially macros of regular Sokoban moves, refer to moving a stone from one object (room or tunnel) to another, rather than simply pushing one stone to an adjacent free square. Planning is done at this high abstraction level, which has a much smaller search space. By splitting the problem into a local component (moves within a room) and a global component (moves between rooms and tunnels), the initial search space is transformed into a hierarchy of smaller spaces, each with a corresponding reduction in the search effort required. The approach is similar to *hierarchical* A^* [5].

[1] The test suite is available at http://xsokoban.lcs.mit.edu/xsokoban.html.

In our approach, the abstract planning problem is solved using the standard planner TLPlan [3]. Our initial results are very encouraging, showing a substantial reduction in the planning search effort required. In effect, the search problem has been split into two much smaller searches: a local preprocessing search and a global search.

The remainder of the paper is structured as follows: In the next section we introduce the domain of Sokoban. Section 3 summarizes our general planning framework for Sokoban and Section 4 provides details about our novel abstract representation to the game. Section 5 presents experimental results and Section 6 contains conclusions and ideas for further work.

2 The Sokoban Domain

Sokoban is a single player game created in Japan in the early 1980s. The puzzle consists of a maze which has two types of squares: inaccessible *wall squares* and accessible *interior squares*. Several *stones* are initially placed on some of the interior squares. There is also a *man* that can walk around by moving from his current position to any adjacent free interior position. A *free* position is an interior square that is not occupied by either a stone or the man. If there is a stone next to the man and the position behind the stone is free, then the man can push the stone to that free square. The man moves forward to the initial position of the stone. The goal of the game is to push all the stones to some specific marked interior positions called *goal squares*. Figure 1 shows an example of a Sokoban problem.

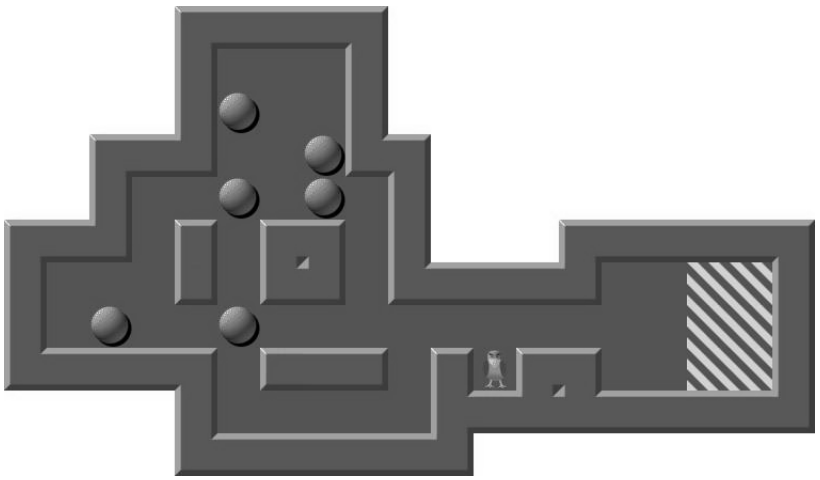


Fig. 1. Problem #1 in the standard 90 problem Sokoban test suite. The six goal squares are the marked ones at the right end of the maze.

One of the most interesting features of Sokoban, which contributes to both its hardness and its beauty, is the presence of *deadlock*. A deadlock refers to an unsolvable position, when there is at least one stone that can never be pushed to a goal square. There are many types of deadlocks in Sokoban, from the simplest, which affect only one stone, to some very subtle blockades, that can involve many stones scattered over the whole puzzle. Humans who create Sokoban levels fully exploit this property of the game to obtain difficult problems. Often, the key issue in solving a problem is to detect some potential deadlocks and develop a strategy that avoids them. In computer programs, the quality of the deadlock detection algorithm decisively affects the efficiency of the whole program. If a deadlock is detected, then the search tree is pruned and the search effort is significantly reduced. Since there is no goal node in the subtree of a deadlock state, this kind of pruning is always safe. In contrast, if a deadlock exists but is not detected, there is the danger that the corresponding node will be expanded and the search continues in the resulting subtree for a long time, with no results.

