Solving Sokoban Optimally using Pattern Databases for Deadlock Detection

André G. Pereira*, Marcus Ritt†, Luciana S. Buriol‡
Institute of Informatics
Federal University of Rio Grande do Sul, Brazil
{agpereira*,marcus.ritt†,buriol‡}@inf.ufrgs.br

***Abstract*—A *pattern database* (PDB) is a look-up table that stores the exact distance of a set of abstract states to some abstract goal state. PDBs are the most promising approach to derive admissible heuristic functions in several problem domains. Sokoban is a hard domain of research in artificial intelligence. The domain-specific characteristics of the problem and the implicit definition of the goal states makes standard PDBs an ineffective heuristic function for Sokoban. We show how to apply standard PDBs as an effective approach to deadlock detection, increasing the number of instances solved with optimality guarantees. Using the proposed approach, we are able to detect five times more deadlocks than the standard heuristic function of Sokoban, solving optimally two more instances, while exploring an order of magnitude less nodes.**

***Keywords*—Sokoban, Pattern databases, Single-agent search, Heuristic search, A*.**

I. Introduction

A *state space* problem can be defined as a quadruple $P = (S, A, t, G)$, where S is the set of states, $t \in S$ is an initial state, G is the set of goal states, and A is the set of operators $A = \{a_1, a_2, \ldots, a_n\}$. Each operator a_i transforms a state $u \in S$ into another state $v \in S$. The set of goal states can be defined explicitly by a set of states or implicitly by a set of goal conditions. Standard single-agent heuristic search algorithms like A* [1] and IDA* [2] aim to find the shortest path from t to some goal state using the evaluation function $f(s) = g(s)+h(s)$. Function $g(s)$ represents to the distance from state t to state s and $h(s)$ is a heuristic function that estimates the distance to reach some goal state from s. A heuristic function is admissible, if it is a lower bound on the shortest path from a state s to any goal state. Using an admissible heuristic function A* and IDA* always find an optimal solution. An admissible technique or solver is one that guarantees optimality.

Sokoban is a PSPACE-complete problem [9] and a well known testbed for artificial intelligence techniques. The problem is defined on a maze, represented by a grid of squares with *immovable blocks* (walls) and *free squares*. The *man* is a movable block which can move between adjacent free squares. There are k goal squares and k movable blocks (stones). The man can push a stone to an adjacent free square. The set of goal states G is defined implicitly as all states in which each stone is placed on a distinct goal square. An optimal solution is a shortest sequence of operations (moves) that moves all stones to the goal squares. Movements of the man are not accounted. A state in Sokoban is completely defined by the position of the stones and by the reachable component of the man. The reachable component of the man corresponds to all the free squares accessible to the man for a given a placement of stones in the maze. Finally, a *dead square* is a square such that a stone on it cannot reach any goal square. This is an example of an obvious deadlock. A state s is in *deadlock* if no goal state is reachable from s.

TABLE I: Search space characteristics of some single-agent search problems [10].

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 I shows some characteristics that makes Sokoban harder to solve than other common state space problems like 24-Puzzle and Rubik's Cube. Sokoban has longer solutions, a greater branching factor and a large state space. Besides these characteristics the presence of deadlocks makes the problem even harder. Unlike other testbeds in artificial intelligence, humans do better than computers in the task of solving Sokoban since all instances have been solved by humans but no computer could solve all of then. Sokoban is a simplified model of general robot motion planning which is a fundamental problem in robotics and has a large range of applications [11].

In recent years the most effective approach to produce admissible heuristic functions for several problems are *pattern databases* (PDBs), introduced by Culberson and Schaeffer [3]. A PDB computes and stores in a look-up table the exact distances among sets of abstract states to some abstract goal state in an *abstract state space*. An abstract state space is defined by an abstraction function ϕ that, in general, maps the state space S into a small abstract state space S' and in which the distance between abstract states u' and v' does not exceed the shortest path in the original state space between states u and v. Once ϕ is defined, a PDB is constructed in a preprocessing phase by a backwards search from the set of abstract goal states G', storing in the look-up table the distance to reach each abstract state s'. During the search each state is mapped to its respective abstract state and the information stored in the PDB is used as an admissible heuristic function. PDBs achieved new state-of-the-art results in several problems such as Sliding-Tile puzzle [3], Rubik's Cube [4], domain-independent planning [5], vertex cover [6], multiple sequence alignment [7] and model checking [8].

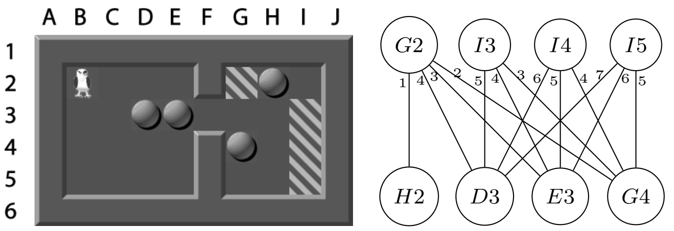

Fig. 1: Standard heuristic function of Sokoban computed by a minimum cost perfect matching in a bipartite graph.

