Finding Optimal Solutions to Sokoban Using Instance Dependent Pattern Databases

André Grahl Pereira and **Marcus Rolf Peter Ritt** and **Luciana Salete Buriol**
Institute of Informatics
Federal University of Rio Grande do Sul
Brazil
andre.grahl.ai@gmail.com, mrpritt@inf.ufrgs.br, buriol@inf.ufrgs.br

Abstract

Pattern databases have been successfully applied to several problems. Their use assumes that the goal state is known, and once the pattern database is built, commonly it can be used by all instances. However, in Sokoban, before solving the puzzle, the goal position of each stone is unknown. Moreover, each Sokoban instance has its own state space search. In this paper we apply pattern databases to Sokoban. The proposed approach uses an instance decomposition, that allows multiple possible goal states to be abstracted into a single state. Thus, an instance dependent pattern database is employed. Experiments with the standard set of instances show that the proposed approach overcomes the current best lower bounds in initial states for several instances. Furthermore, three of these new best lower bounds match exactly with the optimal solution length. Finally, we run experiments of 5 million explored states for each instance. Nine instances were solved with optimality guarantees, while only four instances were solved under the same conditions by previous methods.

Introduction

Sokoban is a one-player puzzle and can be seen as an abstraction of a robot motion planning problem. The puzzle can be defined as a grid-square configuration with a set of k stones (movable squares) and k goal squares. One of the squares is distinguished as the warehouse keeper (Sokoban), or simply the "man". A move consists of pushing a stone from one square to a vertically or horizontally adjacent square, and it can only be executed by the man. A move is valid if the adjacent target square is empty. Some squares are walls and unmovable. The goal is to execute a minimal sequence of pushes to move the stones from their initial positions to the goal squares. Figure 1 illustrates a sample instance.

Even after three decades since Sokoban was created, solving it algorithmically remains a challenge. Much research has been done on this puzzle and in general the attempts were in finding any solution. However, solving a puzzle with guarantee of optimality is a goal of current research (Korf 2012). From the standard set composed by 90 instances, described in Section "Results", only a few were solved with techniques that guarantee optimality, and about more than half of the instances were solved using techniques that do not guarantee optimality.

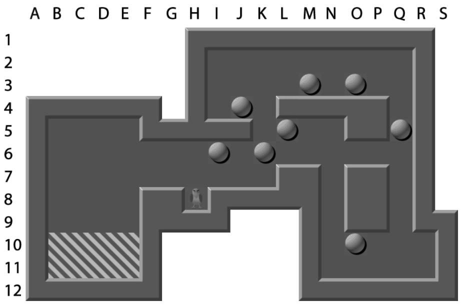

Figure 1: Sokoban instance #78 of the standard set. In position *H8* is the man. In position *J4* there is a stone. Position *B10* represents a goal square. Finally *B9* exemplifies an empty square, while in *A4* there is a wall.

Sokoban was shown to be PSPACE-Complete (Culberson 1998). Table 1 compares some principal characteristics of Sokoban to that of other single-agent search problems. Due to its large branching factor, solution length, and search space size, Sokoban is considered one of the most challenging single-agent search problems.

Characteristic	24-Puzzle	Rubik's Cube	Sokoban
Branching Factor	2.37	13.35	12
- range	1-4	12-15	0-126
Solution Length	100	16	260
- range	80-112	14-18	97-674
Search Space Size	10^{25}	10^{19}	10^{98}

Table 1: Search space properties of some single-agent search problems (Junghanns and Schaeffer 2001; Edelkamp and Schrödl 2012).

The most successful attempts to solve Sokoban are using heuristic search methods. Heuristic search finds solutions to state space search problems. The state space can be defined as a directed graph, whose nodes are the states, and whose arcs are valid transitions between them. Heuristic search aims at finding a path, or a shortest path, from the initial state to some goal state in this graph.

Algorithms like A^* (Hart, Nilsson, and Raphael 1968) and *IDA** (Korf 1985) visit the states guided by a cost function $f(s) = g(s) + h(s)$, where s is the current state, $g(s)$ is the distance from the initial state to the state s, and $h(s)$ is a function that estimates the distance from the current state s to a goal state. Commonly, $h(s)$ is admissible, i.e., it never overestimates the real distance to a goal state, providing a lower bound. This ensures that an optimal solution will be found.