In Sokoban, the solution length can be defined in two ways: either the man movements or stone pushes can be counted. As a consequence, there are two types of optimal solutions. However, since the domain is hard enough, we don't require optimal solutions (nor do humans); *any* solution will do. This relaxation allows us to define important equivalence relationships between local configurations of the maze that lead to a simplification of the initial problem. All the equivalences defined in this paper, which support the introduction of the key concept of abstract states, ignore the optimality condition. If desired, non-optimal solutions can be improved in a post-processing phase.

3 Planning in Sokoban

While many domains that are used as a test-bed for current planning systems are quite simple, Sokoban is recognized in the planning community as a hard planning domain. General purpose planners cannot deal with the game at a satisfactory level. Junghanns and Schaeffer point out the limited performance that state-of-the-art fully automated planners can achieve in Sokoban [8]. One necessary step in improving the planners' performance in Sokoban is to use additional domain-specific knowledge, such as deadlock detection. This is why we chose to use Fahiem Bacchus' TLPlan, one of the few planners that allows users to plug-in libraries that contain domain-specific code [3].

The main Sokoban-specific functions that we implemented deal with:

- Deadlock: Since deadlocks affect the search efficiency, we introduced a quick test to detect local deadlock patterns. We use Junghanns' database, which contains all the local deadlock patterns that can occur in a 5x4 area [6]. Although this enhancement was an important gain, the problem of deadlocks was far from being solved.
- Heuristic evaluation function: Since the heuristic function has a big impact on the quality of the search algorithm, we used a custom heuristic, called *Minmatching*, which is also used in *Rolling Stone* [6].

– State equivalence: The definition of equivalence of states also has a special importance in Sokoban. For illustration, we provide the following example. Suppose that two states have identical stone configurations but different man positions. Suppose further that the man can walk from one position to the other. The two states are equivalent (unless we seek optimal solutions that minimize man movements). We want to encode this relationship, as it leads to an impressive reduction of the search space. For this reason TLPlan was enhanced with functionality such that the planner now supports custom functions to check the equivalence between states.

To enhance system performance, the plain planning representation of Sokoban, which consists of translating the game properties into a planning language such as STRIPS, was also replaced by a partially abstracted representation, called *tunnel* Sokoban. In tunnel Sokoban we identify all the tunnels present in the maze and treat them at a high abstract level (as shown in Section 4). All the possible configurations of a tunnel are reduced to a few abstract states and planning actions such as parking a stone inside a tunnel or or pushing a stone across a tunnel are defined.

The above modifications led to a program that is much more efficient than plain translation with no domain-specific functions. However, the system could not deal with even moderately complex puzzles. Only one from the standard test suite of 90 problems can be solved by this approach. The limitations are present because, even though the approach deals with the factors that make Sokoban hard, it is not powerful enough to do it efficiently. The search space is reduced by the tunnel macros, but the reduction is not big enough to achieve reasonable performance. Moreover, although small deadlocks are detected, there are many larger deadlock patterns that still have to be dealt with. We therefore needed a method to further reduce the search space and deal with deadlocks more efficiently. For this reason we introduced a new representation of the domain, *Abstract Sokoban*, which uses abstraction not only for tunnels but also for the rest of the maze.

4 Abstraction in Sokoban

Abstract Sokoban is a novel and highly abstracted representation of the game that replaces the initial huge search space by several smaller search spaces. The search is decomposed into several local searches and one global search. In other words, the initial problem is transformed into several simpler problems in a divide-and-conquer manner. In this approach planning is done at the global level, in a much simpler abstract space. To obtain the abstraction of the problem, the first step is to perform an initial analysis of the maze that decomposes it into objects of two types: *rooms* and *tunnels*. The next two sub-sections provide details about maze and problem decomposition.

4.1 Maze Decomposition

The concept of a tunnel macro was introduced in *Rolling Stone* [6]. In the current version of our program, which is called *Lisa*, tunnels of 7 abstract types are detected, including some trivial ones that have length 0. Figure 2 shows a few examples of tunnels.

