Hierarchical Planning and Learning for Automatic Solving of Sokoban Problems

Jean-Noël Demaret [a]* François Van Lishout [b] Pascal Gribomont [b]

[a] *Lab for User Cognition & Innovative Design, University of Liège, 4000 Liège, Belgium*
[b] *Montefiore Institute, University of Liège, 4000 Liège, Belgium*

Abstract

The most immediate method used to solve single-player games like Sokoban is to view them as state-space problems. However, classical search algorithms are insufficient to solve nontrivial problems and Sokoban remains a very challenging domain for the AI research community. The current state-of-the-art solver uses a heuristic-based approach coupled with a set of domain-specific enhancements. This paper introduces a new solving method which relies on two principles observable in human in-game behavior: (1) the decomposition of a complex problem into a sequence of simpler subproblems and (2) the non-repetition of encountered mistakes. We demonstrate the relevance of this approach and we show that it enables to reach performance comparable to the current state-of-the-art solver on a difficult 90-problem test suite.

1 Introduction

Artificial Intelligence researchers have always viewed games as privileged subjects of exploration to build computing machinery able to simulate human reasoning capabilities. In many games like Chess or Checkers, the machine has become stronger than the best human players. However, there are games where human performance remains significantly higher than those of the machine although their computing capabilities increase continuously. One of them is Sokoban.

1.1 Sokoban

Sokoban is a japanese single-player game where the player moves in a maze and has to push stones to specific locations, called *goals* (see Figure 1). The player can only push stones and he can only push one at a time. Solving a Sokoban problem consists in finding a sequence of moves that leads to a situation where all goals are filled, i.e., occupied by a stone. Most Sokoban problems admit several solutions. A solution can be expressed either in terms of player's moves or in terms of pushes. In this paper, we only consider the latter and an elementary game action will therefore not be a player's move but the push of a stone.

The most immediate method used to solve a Sokoban problem is to view it as a state-space problem. The structure of this state-space is a graph where each node is a game state whose successors are the game states reachable in one push. Solving a problem consists thus in finding a path in this graph between the node representing the initial game state and a node representing a solution game state, i.e., a game state where all goals are filled. However, classical search algorithms are insufficient to solve nontrivial problems. Sokoban has, in fact, several features that make it a particularly complex problem [4, 5, 9]. The branching factor of a problem, i.e., the number of possible pushes in a given game state, is very high (possibly more than 100) and a solution can be very long (more than 600 pushes). Thereby, the size of the state-space is enormous; it has been estimated at 10^{98} for a 20×20 maze. By comparison, in the case of Chess, the average branching factor is 35, the length of a game is about 50 moves and the size of the state-space is valued at about 10^{40}. Also, contrary to other games, a bad move can lead in Sokoban to a *deadlock*, a situation in which the solution

*F.R.S.-FNRS Research Fellow (Belgium)

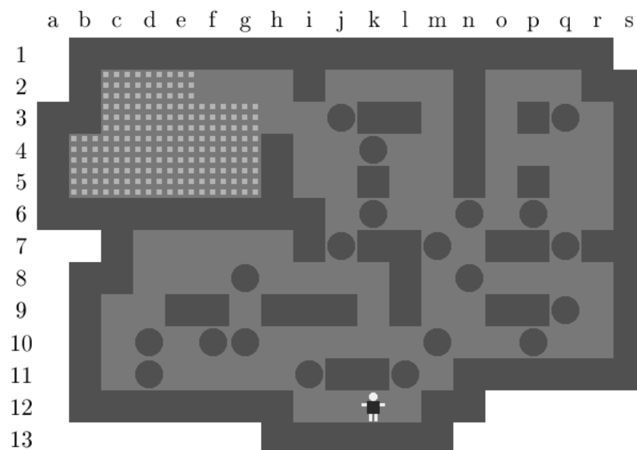

Figure 1: The problem #27 of the benchmark.

game state is not reachable anymore. Figure 3 shows such situation: there is no way to bring the three stones on *h8*, *h9* and *g10* to the goal aera. Finally, Sokoban's wide variety of problems makes it extremely difficult to find common strategies to solve them.