In recent works, pattern databases were successfully applied in single-agent search problems as the Rubik's Cube (Korf 1997), the Sliding-Tile Puzzle (Korf and Felner 2002) and the Tower of Hanoi (Felner, Korf, and Hanan 2004). Pattern databases store the cost of solving subproblems. The goal is to improve the lower bound such that less search states are explored, saving computational effort. Pattern databases are constructed based on a pattern space. A pattern space is an abstraction of the real state space. In this way only part of the state is considered, the rest of the state is marked as "don't care". In the Sliding-Tile Puzzle, for example, this can be done by choosing a subset of tiles as a pattern. Similarly, in Sokoban, this can be achieved by choosing a subset of stones. Generating the pattern database has a high cost. However, this cost is amortized by solving several instances, or by using less states for solving a single instance. In general the pattern database is built in a preprocessing phase by a breadth-first backwards search from the goal pattern until the whole abstract pattern space is reached.

In this paper, we present an approach for using instance dependent pattern databases in Sokoban, resulting in improvements over the state of the art lower bounds proposed for this problem (Junghanns 1999; Junghanns and Schaeffer 2001). Besides improving the lower bounds, this technique allows to solve more instances with guarantee of optimality. The remainder of the paper is organized as follows. In Section "Literature review" we review the literature on Sokoban and pattern databases, while in Section "Proposed Approach" we describe the proposed lower bound and the use of pattern databases in Sokoban. In Section "Experimental Results" we discuss the proposed approach and present experimental results. Finally, in Section "Conclusions and Future Work" we present conclusions and discuss future work.

Literature review

Sokoban

Some early works on Sokoban are reviewed in (Junghanns and Schaeffer 1999). The Rolling Stone solver proposed in the Ph.D. thesis of Junghanns (Junghanns and Schaeffer 1999; Junghanns 1999; Junghanns and Schaeffer 2001) is a milestone of the research effort related to Sokoban. The solver is based on the Iterative Deepening A^* (*IDA**) algorithm using multiple domain independent and dependent enhancements. The goal of Rolling Stone is to find a solution for Sokoban, not necessarily providing an optimal solution. Their solver initially considers an enhanced lower bound, transposition tables, move ordering, deadlocks tables, and tunnel macros. All these techniques are used such that an optimal solution is provided. However, with these techniques, and using as limit 20 million states of the search space, only instances #1, #2, #6, #17, #38 and #78 are solved. Thereby, in an attempt to solve more instances, the authors relaxed the optimality criterion and applied techniques that help finding feasible solutions more often, without guaranteeing optimality. Non optimal techniques such as goal cuts, goal macros, pattern searches, relevance cuts, overestimation and rapid random restart are used. Thus, the Rolling Stone solver was able to solve 57 instances, however, without optimality guarantees. These techniques bring a lot of efficiency to the solver. Disabling pattern searches Rolling Stone was able to solve instance #1 using only 52 nodes. This is impressive considering that the instance has a minimum solution length of the 97, but this is done with no guarantees of optimality. In this paper we demonstrate that it is possible to increase the number of solved instances with guarantees of optimality using fewer nodes, only by applying a more accurate lower bound.

In (Botea, Müller, and Schaeffer 2002), the proposed solver uses abstractions for planning in Sokoban, being able to solve non-optimally ten instances from the standard set.

In (Demaret, Lishout, and Gribomont 2008) the authors describe a solver that uses a hierarchical planning strategy along with deadlocks learning to solve instances of Sokoban. Using this approach, they were able solve non-optimally 54 instances of the standard set. In this case, the stopping criterion used was eight hours of running times per instance.

There are also other solvers that are able to solve a larger set of instances, but also without optimality guarantee. Among the ones with best results JSoko (Meger 2013) is able to solve 65 instances; the solver YASS (Damgaard 2013) is able to solve 75 instances; and the solver Takaken (Takahashi 2013) is able to solve 86 instances. All these solvers employ many non optimal techniques, like a more general version of the goal macros known as packing order calculation. Only for the JSoko solver the solution length is provided. For example, for instance #3 the solver found a solution using 148 pushes, while the optimal solution uses only 131.

In summary, among the current solvers, Rolling Stone using only the techniques that guarantee optimality is still the state of the art for solving Sokoban exactly. For this reason, in the experimental results section of this work we compare our results with the optimal guarantee results of the Rolling Stone.

Pattern Databases

Additive dynamic partitioned pattern databases (Felner, Korf, and Hanan 2004) arise in contrast to additive static partitioned pattern databases. In the latter, a partition of a disjoint subproblem is defined and used in each state during the search. In the former, for every state in the search, the problem is partitioned into disjoint subproblems. Since all possible partitions can be considered, they can capture interactions that are not captured in the static partition, obtaining better results.

For the Sliding-Tile Puzzle the dynamic partitioned PDB of pairs has to compute the PDB for every pair of tiles. This is not the case in Sokoban, since every stone is indistinguishable, and so we can compute a single PDB for stones and every pair of stones can use the same PDB. Therefore, in this context to explore dynamic partitioned PDBs has low cost.