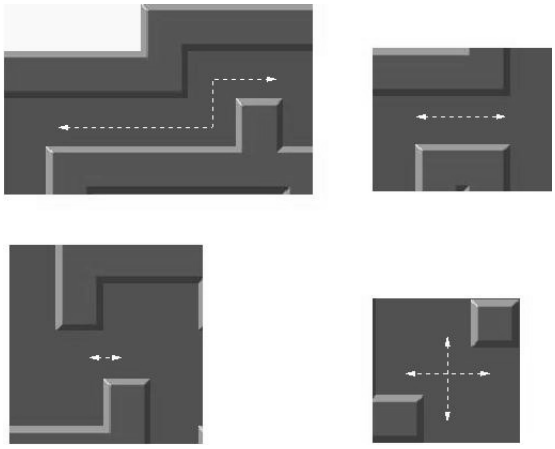


Fig. 2. Various types of tunnels.

Tunnels are simple objects that don't need much processing. Their properties can be obtained with little computational effort. We define abstract states for tunnels that characterize their stone configuration. While a tunnel can have many configurations (the longer it is, the larger the number of configurations it can have), the number of possible abstract states is very small. Depending on its type, a tunnel can have between 1 and 3 possible legal abstract states, as we show in the following example. Since the length of the tunnel in the lower-left corner in Figure 2 is 0 and therefore no stone can be temporarily *parked* inside it, there is only one abstract state that the tunnel can have, corresponding to the situation when it is empty. The *straight* tunnel in the upper-right corner of Figure 2 can have two abstract states: either the tunnel is empty or there is a stone parked inside it. In the latter case it is not important where exactly the stone is placed — all the configurations are equivalent and they are merged into the same abstract state. The tunnel in the upper-left corner of Figure 2 has three possible abstract states. One abstract state is defined for the empty tunnel. There is also one state for the situation when a stone is pushed inside through the left end of the tunnel. The last abstract state is for the case when a stone was pushed inside through the right end of the tunnel. These last two abstract states have different properties. In the first case, the stone can be taken

out through the left end only, while in the second case the same action can be done through the right end only.

After computing tunnels, all the remaining interior points are grouped together in connected components called *rooms*. Two points belong to the same room if and only if there is a connection between them that does not cross any tunnel. We define abstract states for rooms in the following way: One abstract state represents all the room configurations that can be obtained from each other in such a way that neither any stone nor the man leaves or enters the room (we say that these configurations are *equivalent*). We call a room that contains at least one goal square a *goal room*. We emphasize once more that the definition of abstract states for rooms and tunnels preserves solvability but not optimality of solutions.

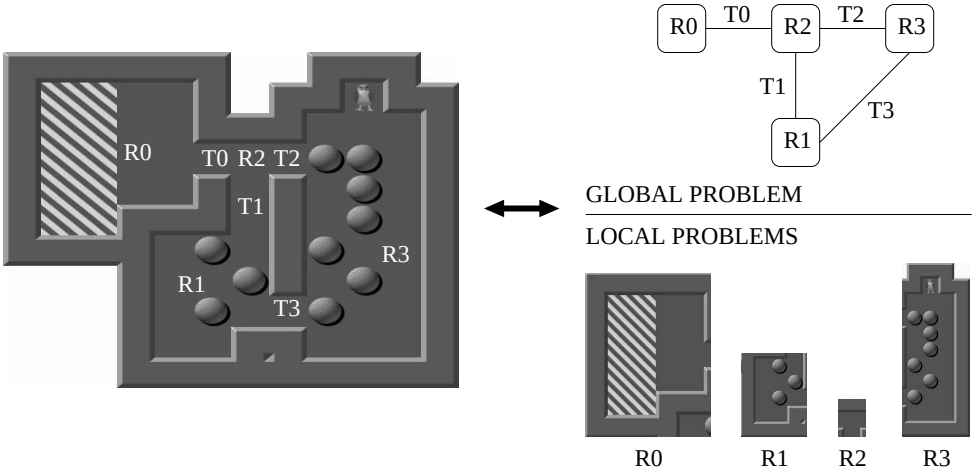


Fig. 3. A problem (#6 of the test suite) is decomposed into several abstract subproblems. There is one global problem as well as one local problem for each room. Rooms and tunnels are denoted by R and T, respectively.