In this paper we introduce the idea of using standard PDBs in Sokoban for deadlock detection. Because of the implicitly defined set of goal states and the domain-specific characteristics of the problem standard PDBs in Sokoban lead to ineffective heuristic functions. We show that a standard PDB nevertheless can effectively detect deadlocks, and may be used together with other heuristic functions to increase the number of optimally solved instances. We start by reviewing some related work. Then we show how to construct the PDB, how to compute the heuristic function, and explain why it is ineffective. Finally, we present computational experiments to show the effectiveness of the standard PDB to detect deadlocks, and how this technique can be used with another heuristic function to solve instances of Sokoban.

II. Related Work

The Rolling Stone (RS) solver proposed by Junghanns and Schaeffer [10] is an important milestone in the research on Sokoban. RS is built on an IDA* using domain-independent and domain-dependent enhancements. The solver was developed to find some solution for the largest possible number of instances. Thus, several non-admissible techniques have been introduced. The non-admissible version of RS is able to solve 57 instances. There is also an admissible version of RS (which we call RS*). RS* is able to solve 6 instances optimally. RS introduced two techniques for deadlock detection: deadlock tables and pattern searches. A deadlock table is an admissible technique to detect deadlocks in a small window of five by four squares. All possible configurations of stones, walls and the man in this small window are enumerated and eventual deadlocks are recorded. Deadlock tables of this size have more than 20 million entries and their construction takes several days. During the search the information of deadlock tables is used to detect deadlocks in windows of five by four. A pattern search is a non-admissible technique that aims at finding deadlocks and calculates penalties during the search by solving subsets of stones. Both techniques detect a small subset of the total possible deadlocks that occur in Sokoban.

RS also introduced the *enhanced minimal matching* (EMM), the standard heuristic function for Sokoban. EMM is computed using three components: backout conflicts (BC), a minimum cost perfect matching, and linear conflicts (LC). BCs compute distances for pushing a stone from each square to each other square considering the position of the man when his movement is restricted in articulation squares by a stone. A lower bound on the minimal number of pushes needed to bring the stones to the goal squares is computed by a minimum weight perfect matching between stones and goal squares, where the weight of the edge between stone s and goal square g is the smallest number of pushes needed to bring s to g. The resulting graph can be seen in the right part of Figure 1. Finally, LC adds a penalty of two when a pair of adjacent stones is in the optimal path of each other. This type of conflict can be seen between the stones in positions $D3$ and $E3$ in the left part of Figure 1. A good heuristic function for Sokoban must be effective in detecting deadlocks since a state in deadlock has infinite distance to any goal state. The EMM can detect non obvious deadlocks, for example, a state where two stones can be pushed only to a single goal square.

There are other non-admissible solvers in the literature for Sokoban [12, 13]. However, the best non-admissible solvers are have been developed by the community interested in Sokoban[1].

PDBs were introduced in domain-independent planning by Edelkamp [5]. In this work a set of easy Sokoban instances was used in the experiments. The problem was transformed into a problem with an explicit defined goal state by an arbitrary assignment of stones to goal squares. The approach introduced by Haslum et al. [14] obtains the best results with PDBs in domain-independent planning. In a simple way, they build a standard PDB trying to select good abstractions automatically. Their work also uses a set of easy Sokoban instances for evaluation. A state-of-the-art implementation of PDBs according to Haslum et al. [14] does not solve any standard instance of Sokoban.

PDBs have already been introduced as a domain-dependent technique to Sokoban in the work of Pereira et al. [15]. This approach solves the problem of the implicit definition of the goal states using an instance decomposition introducing a explicitly defined intermediate goal state. The intermediate goal state allows to decompose the original problem into two subproblems in two zones of the instance, the maze zone and the goal zone. For the maze zone subproblem they build an intermediate pattern database (IPDB) and for goal zone subproblem they use the minimum cost perfect matching with simple distances. This approach is the currently best admissible heuristic function for Sokoban, and is able to solve nine instances whereas RS* is able to solve four with the same limit of five million of explored nodes. IPDB also effectively detects deadlocks. However, it can only detect deadlocks that occur completely in the maze zone.

III. Pattern Databases for Sokoban

In this section we show how to build standard PDBs efficiently, how to compute the heuristic function and explain why this heuristic function presents poor results in Sokoban.

A. Pattern Database Construction

In domains like Sliding-Tile puzzle or Rubik's Cube the construction of the PDB can take several hours or even days. This is acceptable, since the PDB can be used to solve several different instances of the same problem and the construction cost can be amortized over several executions. However, this is not the case for Sokoban since each instance is a different problem with a different state space. Thus, a PDB for Sokoban must be constructed efficiently for each instance. This limits the size of the PDB. Furthermore the PDB must be effective

[1]http://www.sokobano.de/wiki/, accessed in June, 2014.