When the pattern database is composed by pairs, the highest heuristic value can be calculated by a maximum weight matching. For patterns composed by triples, or larger, the matching problem becomes NP-Complete (Felner, Korf, and Hanan 2004) and other approaches are used to compute the heuristic value.

Pattern databases can also differ from each other in how they are used. The use of pattern databases assumes that the goal state is known, and once the pattern database is built, commonly it can be used by all instances. However, in some applications, an instance dependent pattern database can be used, i.e., a pattern database is built for each instance.

The use of pattern databases was proposed in (Culberson and Schaeffer 1998) and originally applied to the Fifteen Puzzle. Since that, pattern databases have been improved and applied to different domains (Korf 1997; Korf and Felner 2002; Felner, Korf, and Hanan 2004). In (Zhou and Hansen 2004) a method was proposed for reducing the memory required during the construction of an instance dependent pattern database considering specific start and goal states. This method was improved in (Felner and Adler 2005) enabling the computation of larger pattern databases, and was applied to the Twenty-Four Puzzle, achieving a speedup over previous approaches. Inspired by these works, in Subsection ”Lower Bound” we discuss the idea of an instance dependent pattern database, and the related issues to its application on Sokoban.

From an extensive literature review, the only works we found that had applied pattern databases to Sokoban are domain independent approaches. As a result, they did not generate significant results for the standard set of instances, only solving other sets of instances with a smaller state space search (Edelkamp 2001; Haslum et al. 2007). It could be argued that the deadlock tables or the pattern searches from Rolling Stone or the solution for the local problem from (Botea, Müller, and Schaeffer 2002) are the first application of PDB to Sokoban. But these approaches resemble more with pre-processing techniques than with PDB, since they do not have specific characteristics of PDB, like a target pattern and the distance for a set of pattern to the target pattern computed by retrograde analysis (Culberson and Schaeffer 1998).

Proposed Approach

The standard heuristic function $h(s)$ used by the search method when solving Sokoban is calculated as the minimum cost perfect matching in a bipartite graph (Junghanns and Schaeffer 2001) we call this procedure a *minmatching* in this work, while with *maxmatching* we refer to the problem of finding the maximum weighted matching in a graph). The graph partitions separate the goal set from the stone set. Since the goal of each stone is unknown, the minmatching is used for deciding the goal destination of each stone with

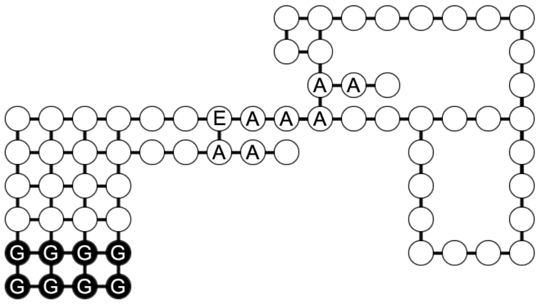

Figure 2: Graph generated from the instance in Figure 1. Nodes G are the goals, nodes labeled A and E are the articulation points, and E is the entrance.

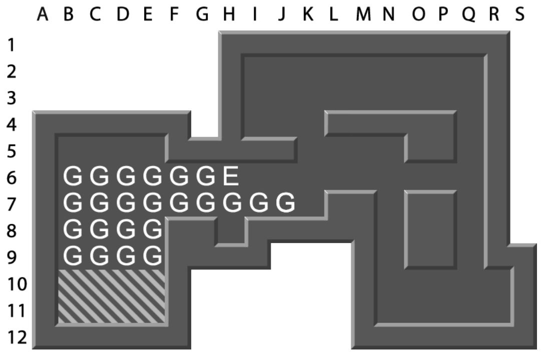

Figure 3: Decomposition of the instance presented in Figure 1, accordingly to the graph in Figure 2.

a minimum cost, thus providing a lower bound. In our approach, the lower bound is calculated as follows. Initially, the instance is decomposed into two zones. Pattern databases are used to calculate the lower bound of one zone, while minmatching is used to calculate the lower bound of the second zone. Next we detail this procedure.

Lower Bound

For decomposing the instance into two zones, we propose a method that creates an abstract goal state. This allows multiple possible goal states to be abstracted into a single goal state, and thus an instance dependent pattern database is applied in one of the zones. We call them the *maze zone* and the *goal zone*. The goal zone is the smallest set of squares that contains all goal squares, and to which stones can enter only by passing over a determined square, called the *entrance*. The smallest goal zone is chosen for maximizing the use of the pattern database. In our approach we use a pattern database that stores the distance to move stones to the entrance. This is the first part of the proposed lower bound. The second part is calculated using a minmatching as if all the stones outside that zone were positioned at the entrance.