4.2 Problem Decomposition

Once the maze is split into rooms and tunnels, the initial problem can be decomposed into several smaller ones, as shown in Figure 3. At the global level, a search problem is transformed into a graph (R_i, T_j), where the nodes R_i represent rooms and the edges T_j represent tunnels. Besides the global planning problem, we also get several local search problems, one for each room. The complexity of a local problem depends on both the size and the shape of a room. The local problem attached to the one-square room $R2$ is much simpler than the one attached to the largest room $R3$. While the complexity of the initial

problem increases exponentially with the size of the maze, the complexity of the local problems increase exponentially with the size of the rooms only. Moreover, since the local computation is done only once, its results can be reused many times during the global-level search.

Local Problems Local problem for each room is to compute and provide quick access to information needed at planning time. So at the local level, we compute the graph of abstract states of the room. For each abstract state, some properties are also calculated that will be used to check action preconditions at planning time. For instance, for an abstract state we might want to know whether we can push one more stone into the room through a certain tunnel. Another important result of the local computation is that all the states that contain local deadlock patterns are detected and eliminated from the search space.

The steps of the local abstraction are summarized as follows.

Algorithm 1 Rooms Local Processing

1: remove dead squares;
2: build local move graph;
3: mark deadlock configurations;
4: run SCC (strongly connected components) algorithm, find abstract states (AS);
5: determine properties of AS, find abstract moves (actions) for AS.

The removal of dead squares is done earlier, during maze preprocessing. Two types of squares are marked as dead: Some squares are completely useless and we remove them from the maze (e.g., tunnels that have one end closed). As done in *Rolling Stone* [6], we also mark the *stone-dead* squares, where the man can go but stones cannot be pushed because of deadlock. Next, starting from the initial stone configuration of the room, we compute the *local move graph* of all possible configurations that the room can have. We can use the local move graph to detect all deadlock configurations that can occur locally in the room. We consider a local position to be deadlocked if we cannot clear the room of stones or, equivalently, if there is no path in the local move graph from that position to the empty position. We mark positions using retrograde analysis, starting from the empty position, which is marked as legal. After all the *n-stone* configurations have been marked as either legal or deadlock, we go to the next level and mark the $(n+1)$-*stone* configurations: if there is a path from the current $(n+1)$-*stone* position to a legal *n-stone* or $(n+1)$-*stone* position, then the current position is legal too. Otherwise, it is deadlocked.

In the next step, to obtain the abstract states of the room, we run the SCC algorithm, which computes the strongly connected components of the local move graph. Each strongly connected component becomes an abstract state. In this way, equivalent configurations are merged together into the same abstract state. All deadlock states are mapped into one abstract deadlock state. To be able to

check action preconditions at the planning level, for each abstract state some predicates are also computed (such as "can push one more stone inside the room through entrance X"). When the value of these predicates is $TRUE$, we also compute the resulting abstract states after we perform the corresponding actions, such as pushing one stone, pulling one stone, etc.

For rooms that have up to 15 non-dead squares we are usually able to complete the local processing described above. However, for large rooms it is not feasible to compute all the possible abstract states. Our current program currently handles only cases where a large room is directly linked to a goal room. Given an abstract state of the large room, first we check whether we can push a stone to the goal room. If so, we accomplish this action and don't compute preconditions of any other possible actions. This optimization leads to a significant reduction of the local problem complexity. Otherwise, preconditions of other possible actions are computed. To further speed up the computation, we store minimal deadlock patterns and maximal legal patterns. Any position that contains a deadlock pattern is illegal. Any position that is contained in a legal pattern is legal. How to cope with large rooms in general is still an open question and constitutes one of our main directions for future research.

To summarize, the original problem can be abstracted into rooms and tunnels. The internal state of a room can be computed by retrograde analysis as part of the preprocessing stage. Our simplistic approach to computing rooms turns out to be the bottleneck in performance (see Section 5). However, it was done this way for simplicity, since it was more important to determine the viability of the abstraction approach than it was to maximize performance. Obviously, not all the states computed in the preprocessing will be needed during a planning search. A better approach would be to compute only subsets of the room's state as needed by the planner.