1.2 State of the Art

The program *Rolling Stone* [5] from the University of Alberta is currently the best documented Sokoban solver. It uses the heuristic search IDA*-algorithm [6] coupled with a set of domain-specific enhancements that enable it to solve 59 problems of a difficult 90-problem test suite[1]. Despite this significant result, it seems that the performance of this program, and therefore of this heuristic-based approach, has reached its limits. On the one hand, as explained in [5], a heuristic function is difficult to design and expensive to calculate ($O(n^3)$ for a problem of n stones). On the other hand, [5] also demonstrates that an informed search algorithm cannot solve any problem of the benchmark without the use of domain-specific enhancements. Finally, *Rolling Stone* no longer seems to be under research and recent publications on Sokoban from the University of Alberta explore different methods of resolution [2].

A completely different approach was adopted by *Talking Stones* [7], a program developed a the University of Liège. It relies on a multi-agent representation of a Sokoban problem which is to view the primitive elements of the game (the stones in the case of Sokoban) as agents which collaborate with each other to achieve a common objective. The program implements an algorithm capable of solving immediately the whole subclass of Sokoban problems satisfying the following three conditions:

- It must be possible to determine in advance the order in which goals will be filled.
- It must be possible to bring at least one stone in the first goal to be filled, without having to modify the position of another stone.
- For each stone satisfying the previous condition, the problem obtained by removing that stone and replacing the first goal by a wall must also belong to the subclass.

These conditions are very restrictive and most difficult Sokoban problems don't satisfy them (only one problem of the 90-problem benchmark). For this reason, *Talking Stones* uses a classical search algorithm to explore the state-space of the problem in order to find a game state belonging to the subclass. Note that it is trivially the case for the solution game state and that this solving protocol is therefore theoretically complete.The first version of the program is able to solve 9 problems of the benchmark without the use of any other enhancement. We will see that the method presented in this paper will allow us to eliminate the two last conditions and to achieve significantly better performance.

At present, no solver is able to solve all the problems of the benchmark and Sokoban still remains a very interesting domain for researchers working in the field of search and planning problems (e.g., [1]).

[1]This test suite is available at http://www.cs.cornell.edu/andru/xsokoban.html

2 Divide to Plan

The systematic exploration of all possibilities is an approach that suits the workings of a machine but does not correspond to the approach adopted by a human player facing a Sokoban problem. We can indeed notice that the human player builds his in-game behavior around the definition of high-level strategic objectives. An important point is that the player defines a strategic objective *before* searching a sequence of moves that would allow him to achieve it. This observation led us to address the automatic solving of Sokoban problems from a new angle, the hierarchical planning. The principle of this approach is to divide a complex problem into a sequence of subproblems of lesser importance in order to facilitate its resolution [8].

In the field of search problems, the use of hierarchical planning introduces an additional level of search involving actions of higher level than the elementary actions (e.g., moves) used by classical search algorithms to explore the state-space of a problem. In this approach, the problem is first solved using these high-level actions, then the realization of each high level action is transformed into a sequence of elementary actions to obtain the desired solution to the problem.

We have already applied this principle of hierarchical decomposition choosing to use the push of a stone as elementary game action. A push corresponds indeed to a sequence of player's moves and can therefore be regarded as a higher-level action.

2.1 Decomposition of a Sokoban Problem

How to divide a Sokoban problem into a sequence of subproblems? The decomposition that we have chosen is based on the following observation: the solution to a Sokoban problem can be divided into a series of sequences of pushes to the end of which a stone is brought into its final location, i.e., the goal it will occupy in the solution game state. Each of these sequences may itself be decomposed into two phases: (1) an *extrication phase*, during which a number of pushes are conducted on different stones, and (2) a *storage phase*, during which a succession of pushes relating to a single stone brings it into its final location. So, the planning phase of our solving method is to determine the order in which the goals will be filled. We call the produced ordered list of goals the *goal scheduling*. Each subproblem will therefore consist of finding a way to fill a given goal, i.e., a sequence of pushes such as we have described above.