For calculating the entrance, and then the maze and goal zones, we model an instance as an undirected graph $G = (V, E)$, where V is the set of nodes, and E the set of edges. The set of nodes are all non dead squares. A dead square is any square that, if it would hold a stone, this stone could not reach any goal. The set of edges is formed as follows. Every two adjacent goal nodes are connected. In addition, for every vertically or horizontally adjacent nodes u and v, the edge (u, v) belongs to E if i) u or v is connected to the goal nodes, and if ii) u can reach v by a movement of stone. For example, Figure 2 illustrates the graph constructed considering the Sokoban instance from Figure 1.

Once the graph is constructed, the entrance is then calculated. For that, all articulation points of the graph are detected. An articulation point is any node that if removed, disconnects the graph. Note that the graph may not have any articulation point. In this case, our lower bound reduces to the minmatching lower bound. On the other hand, a graph can have several articulation points, what happens in 88 from the 90 instances from the standard set. The entrance is the articulation point that minimizes the size of the goal zone.

Figure 3 shows an example of such a decomposition. The square marked with E is the entrance. The G squares plus the goal set are defined as the goal zone. All squares, including E, that are not in the goal zone, form the maze zone.

Having the entrance, maze and goal zones identified, the lower bound can then be calculated. The lower bound is calculated as the sum of two parts: the maze lower bound and the goal lower bound. The maze lower bound is calculated using pattern databases, which are described in the next section. The goal lower bound is calculated by a minmatching. For solving the minmatching, the stones in the maze zone are treated as if they were positioned at the entrance, while stones in the goal zone are positioned where they currently are.

The idea of decomposing instances of Sokoban has been explored in different works. Rolling Stone proposed the goal area, that is used to solved two problems independently, being computed using a highly pruned branch-and-bound. In (Botea, Müller, and Schaeffer 2002) the instance is decomposed into rooms and tunnels, and thus each subproblem is efficiently solved independently. The main difference of these approaches with ours is how the decomposition is made and how it is used. We decompose the instance choosing the smallest goal zone and we not use the decomposition to solve problems independently, we only use it to make the application of PDB efficient.

An Instance Dependent Pattern Database for Sokoban

Having the entrance identified, the construction of a pattern database is done exactly like in other domains. But some particularities can be explored to achieve better results. For example, in the Sliding-Tile Puzzle domain when the pattern separates the state into disjoint parts, one pattern database is created for each part (Korf and Felner 2002). This is not necessary in Sokoban since the stones are indistinguishable. Thus, we only have to construct one pattern database for each possible size of a part and thereby the construction cost is reduced.

In preliminary experiments we found relevant results with pattern databases with two stones. Using a dynamic-partitioned additive pattern database of two stones, we obtain the highest admissible heuristic value in polynomial time by a maximum weight matching. Moreover, all the specific domain enhancements of Sokoban implemented in Rolling Stone are incorporated.

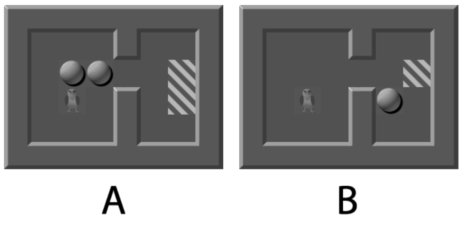

Figure 4: Example of a linear conflict in (A) and a backout conflict in (B).

Thus, this paper focus on pattern databases with two stones. The pattern database store the distance to move the two stones to the entrance, for every possible position of the man. This is computed by a single breadth-first backwards search from the entrance to the maze zone. This pattern database can be quickly computed and captures interactions that improve the lower bound in comparison with the ones found by Rolling Stone. Since the pattern database are constructed exhaustively, if during the search some configuration of the man and two stones is not found in the pattern database, the state must be a deadlock and can be discarded.

This database of two stones automatically incorporates the two enhancements that were presented in Rolling Stone (Junghanns and Schaeffer 2001): backout and linear conflicts. Figure 4 shows an example of a linear conflict in A, and a backout conflict in B. Linear conflicts increase the lower bound with a penalty of two when a pair of adjacent stones is in the optimal path of each other. This increase is achieved by the pattern database since every interaction of the paths of the two stones is calculated exactly. The backout conflict consists in considering the position of the man when his movement is restricted in articulation points by one stone. This is also achieved by the pattern database since that restriction is considered in its construction.

Experimental Results

In this section, we describe the experimental results performed. Initially, in Subsection Lower Bound Results we report the lower bounds obtained by applying the proposed approach. Subsection A Sokoban Solver presents results for a Sokoban solver that takes into account these lower bounds. We would like to emphasize that all results presented here only consider the new lower bound by using pattern databases. If linear and the backout conflicts are added to the goal zone lower bound, the results can be further improved. However in this work we aim at exploring results with the new lower bound without these enhancements.