Global Problem In the global problem, the abstraction is obtained by mapping the maze to a small graph of rooms connected by tunnels. This global problem is solved by planning in abstract Sokoban. To run our experiments, we used TLPlan enhanced with domain-specific knowledge, as described in Section 3. Planning actions now refer to moving one stone from one room or tunnel to another, rather than simply pushing one stone or moving the man by one square. Objects involved in a stone movement change their abstract states after the corresponding action is completed. Table 1 summarizes the action types in the 3 planning approaches discussed in this paper: plain, tunnels, and abstract Sokoban. While the actions always refer to pushing a stone from one node to another, the meaning of nodes is different: regular squares in plain Sokoban, tunnels and squares in tunnel Sokoban, and rooms and tunnels in abstract Sokoban.

The planning goal is expressed as the conjunction of the conditions that each goal room should obey. The overall goal is reached when all goal rooms are in the abstract state where all its goal squares are occupied by a stone. In Figure 3, $R0$ is the only goal room of the maze and it contains 8 goal squares.

Representation Type	Nodes	Actions
plain Sokoban	squares	$push(node_1, node_2)$
tunnels Sokoban	squares + tunnels	$push(node_1, node_2)$ $cross(tunnel)$
abstract Sokoban	rooms + tunnels (objects)	$push(node_1, node_2)$

Table 1. A comparison of actions in the three Sokoban planning representations.

Compared to plain Sokoban and tunnel Sokoban, the abstract representation shows greater promise for addressing the game as a planning problem. As will be shown in Section 5, problems that cannot be solved by the first two approaches are easily handled in the abstract one. The two main factors that explain this important improvement are search space size and deadlock detection. In abstract Sokoban the search space is much smaller than in plain and tunnel Sokoban. Both branching factor and solution length are greatly reduced as a result of the abstraction. One abstract move or action is typically composed of several normal Sokoban moves. Planning in abstract Sokoban is also simpler because there are less deadlocks to deal with. All the deadlocks that can occur inside one room are handled by the local analysis. A move never goes into a local deadlock. The only deadlocks that still have to be considered at the planning level are the large ones that involve interactions between several rooms and tunnels.

5 Experimental Results

In this section we present experimental results for solving Sokoban using abstraction. Since it is more meaningful to run the tests on real problems (as opposed to using toy problems), we have chosen 10 out of the 90 levels from the standard test suite for our experiments. The 10 problems, which are shown in Appendix A, are solvable by our abstract Sokoban system. Six problems (1, 2, 6, 7, 17, 80) are composed of small rooms and therefore the local preprocessing can be done completely. The other four (3, 4, 5, 9) contain both large and small rooms. Our current framework for large rooms can successfully cope with these problems. These results should be viewed as preliminary, as we scale the system to handle the remaining 80 problems. Our system, called *Lisa*, uses abstract Sokoban and TLPlan. To evaluate its performance, we compared *Lisa* with two other approaches: TLPlan + tunnel Sokoban (or simply tunnel Sokoban) and *Rolling Stone*, which is the best search-based Sokoban solver available.

Figure 4 shows how the solution length is reduced in *Lisa* and *Rolling Stone*. *SP* represents the number of stone pushes in the solutions found by *Rolling Stone*. These are the best values that we know of, and in many cases they are equal to the optimal values. Since *Rolling Stone* uses macro moves in its tunnel

macros and goal macros, RS, which is the length of the solution found by *Rolling Stone*, is smaller than SP. AS, which stands for abstract Sokoban, is the length of the solution found by *Lisa* (i.e., the number of planning actions). AS is much smaller than SP, as one planning action in abstract Sokoban corresponds to several regular moves. Since the solution length reduction is a measure of how the search space is reduced, the graph suggests that our global search space is smaller than the main search space used in *Rolling Stone*. This is an important result, as it promises an exponential reduction in the search space.