It is interesting to notice that any scheduling will be a valid decomposition of a solution. Indeed the defined decomposition does not exclude that goals are filled during the extrication phase. Given a set of n goals to fill, in the worst case scenario, i.e., if the first goal of the scheduling is the last goal that has to be filled, the extrication phase of the first sequence will fill the $(n-1)$ first goals and the storage phase will bring the last stone into the last goal. In this scenario, the resolution of the first subproblem has led to solve the entire problem and the sequences of pushes corresponding to the resolution of the $(n-1)$ remaining subproblems will therefore be empty. However, such a situation loses the entire relevance of the method and is only useful to demonstrate its theoretical completeness.

Indeed, a scheduling will be meaningful only if it divides the solution in n non-empty sequences, i.e., if it really corresponds to the order in which the goals will effectively be filled. Actually, the main benefit of the method will be to significantly reduce the depth of the search tree associated to the resolution of each subproblem, replacing the storage phase of a sequence by a single game action that we will call a *macro-push*. For example, in Figure 3 (left), a macro-push could be used to bring the stone on *k9* to the goal *p4* in a single game action.

Our approach will thus be adapted to the subclass of Sokoban problems where it is possible to determine in advance and without ambiguity the order in which the goals will be filled. As we will see, this is mainly the case for problems having only one entrance and one goal area. An *entrance* is a location from which a stone can enter into a goal area. A *goal area* is a set of goals occupying contiguous locations.

2.2 Goal Scheduling

A goal scheduling is a permutation of the list of the n goals of a problem. The determination of a scheduling is to select a particular permutation among the $n!$ possible permutations of this list. The scheduling that we search is a scheduling which corresponds to the order in which the goals of the problem can effectively be filled. We call such a scheduling an *effective scheduling*. Formally, a scheduling will be effective for a given problem if at least one solution in which goals are filled in the order defined by this scheduling exists. This definition of effectiveness is however hardly exploitable. Indeed, proving that a scheduling is effective means finding a solution to the problem and therefore makes goal scheduling meaningless. It is thus essential

to identify a less restrictive condition able to generate a “good scheduling” from an operationally reasonable analysis of the initial game state of a problem.

A necessary condition for a scheduling to be effective is that the scheduling would not create any situation of deadlock, i.e., it does not lead to a situation in which the stones occupying filled goals make some unfilled goals unreachable. For example, in Figure 2, the goal *k7* cannot be filled before the goal *i7* without create a deadlock. We call *consistent scheduling* a scheduling which guarantees that no goal of the problem will ever be made unreachable by the stones occupying the filled goals. The latter is important because the consistency of a scheduling does not exclude that a goal is made unreachable by the other stones of the problem (see Figure 2).

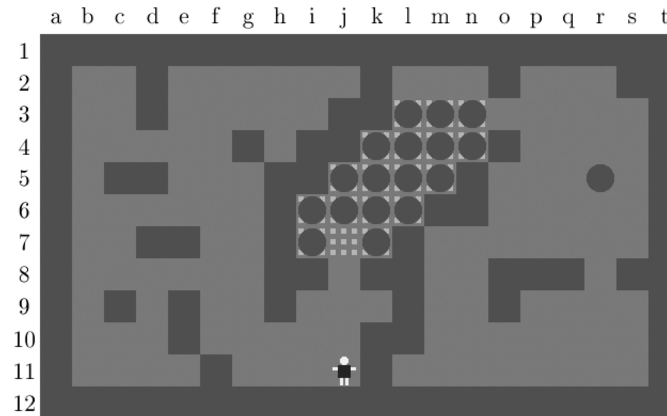

Figure 2: The goal *j7* could be theoretically filled through the entrance *j8* but the location of the stone *r4*, that cannot be bring to this entry, makes it effectively unreachable.

Therefore, we cannot exclude that a scheduling, though consistent, is such that all the sequences of pushes filling the goals in the order defined by the scheduling lead to a deadlock. A consistent scheduling is thus not always an effective one. Let $\mathcal{O}$ be the set of all $n!$ possible schedulings in a problem of n goals, $\mathcal{O}_{consistent}$, the set of all consistent schedulings and $\mathcal{O}_{effective}$, the set of all effective schedulings. We can establish the relationship

$$\mathcal{O}_{effective} \subseteq \mathcal{O}_{consistent} \subseteq \mathcal{O}$$

The proof that a scheduling is consistent is relatively easy. It consists of filling the goals of the problem one by one in the order defined by the scheduling and check at every step if all unfilled goals are reachable. So, the consistency will be the criterion we will use to generate a “good scheduling”. This solution is not ideal since it does not guarantee that the produced scheduling is always effective. However, the condition of consistency is sufficient for a subclass of Sokoban problems defined by an easily identifiable feature.