All experiments were performed on a Core2 Quad Q8200 computer with 4 GB of RAM, and considering the standard set composed of 90 instances[1] with several degrees of difficulty.

[1]http://www.cs.cornell.edu/andru/xsokoban.html, accessed in April 14th, 2013.

Lower Bound Results

Pattern databases are used only in instances which have an entrance square. In case there is not such a special square, the lower bound is calculated applying a minmatching in the whole instance. Considering the standard set, 88 out of 90 instances can be decomposed into a goal zone and maze zones, and thus have an entrance. The only two instances that cannot be decomposed are #7 and #38.

Table 2 shows the lower bounds of the initial states considering the instances from the standard set. MM is the lower bound obtained by the minmatching; RS is the minmatching with enhancements (linear and backout conflicts) proposed by the Rolling Stone solver (Junghanns 1999); PDB is the lower bound using pattern databases. UB is the upper bound obtained from the global Sokoban score file[2]. Note that if a lower bound value matches with that of the UB, then it can be concluded that the corresponding solution length is optimal.

Comparing with RS, the proposed approach obtained better results in 24 instances, and worse in 34 instances. Thus, 24 new best lower bounds were provided. Furthermore, three of these new best lower bounds, namely those of instances #2, #3 and #73, match exactly with the optimal solution length. This was not achieved by MM or RS for these instances.

The good quality of the lower bounds for instances #2 and #3 are obtained due to the fact that the pattern database incorporates interactions of the stones with the instance and the position of the man. For example, for case A of Figure 5, RS using backout conflicts is able to increase the lower bound of MM by 10. However, in this case, RS does not consider the interaction between the stones E8 and H8 (we call the stones accordingly with their initial positions). The pattern database considers the interaction between stones. This is done as follows. The optimal path of stone E8 is build pushing it until location G5, and then to the goal. However, since there is a stone in position H8, the stone E8 is pushed until position G4 because the stone H8 blocks the path the man has to follow to realize the optimal path of stone E8. Next, stone E8 is pushed back to position G5, and then to the goal. In this way, stone E8 leaves the optimal path. Thus, our approach is able to capture the interaction between stones and in this case a penalty cost of two is added to the lower bound, obtaining the exact distance. A similar interaction occurs in case B. Consider the optimal path of each stone when the instance has only one stone. In this case, each stone would be simply pushed to the goal. However, when considering both stones, the man cannot access the initial position. To solve this, stone N7 has to leave its optimal path. First it is pushed to position O7, and stone M8 is able to follow its optimal path. Next, stone N7 is pushed back to its original position, and then to the goal.

In the examples above we explained two cases of interaction of two stones. However, PDB captures the interaction of any pair of stones combined with the man position. This is not the case of RS that considers only the interaction in some special situations. The pattern database can also identify deadlocks not captured by Rolling Stone. The minmatching can detect deadlocks when two stones have to be positioned in a single goal, characterizing a deadlock. In fact, the pattern database can detect any deadlock with the interaction of two stones and the position of the man. This is done by checking in the pattern database: If the configuration does not exist, the lower bound is not calculated and a deadlock has been detected. Figure 6 shows two instances with positions that are identified as deadlocks. In case A the stones E8 and G8, K3 and L3 are in deadlock, and in case B the stones G8 and J8, N8 and O7. All these deadlocks are not identified by the lower bound of Rolling Stone.