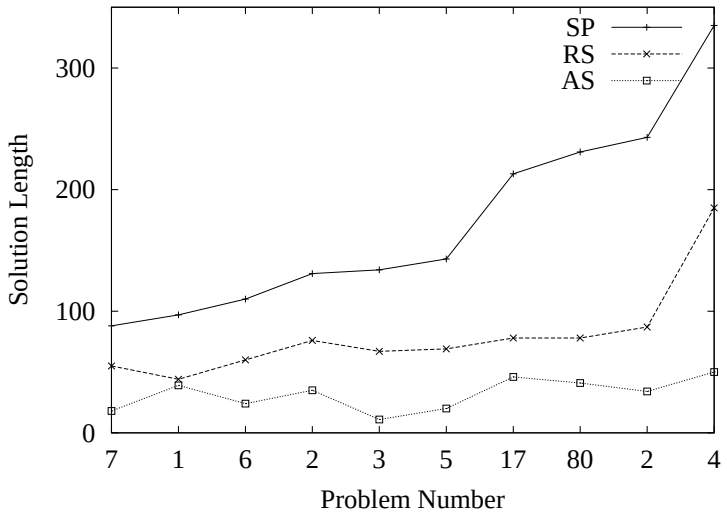


Fig. 4. Solution length in *Lisa* (AS) and *Rolling Stone* (RS). SP is close to the number of stone pushes in the optimal solution.

When using tunnel Sokoban, TLPlan can seldom solve a problem entirely (in our test subset, only the simplest problem, which has 6 stones, can be solved). For this reason, we solve sub-problems of the initial problem. A sub-problem is obtained by removing from the initial configuration some stones as well as an equal number of goal squares. Figure 5 reveals how the effort for solving sub-problems of Problem #6 evolves for *Rolling Stone* (RS), tunnel Sokoban (TS), and *Lisa* (AS). Here the number of expanded nodes in the main search are plotted (note the logarithmic scale). Tunnel Sokoban is only able to solve sub-problems with 7 or less stones. For the next sub-problem (8 stones) this system didn't find a solution after running for more than 20 CPU hours. Compared to *Rolling Stone*, *Lisa* achieves a reduction by a factor that remains stable over the whole set of sub-problems of Problem #6. This graph also illustrates how dramatically the

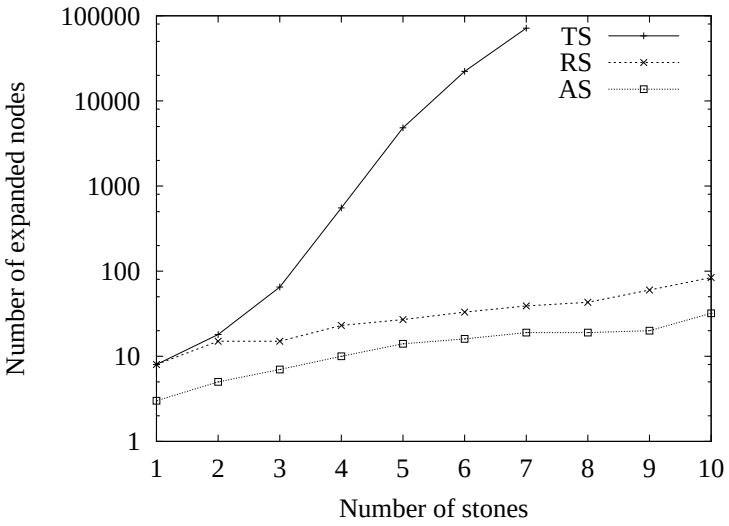


Fig. 5. The expanded nodes in the main search for abstract Sokoban (AS), tunnel Sokoban (TS), and *Rolling Stone* (RS). The values were obtained for sub-problems of Problem #6.

search effort can be reduced by the choice of problem representation and solving method.

Table 2 presents a more detailed comparison between abstract Sokoban and tunnel Sokoban. The data demonstrates a huge difference in terms of efficiency between the two approaches. Even if the numbers in the PPN column seem to be large, the preprocessing is fast. The processing cost for a node in the local search space is low, since there is no heuristic function to be computed. of the local search space are cheap (e.g., there is no heuristic function to be computed).

Table 3 shows a comparison between *Lisa* and *Rolling Stone*. As in the case of abstract Sokoban, *Rolling Stone* also uses two types of search and, to be able to perform a measurement, we consider a one-to-one correspondence between the search spaces in the two approaches. At the global level *Rolling Stone* does the so-called top-level search, whose purpose is to find a goal state. We denote the number of searched nodes at the top level by TLN and compare it to PlN from abstract Sokoban. There is also the pattern search in *Rolling Stone*, whose main goal is to determine deadlock patterns and find better bounds for the heuristic function [7] (PSN is the number of expanded nodes in the pattern search). We compare pattern search in *Rolling Stone* with local preprocessing in *Lisa*, as they both are means to simplify the main search.