Indeed, in a problem having only one entrance (and therefore, one goal area), the accessibility of a goal only depends on the occupation of the goal area and not on the possibility to bring a stone to a given entry. We have therefore for this situation the relationship

$$\mathcal{O}_{effective} = \mathcal{O}_{consistent}$$

In this case, the scheduling of goals can be determined without ambiguity since the generated consistent scheduling will always be effective and not arbitrarily chosen in the set of all consistent schedulings.

Our goal scheduling algorithm starts from a game state in which the goal area is fully occupied by stones. It consists then in emptying it by gradually evacuating the stones one by one either through the single entry or through an arbitrarily chosen one. The scheduling of the goals is then obtained by reversing the order in which the goals have been emptied.

3 Learn from Deadlocks

An important feature of an intelligent agent is its ability to learn. In its simplest form, it is reflected by its capability to not commit the same mistake twice. In the case of solving a Sokoban problem, making a mistake is to create a deadlock. In most cases, a deadlock situation is not induced by the position of all the stones of the problem but by a subset of them. Accordingly, all game states that contain this subset, which we call a *local deadlock*, are also deadlock situations (see Figure 3). There may be a potentially very high number of such game states. This number is usually inversely proportional to the number of stones in the subset. Identifying and remembering this subset of stones makes it possible to exclude from the search tree a significant number of game states and thus reduce significantly its branching factor.

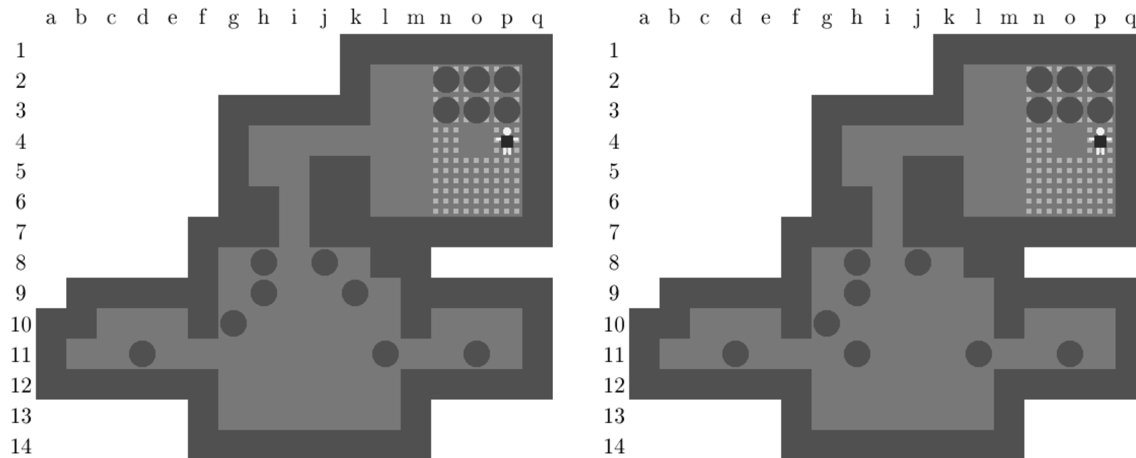

Figure 3: Two game states that contain the same local deadlock (*h8*,*h9*,*g10*).

It is important to notice that some deadlock situations also depend on the player's position. For example, the game states showed in Figure 3 would not be deadlock stiuations if the player was on *g8*. That information will therefore be associated with each stored subset of stones so as not to exclude game states from the search tree that would be wrongfully identified as deadlock situations.

In order to avoid using too many resources, the identification of local deadlocks is limited to the analysis of game states from which a subproblem has not been resolved. It is, indeed, reasonable to assume that such a game state will contain most of the time at least a subset of stones involving a deadlock.

Also, the number of existing subsets of stones increases exponentially with the number of stones present in the analyzed game state. It is thus not feasible to examine each subset of stones thoroughly. It is therefore necessary to choose a limited number of relevant subsets of stones. In our experiments, we have noticed that many local deadlocks were composed of groups of contiguous stones. We have therefore chosen to use these groups of contiguous stones, which we call clusters, as constituents for examined subsets. The examined subsets will be the subsets consisting of one, two or three clusters.