#	MM	RS	PDB	UB	#	MM	RS	PDB	UB
1	**95**	**95**	**95**	97	46	219	**223**	**223**	247
2	119	129	**131**	131	47	197	**199**	197	209
3	128	132	**134**	134	48	**200**	**200**	**200**	200
4	331	**355**	**355**	355	49	96	**104**	96	124
5	135	139	**141**	143	50	96	**100**	98	370
6	104	**106**	**106**	110	51	**118**	**118**	**118**	118
7	**80**	**80**	**80**	88	52	365	367	**369**	421
8	**220**	**220**	**220**	230	53	**186**	**186**	**186**	186
9	215	229	**231**	237	54	177	177	**181**	187
10	494	**506**	496	512	55	118	**120**	**120**	120
11	197	**207**	205	241	56	191	**193**	191	203
12	**206**	**206**	**206**	212	57	215	**217**	215	225
13	**220**	**220**	**220**	238	58	189	**197**	**197**	199
14	**231**	**231**	**231**	239	59	**218**	**218**	**218**	230
15	94	**96**	94	122	60	**148**	**148**	**148**	152
16	160	**162**	**162**	186	61	241	**243**	241	263
17	121	**201**	**201**	213	62	235	237	**239**	245
18	90	**106**	90	124	63	425	427	**429**	431
19	278	**286**	**286**	302	64	331	367	**373**	385
20	302	**446**	358	462	65	181	**203**	195	211
21	123	**131**	123	147	66	185	**187**	185	325
22	306	**308**	306	324	67	367	377	**385**	401
23	286	424	**430**	448	68	317	321	**325**	341
24	442	**518**	442	544	69	207	**219**	209	433
25	326	**368**	336	386	70	**329**	**329**	**329**	333
26	149	**163**	161	195	71	290	**294**	290	308
27	351	353	**355**	363	72	284	**288**	286	296
28	284	286	**290**	308	73	433	437	**441**	441
29	124	122	**128**	164	74	158	**172**	170	212
30	357	359	**385**	465	75	261	**263**	261	295
31	228	**232**	**232**	250	76	192	**194**	192	204
32	111	**113**	111	139	77	**360**	**360**	**360**	368
33	140	**150**	140	174	78	134	**136**	**136**	136
34	152	**154**	152	168	79	164	**166**	164	174
35	362	**364**	**364**	378	80	219	**231**	225	231
36	501	**507**	505	521	81	**167**	**167**	**167**	173
37	220	**242**	220	284	82	131	**135**	**135**	143
38	**73**	**73**	**73**	81	83	190	**194**	**194**	194
39	650	**652**	**652**	672	84	147	149	**151**	155
40	310	310	**312**	324	85	303	303	**305**	329
41	201	**221**	207	237	86	**122**	**122**	**122**	134
42	**208**	**208**	**208**	218	87	**221**	**221**	**221**	233
43	132	132	**134**	146	88	306	**336**	314	390
44	167	167	**169**	179	89	345	353	**355**	379
45	274	284	**286**	300	90	436	**442**	438	460

Table 2: Lower bounds. The bold values sign the best lower bound for each instance.

[2]http://www.cs.cornell.edu/andru/xsokoban/scores.txt, accessed in April 14th, 2013.

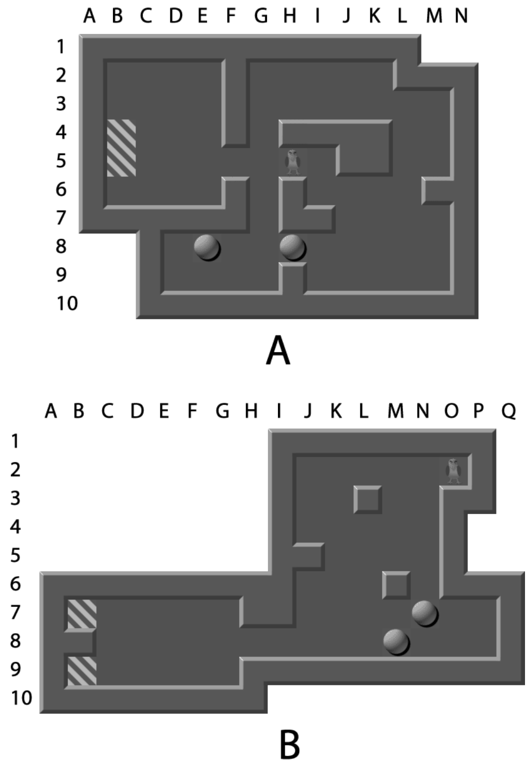
A B C D E F G H I J K L M N
1 2 3 4 5 6 7 8 9 10
A
A B C D E F G H I J K L M N O P Q
1 2 3 4 5 6 7 8 9 10
B

Figure 5: Example of penalties captured in instances #2 and #3 that are not captured by the Rolling Stone. In case (A) and (B) a penalty of two is added to the lower bound and so the exact distance to solution is obtained.

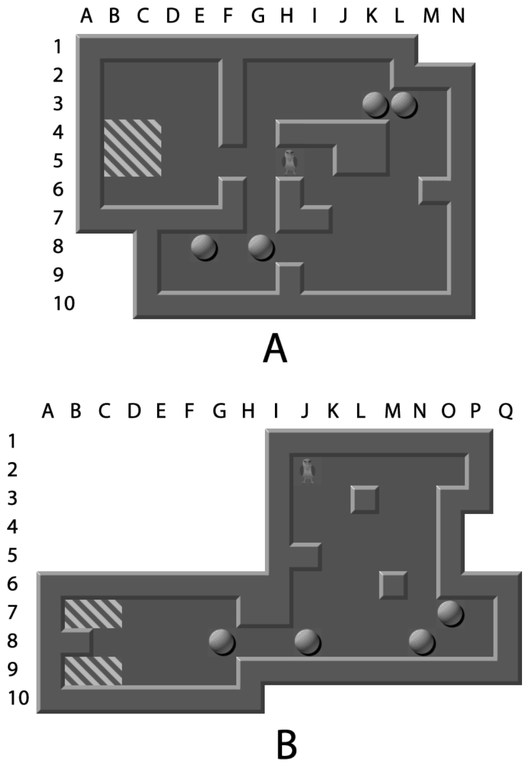
A B C D E F G H I J K L M N
1 2 3 4 5 6 7 8 9 10
A
A B C D E F G H I J K L M N O P Q
1 2 3 4 5 6 7 8 9 10
B

Figure 6: Deadlocks detected by the pattern database. In case (A) the stones E8 and G8, K3 and L3 are in deadlock, and in the case (B) the stones G8 and J8, N8 and O7.