The good news is that, for many problems, the number of planning nodes PlN is smaller than PSN, which supports the claim that our global search space is smaller than the one considered by *Rolling Stone*. In contrast, when

Sub-problem	Abstract Sokoban PlN	PPN	Time	Tunnel Sokoban PlN	Time
1(6)	71	1,044	1.57	10,589	126.24
2(6)	24	61,113	0.93	80,740	9490.21
3(7)	8	482	0.12	77,919	12248.66
4(6)	9	41,065	0.80	27,514	3061.94
5(6)	7	404	0.20	53,141	11733.83
6(7)	19	54,317	1.06	71,579	8189.77
7(8)	13	26,011	0.75	132	0.88
9(6)	13	245	0.25	35,799	4883.55
17(5)	1047	306,224	29.63	14,189	391.42
80(6)	10	395,583	3.02	14,266	949.98

Table 2. Abstract Sokoban vs. tunnel Sokoban. Sub-problem $x(y)$ is obtained from problem x by placing y stones in the maze. The sub-problems listed are the largest that tunnel Sokoban can solve. PlN is the number of expanded nodes during the planning search. PPN is the number of expanded nodes during the preprocessing phase. The time is measured in seconds.

Problem	Abstract Sokoban PlN	PPN	Time	Rolling Stone TLN	PSN	Time
1	71	1,044	1.57	50	1,042	0.14
2	635	62,037	16.10	80	7,530	0.63
3	12	19,948	2.04	87	12,902	0.23
4	128	69,511	3.20	187	50,369	3.27
5	36	297,334	23.14	202	43,294	1.72
6	36	54,414	1.37	84	5,118	0.31
7	54	35,813	1.57	1,392	28,460	1.37
9	35	7,607	1.01	1,884	436,801	22.17
17	8091	444,073	166.98	2,038	29,116	2.23
80	47	877,914	4.56	165	26,943	2.25

Table 3. Abstract Sokoban vs. *Rolling Stone*. PlN is the number of expanded nodes in the planning search, PPN is the number of preprocessing nodes, TLN is the number of top-level nodes, and PSN is the number of nodes in the pattern search. The time is measured in seconds.

analyzing PSN and PPN, we infer that *Rolling Stone* is more efficient from this perspective. Indeed, *Rolling Stone* is a finely tuned application, developed over several years of effort, while there is still room to improve our local computation. Although preprocessing search is very fast, rooms can be big. We therefore need ways to get around the performance bottleneck induced by preprocessing. Some ideas have already been implemented in our system, as shown in Section 4. Problems #3, #4, and #9, which contain large rooms, are solved efficiently by our program. However, there are many improvements that can be further added in order to process efficiently large rooms. *Rolling Stone* is also faster than our system, with the exception of Problem #4 and Problem #9. The overhead is determined by the local processing as well as the usage of a general purpose planner. The version of TLPlan that we used in our experiments is often two orders of magnitude slower than the latest experimental version of the planner [2]. On the other hand, our method has the advantage that other planners too can be used to solve the global planning problem, whereas *Rolling Stone* is a special purpose system.

Our first results confirm that our problem-solving architecture works. We have already obtained evidence that abstract Sokoban is more efficient than other planning representations of the game. To the best of our knowledge, no previous planning attempts in Sokoban led to solving real problems. However, as the comparison with *Rolling Stone* shows, parts of our architecture, such as local computation, are still in an early stage. A few of the ideas about how to improve our system's performance are presented in Section 6.

6 Conclusions and Future Work

In games such as Sokoban, approaches based on heuristic search seem to be of limited value. In this paper we have proposed an alternative problem-solving architecture based on AI planning. Since the classical representation of Sokoban as a planning domain did not lead to acceptable results, we have also introduced abstract Sokoban, mapping the puzzle to a highly abstracted planning domain. Our first experimental results support the claim that abstract Sokoban outperforms other planning representations of the game. Furthermore, since abstraction leads to a huge reduction of the global search space, we are encouraged to say that planning could be used to overcome the limitations exhibited by heuristic search in Sokoban.