The method used to determine if a given subset of stones implies a deadlock uses a classical search algorithm. It searches a way to fill the target goal of the subproblem for which no solution has been found, from a temporary game state in which stones outside the subset were removed. If it is not possible to bring a stone towards the target goal, the subset is identified as a local deadlock and is added to the database.

4 Solving Protocol

Conceptually, our solving protocol is organized around four search functions which are each responsible for solving a particular problem :

- The first function is in charge of determining the goal scheduling by analyzing the initial game state.
- The task of the second function is to fill one after the other the goals of the problem in the order defined by the scheduling.

- The third function is in charge of solving the subproblem consisting of filling the current target goal, i.e., the first unfilled goal of the scheduling, from a given game state.
- The role of the fourth function is to search for possible local deadlocks in a given game state.

Starting from these four search functions, our solving method can be described as follows. To begin with, the first function analyzes the initial game state and determines the order in which the goals are to be filled. The resulting goal scheduling and the initial game state are then passed as arguments to the second function. The latter then searches a way of filling the first goal of the scheduling. To do so, it gives control to the third function and passes it the initial game state and the position of the current target goal as arguments. The third function launches a search which aims at finding a sequence of moves leading to a game state in which the target goal is filled. This research is implemented as a classical search algorithm which used a transposition table in order to avoid visiting the same game state twice and verifies at each step the possibility of performing a macro-push to the target goal. The solution node found is returned to the second function which uses the game state content in this node as a starting point from which a way of filling the second goal of the scheduling will be found. If, at any step of the solving process, a way of filling the nth goal cannot be found, a search for local deadlocks is made (fourth function) and another way of filling the $(n-1)$th goal is then searched.

Operationally, the second function can be seen as doing a depth-first walk in a tree whose root is the initial game state of the problem and where a node located at a depth n is a game state in which the $(n-1)$th first goals of the scheduling are filled. In this tree, a node at depth n has, as successors, the game states reachable in any number of moves and where the first n goals of the scheduling are filled.

4.1 Results

In order to demonstrate the relevance of our approach with experimental results, we have implemented the introduced solving protocol as a Scheme program. A detailed description of this implementation is given in [3]. We have then evaluated its performance on the classical 90-problem test suite used by the previously mentioned start-of-the-art programs. Our program was able to solve 54 problems of this difficult benchmark.

During our experimentations, we realized that some problems for which no consistent goal scheduling could be found, became accessible and sometimes easily solvable by our program if the disposition of the goals was previously rearranged. In fact, to solve those problems, it is necessary to bring stones to temporary positions before bringing them to their final location in order to avoid making some of the goals unreachable. Figure 4 shows an example of such rearrangement.

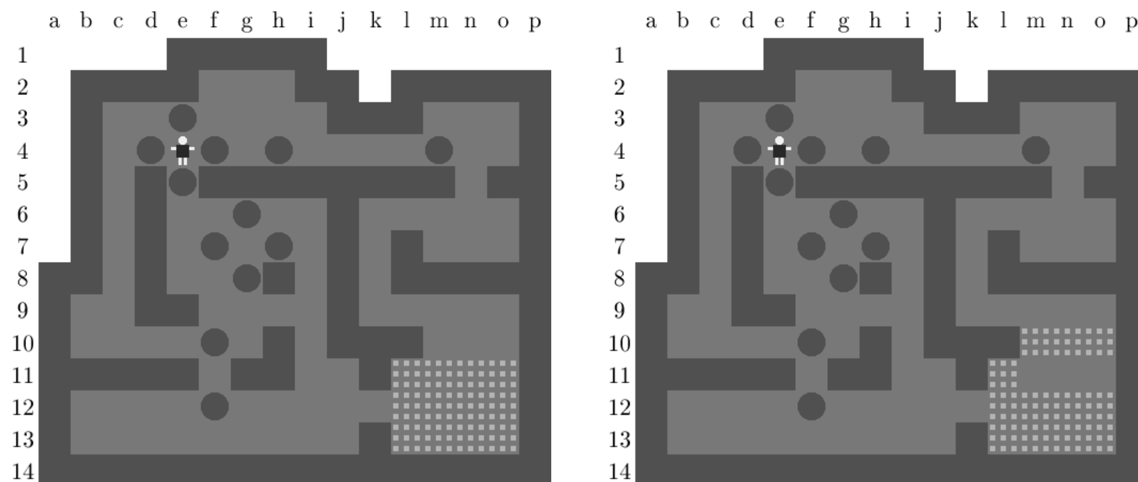

Figure 4: An example of goal rearrangement (right) for the problem #87 of the benchmark (left).