A Sokoban Solver

In this subsection we report results obtained by a Sokoban solver, using the proposed lower bound, within a limit of 5 million explored nodes. The solver is based on an A^* using a bitmap for storing the states from the closed set. The search is guided taken into consideration the proposed lower bound. Nine instances were solved to optimality.

Table 3 shows the number of explored nodes. In the table, column PDB shows results found by our solver. The column Pre reports the number of states used to construct the PDB for each instance. Both columns Static and Dynamic inform the total number of explored nodes for the construction of the pattern database and for solving the instance. The column Static is the statically-partitioned PDB where bipartitions are considered. In one partition the stones are grouped linearly. For example, in an instance with six stones, stone 0 is paired with stone 1, stone 2 with 3, and the stone 4 with 5. For the other partition, the first half of the stones are paired with the second half. Therefore, in the six-stone instance example, the stone 0 is paired with stone 3, the stone 1 with 4, and the stone 2 with 5. For each partition, the cost of each pair is consulted in the PDB and the maximum over the two partitions is added to the solution total cost. The column Dynamic refers to the proposed approach using a dynamic-partitioned PDB calculated by a maxmatching. Column RS-LB reports results of the solver using A^* and a bitmap for storing the states from the closed set guided only by the enhanced lower bound from the Rolling Stone with the linear and backout conflicts. We compare our results with RS-LB, since the only difference between both, in this case, is the lower bound used. Thus, the results attempt to evidence the more efficient lower bound. RS-ALL is the Rolling Stone using several techniques that guarantee optimality: the enhanced lower bound, the move ordering, the tunnel macros, and the deadlock tables with approximately 22 million entries. The nine instances in this table are those that were solved by at least one of the considered approaches, within the imposed search limits. Since we are only considering approaches with optimality guarantees, we only compare results from RS that attend this characteristic.

The instance #38 cannot be decomposed and thus it was solved using just the minmatching. In column Pre it can be observed that the number of states explored when creating the pattern database is very small when comparing with the cost of solving the instance. For the 90 instances from the standard set, in the worst case, 4,248 nodes were explored by the construction of the PDB (for instance #23).

Using only the lower bound, without any enhancement, we can solve six instances with the statically-partitioned PDB and nine with the dynamic-partitioned PDB. By the experimental results performed we concluded that the dynamic-partitioned PDB brings better results for Sokoban because with this approach we were able to solve more in-

#	PDB			RS-LB	RS-ALL
	Pre	Static	Dynamic		
1	842	8,003	2,972	26,688	223
2	624	27,745	6,602	>	620,030
3	661	1,821,648	1,237,663	>	>
6	497	>	304,705	>	>
17	1,550	306,059	18,536	2,141,190	>
38	0	199,189	199,189	199,189	415,485
55	971	>	136,768	>	>
78	740	148,211	9,294	3,603,501	871
83	1,413	>	67,520	>	>

Table 3: Number of explored states by Sokoban solvers. A symbol '>' indicates that no results were obtained within the search limit of 5 million explored nodes.

#	PDB		RS-LB
	Static	Dynamic	
1	0.33	0.45	0.99
2	2.68	3.57	>
3	113.66	221.56	>
6	>	55.56	>
17	6.26	1.52	34.18
38	6.86	6.86	7.21
55	>	46.12	>
78	66.28	5.54	1723.16
83	>	82.66	>

Table 4: Computational times in seconds spent by Sokoban solvers to solve each instance. A symbol '>' means that no results were obtained within the search limit of 5 million explored nodes.

stances and always using less explored states.

In column RS-LB one can analyze the effect of using RS lower bound in place of our proposed LB. In this case only four instances could be solved. Moreover, considerably more nodes were explored in the resolution. The efficiency of pattern databases becomes more evident when we compare with RS-ALL, which uses several specific domain enhancements, and again solves only four instances.

In Table 4 we report the computational times in seconds obtained by the Sokoban solvers. Column Static refers to the statically-partitioned PDB, while column Dynamic refers to the dynamic-partitioned PDB. Column RS-LB presents the computational times for the solver using the RS lower bound by our solver.