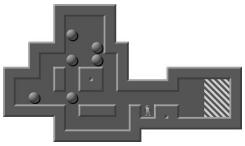
There are many directions that we plan to explore with abstract Sokoban. Our framework does not currently handle all types of large rooms. In addition, the rooms local computation can be further optimized. As a consequence, one of our main future work directions is to complete and improve the local processing for both goal and regular rooms. Another enhancement that we expect to have a great impact on the system performance is a smarter decomposition of the maze into rooms and tunnels. While our heuristic rule that guides this process is quite rigid, it can be replaced by a strategy aiming to optimize several parameters (e.g., minimize the number of rooms and tunnels, minimize the interactions between

rooms and tunnels). Moreover, the global planning search space can be further simplified, detecting large deadlocks that involve interactions between several rooms and tunnels. One important future research topic is to try our ideas in real-life planning domains (e.g., robotics related). Automatic abstraction of planning domains can also be an interesting extension of our work.

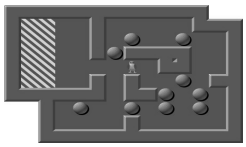
References

1. F. Bacchus. AIPS'00 Planning Competition. *AI Magazine*, pages 47–56, 2001.
2. F. Bacchus. Personal communication, March 2002.
3. F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 16:123–191, 2000.
4. J. Culberson. SOKOBAN is PSPACE-complete. Technical report, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1997. `ftp://ftp.cs.ualberta.ca/pub/TechReports/1997/TR97-02`.
5. R. Holte, M. Perez, R. Zimmer, and A. MacDonald. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. Technical report, University of Ottawa, TR-95-18, 1995.
6. A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search.* PhD thesis, 1999.
7. A. Junghanns and J. Schaeffer. Single-Agent Search in the Presence of Deadlock. In *Proceedings AAAI-98, Madison/WI, USA*, pages 419–424, July 1998.
8. A. Junghanns and J. Schaeffer. Domain-Dependent Single-Agent Search Enhancements. In *Proceedings IJCAI-99, Stockholm, Sweden*, pages 570–575, August 1999.
9. A. Junghanns and J. Schaeffer. Sokoban: Enhancing single-agent search using domain knowledge. *Artificial Intelligence*, 129(1–2):219–251, 2001.
10. R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.
11. D. McDermott. Using Regression-Match Graphs to Control Search in Planning. 1997. `http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/mcdermott.html`.
12. A. Shapiro. *Structured Induction in Expert Systems.* Turing Institute Press. Addison-Wesley, 1987.
13. D. E. Wilkins. Using knowledge to control tree searching. *Artificial Intelligence*, 18:1–51, 1982.

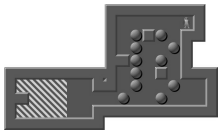
A The 10 Problem Test Suite



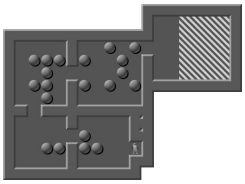
Problem #1



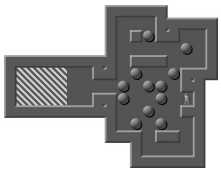
Problem #2



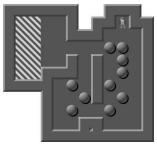
Problem #3



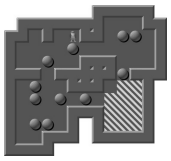
Problem #4



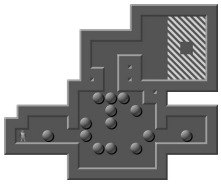
Problem #5



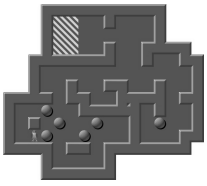
Problem #6



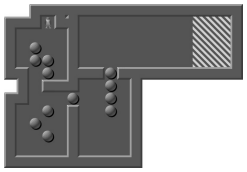
Problem #7



Problem #9



Problem #17



Problem #80