It is easy to be convinced that obtaining a solution to the initial problem from the one of the rearranged problem is almost immediate. Therefore, the design of a procedure to automate this rearrangement would allow us to add 7 problems to the list of problems solved by our program. It would therefore be able to solve

61 of the 90 problems of the benchmark and thus to compete with the performance of the state-of-the-art solver *Rolling Stone* (59 problems solved).

Table 1 shows results obtained on the 61 solved problems. Total nodes represents the total number of nodes that were explored during the resolution of subproblems and the search for local deadlocks. The next column shows the total nodes obtained by *Rolling Stone* [5]. At last, Time is the time consumed for solving the problem on a standard desktop machine[2].

Problem	Total nodes	Rolling Stone total nodes	Time	Problem	Total nodes	Rolling Stone total nodes	Time
1	97	1,267	1 s	51	121	29,569	6 s
2	291	7,530	3 s	53	15	22,308	8 s
3	32	14,095	1 s	54	162	66,306	19 s
4	6,858	50,369	1 min 26 s	55	15	2,993	6 s
5	133	35,974	4 s	56	35	50,924	7 s
6	193	5,503	2 s	57	65	128,282	21 s
7	427	15,790	4 s	58	306	138,838	10 s
8	77	409,714	14 s	59	272	409,470	13 s
9	2,193	407,103	27 s	60	182	31,413	8 s
10	710	19,967,875	3 min 51 s	61	310	77,555	21 s
11	83,711	2,331,950	27 min 25 s	62	1,649	69,728	56 s
12	31	372,264	10 s	63	40	578,066	10 s
13	17,918	unsolved	11 min 2 s	64	154	186,508	8 s
17	1,427	33,901	13 s	65	122	23,004	10 s
19	454	6,089,182	20 s	67	580	104,356	25 s
21	2,402	258,852	36 s	68	395	236,157	13 s
24	403,229	unsolved	3 h 2 min	70	2,161	178,657	1 min 34 s
25	516,000	592,585	6 h 11 min	72	86	45,735	9 s
26	1,171	126,379	19 s	73	51	103,494	9 s
27	10,183	unsolved	1 min 46 s	75	2,133,007	5,095,054	6 h 23 min
34	503	442,025	12 s	76	90,592	1,980,094	27 min 27 s
35*	168	unsolved	21 s	78	8	4,913	2 s
36	147,434	5,785,290	1 h 5 min	79	204	13,114	11 s
37	1,078	unsolved	1 min 5 s	80	344	26,309	10 s
38	14,749	56,563	2 min 9 s	81	2,604	206,423	40 s
39*	1,646,578	unsolved	5 h 50 min	82	66	45,014	4 s
43	1166	523,907	16 s	83	1040	6,856	19 s
44*	736,375	unsolved	2 h 14 min	84	695	7,818	52 s
45	942,964	410,134	3 h 13 min	86*	143	unsolved	2 s
46*	53,584	unsolved	14 min 31 s	87*	80	unsolved	4 s
47*	264	unsolved	5 s				

Table 1: Results obtained on the 61 solved problems. Rearranged problems are marked with an asterisk (*).

4.2 Comparison with Other Methods

This work is the continuation of research conducted at the University of Liège in 2005 around the solver *Talking Stones*. Note that the subclass of problems that could be solved efficiently by our method includes all the problems solved immediately by *Talking Stones*. Indeed, these problems should satisfy three conditions, the first one being that the order in which the goals will be filled can be determined in advance. From this point of view, our method can be regarded as a generalization of the solving protocol used by *Talking Stones* since the necessity to satisfy the two other very restrictive conditions has been eliminated.