The overall time results from PDB are always better than the ones from RS-LB. This indicates that the computation of the zones and the construction of PDB are not significantly time expensive, even for instances with small number of explored nodes for the solution, like for instance #1. For solving instance #78, for example, PDB uses considerable less computational time than RS-LB. Comparing Static with Dynamic PDBs, when the difference between the number of explored nodes is small, the cost of computing the maxmatching makes the Static PDB to present better time results. But, for instances like the #78 the computational time difference becomes significant.

The Static PDB achieved a mean of 21,695 nodes per second for the solved instances and the Dynamic PDB a mean of 7,339 nodes per second. As expected, the inclusion of the calculation of the maxmatching in the Dynamic PDB reduced the number of explored nodes per second. The RS-LB achieved a mean of 29,794 nodes per second, which is comparable with the results from the PDB lower bounds. The Rolling Stone solver explores 20 million nodes in about 3 hours. Updating the values for current computers the Rolling Stone solver can explore about 29,630 nodes per second. This is comparable with our results.

A property that improves the lower bound in Sokoban is the capacity of detecting deadlocks. The RS lower bound detects deadlocks only when two stones can go only for a single goal. The PDB can detect all deadlocks formed by two stones. The Dynamic PDB with the maxmatching classifies as deadlock 15,38% and with the minmatching 0,61% of the calculated lower bounds. The RS-LB classified as deadlock only 1,2% of the calculated lower bounds. Thereby, the PDB can detect considerable more deadlocks than RS-LB.

Both the maxmatching and minmatching have simple implementations, and both can be improved with optimized implementations such as dynamic updates, and identifying situations where the recalculation is not necessary. But since the PDB lower bounds also use the minmatching, every improvement in the RS-LB generates an improvement in the PDB lower bounds.

Considering the series of analysis presented in this section about LB quality, number of explored nodes and computational times, we conclude that the use of PDB in Sokoban yields significant results. Since, the proposed approach can solve more instances with optimality guarantees, using less nodes, spending less computational time, and detecting more deadlocks, we conclude that PDBs provides results that are competitive with the state of the art results of approaches for solving Sokoban.

Conclusions and Future Work

In this paper we propose an approach that uses instance dependent pattern databases in Sokoban. The goal of this work is to show the power of pure pattern databases as a lower bound. The PDB can capture all interactions between the man and the stones, increasing the value of the lower bound. Moreover, it early detects more deadlocks than the state of the art lower bounds.

The proposed approach found optimal solutions for nine instances of the standard set of Sokoban overcoming results from the literature for methods that guarantee optimality. We also have found better lower bounds for initial states in 24 instances. Moreover, if domain specific enhancements are added to the proposed approach, the results are likely to improve even further.

In a next step of this work, we intend to incorporate domain specific enhancements to the goal lower bound, such as backout and linear conflicts. Moreover, we plan to investigate dynamically partitioned pattern databases (Felner, Korf, and Hanan 2004) with more than two stones.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2002. Using abstraction for planning in Sokoban. In *Computers and Games*, 360–375.

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

Culberson, J. C. 1998. Sokoban is PSPACE-Complete. In *Proceedings of the International Conference on Fun with Algorithms*, 65–76.

Damgaard, B. 2013. Sokoban YASS. http://sourceforge.net/projects/sokobanyasc/.

Demaret, J.-N.; Lishout, F. V.; and Gribomont, P. 2008. Hierarchical planning and learning for automatic solving of Sokoban problems. In *20th Belgium-Netherlands Conference on Artificial Intelligence*, 57–64.

Edelkamp, S., and Schrödl, S. 2012. *Heuristic Search - Theory and Applications*. Academic Press.

Edelkamp, S. 2001. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning*, 13–24.

Felner, A., and Adler, A. 2005. Solving the 24 puzzle with instance dependent pattern databases. In *SARA*, 248–260.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *J. Artif. Intell. Res. (JAIR)* 22:279–318.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics* 4(2):100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, 1007–1012.

Junghanns, A., and Schaeffer, J. 1999. Domain-dependent single-agent search enhancements. In *IJCAI*, 570–577.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence* 129(1-2):219–251.

Junghanns, A. 1999. *Pushing the Limits: New Developments in Single-Agent Search*. Ph.D. Dissertation, University of Alberta.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1-2):9–22.

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

Korf, R. E. 1997. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, 700–705.

Korf, R. E. 2012. Research challenges in combinatorial search. In *AAAI*.

Meger, M. 2013. JSoko. http://sourceforge.net/projects/jsokoapplet/.

Takahashi, K. 2013. Takaken Solver. http://www.ic-net.or.jp/home/takaken/e/soko/.

Zhou, R., and Hansen, E. A. 2004. Space-efficient memory-based heuristics. In *AAAI*, 677–682.