Similarly, we can establish links between the ideas behind our approach and some of the domain-specific enhancements used by *Rolling Stone*. First, the use of a macro-push to replace the storage phase in the resolution of a subproblem is similar to *Goal Macros* (see [5], enhancement #5), which consists in bringing a stone entering in a goal area immediately into an appropriate final location. Secondly, *Pattern Search* (see [5], enhancement #7) is used in *Rolling Stone* to analyze some game states deemed critical (by a specific heuristic) to identify potential conflicts existing within subsets of stones. Those subsets are stored in memory and then used to adjust the estimation produced by the heuristic function. The idea behind this improvement is therefore similar to the one consisting of searching for local deadlocks. Finally, like *Rolling Stone*, we use a transposition table (see [5], enhancement #1) to avoid visiting the same node of the search tree twice during the resolution of a subproblem.

In [5], the authors show that the three enhancements described above are the most important in term of performance for *Rolling Stone*. This assertion is clearly confirmed by our approach which is able to reach comparable results with a solving protocol which is mainly based on ideas similar to those that support them. Furthermore, [5] also demonstrates that the use of a heuristic function is insufficient but necessary to

[2] 1.8GHz CPU and 512 MB RAM

solve any problem of the benchmark. Our results suggest instead that the relevance of a heuristic function in comparison with the domain-specific enhancements must be relativized in the case of Sokoban.

5 Conclusion and Future Work

This paper focuses on automatic solving of Sokoban problems. From the observation of human in-game behavior, we have developed a new solving method based on planning by decomposition into subproblems. We have described such decomposition and we have demonstrated its relevance for the subclass of Sokoban problems where it is possible to determine in advance the order in which the goals will be filled. We have then incorporated in our method a fundamental learning concept. By analyzing encountered deadlocks and by storing collected information in a database, our program was able to eliminate many game states from the search tree. The solver implemented from the resulting solving protocol was thus able to solve 54 problems of a difficult 90-problem test suite. At last, we have shown that a preliminary goal rearrangement allows our program to solve 61 problems of the benchmark and therefore to reach performance comparable to the currently best documented solver (59 problems solved).

The major limitation of the introduced method resides in its inability to efficiently handle the existence of several entrances and consequently the existence of several goal areas in a problem. In this type of problem, a preliminary analysis of the initial game situation is not sufficient to determine an effective goal scheduling. To overcome this limitation, we consider exploring the possibility of not determining in advance a full goal scheduling but recomputing a list of possible target goals at each step of the solving process.

In future work, we plan also to develop an automatic goal rearrangement procedure. This procedure will take the form of an additional search function which will be called when no goal scheduling can be found. A valid goal rearrangement must have two properties: (1) it must be possible to find a goal scheduling for the rearranged problem and (2) it must be possible to reach a game situation in which the initial problem is solved from the solved rearranged problem.

Finally, we plan to inject in our program the heuristic function and the domain-specific enhancements introduced in [5] to improve the search algorithm used in subproblem resolution. We hope that the combination of a high-level planning and learning approach with these lower-level search optimizations will lead to significant advancement in the challenging domain of automatic solving of Sokoban problems.

References

[1] M. S. Berger and J. H. Lawton. Multi-agent planning in Sokoban. *Lecture Notes in Computer Science*, 4696:334–336, 2007.

[2] A. Botea, M. Muller, and J. Schaeffer. Using abstraction for planning in Sokoban. *Lecture Notes in Computer Science*, 2883:360–375, 2003.

[3] J.-N. Demaret. *L'intelligence artificielle et les jeux : cas du Sokoban*. Master thesis, University of Liège, 2007.

[4] A. Junghanns. *Pushing the limits: New developments in single-agent search*. PhD thesis, University of Alberta, 1999.

[5] A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1):219–251, 2001.

[6] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 26(1):35–77, 1985.

[7] F. Van Lishout and P. Gribomont. Single-player games: Introduction to a new solving method combining state-space modelling with a multi-agent representation. In *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'2006)*, pages 331–337, 2006.

[8] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 1995.

[9] W. Wesselink and H. Zantema. Shortest solutions for Sokoban. In *Proceedings of the 15th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'2003)*, pages 323–330, 2003.