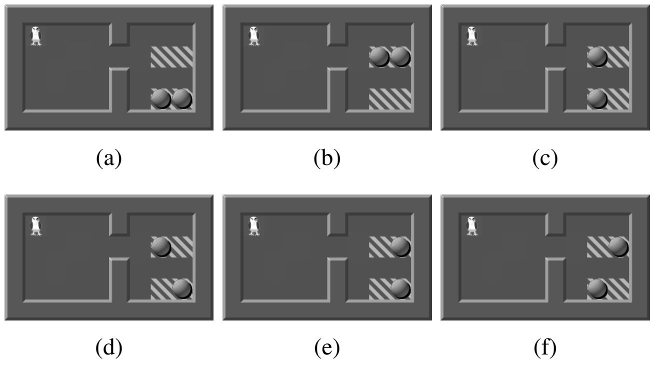
Fig. 2: Initial abstract states used for building an MPDB.

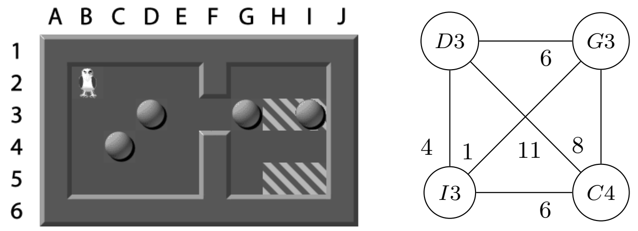

Fig. 3: Example instance and general graph for compute the heuristic value from a MPDB.

enough to amortize the cost of its construction. We start by describing how to obtain an abstraction for Sokoban, then explain how to build the PDB efficiently and how a PDB can be used for deadlock detection, and finally compare it to other approaches.

An abstraction for a Sokoban instance with k stones can be defined considering only a subset of k' stones. In other words, if we remove from a state s with k stones all but k' stones we obtain a subproblem of the original problem. A goal state for the abstraction is a state where each of the k' stones is placed on a different goal square. This abstraction is admissible since the cost for solving an instance with a subset of stones never exceeds the cost of solving the same instance with all stones. A standard PDB is constructed from the set of abstract goal states G'. The set of abstract goal states for a simple instance can be seen in Figure 2.

The set of abstract goal states serves as the set of initial states for the reverse search during the construction of the PDB. Since these states are abstract goal states they have cost 0. Each reverse move applied to an abstract state increases the distance to the set of abstract goal states by one. The PDB is completely constructed when the whole abstract state space has been explored and each abstract state has its shortest distance to the set of abstract goal states stored in the look-up table. We call a PDB which is built from various abstract goal states a *multiple goal state PDB* (MPDB). The number of stones in the abstraction determines the size of the MPDB: an MPDB-k' is built from an abstraction of k' stones.

An MPDB-2 can be stored in a three-dimensional array, using two indices for the position of each stone and one index for the position of the man. In this representation each different position of the man or the stones represents a different entry in the MPDB. This is possible because an MPDB-2 has a small number of entries. An array enables fast queries since the index for the positions of the stones and the man can be computed in constant time. For an MPDB-k' with $k' > 2$ we use a sparse storage based on a hash table. We store an entry for each unique abstract state, i.e., a unique placement of k' stones with a different reachable component for the man. This reduces the memory used by the MPDB significantly, but increases the cost of a query, since we have to sort the stones in a specific order, and we have to determine the reachable component of the man.

An MPDB can also be built to detect deadlocks. In this case we do not store the distances for all abstract states, but only the reachable abstract states. This further decreases the memory used by the MPDB. The MPDB-k' is built exhaustively. Therefore, each placement of k' stones with a reachable component of the man not contained in the MPDB must be a deadlock. An MPDB detects strictly more deadlocks than an IPDB since the latter can only detect deadlocks completely contained in the maze zone of the instance. MPDBs do not have this restriction. An MPDB-2 detects all deadlocks detected by the EMM. Actually, MPDBs are able to detect all deadlocks formed by k' stones. Furthermore, deadlock tables detect deadlocks only in a small part of the instance. Again MPDBs do not have this restriction. A pattern search also depends on implementation design decisions and can miss non obvious deadlocks.

B. Heuristic Value Computation

Given an MPDB we have to compute the heuristic value. Since each entry in the MPDB stores the cost of solving a subset of stones, we can sum the cost of solving disjoint subsets of stones, and still obtain an admissible heuristic function. Such a PDB is called *additive*. Since the interaction of stones changes during the search, we chose to use dynamic-additive PDBs, where for each state in the search we compute a different partition of stones into disjoint subsets. For an MPDB-2, a maximum weight matching in a general graph yields the highest heuristic value in polynomial time. For an MPDB-k' with $k > 2$ the corresponding matching problem becomes NP-complete, and thus heuristic approaches must be employed, as described below.

To compute the heuristic value for an MPDB-2 we build a graph with a vertex for each stone. Each pair of vertices is connected by an edge with a weight equal to the distance stored in the MPDB-2 for the corresponding pair of stones. This distance is defined as the shortest path from the pair of stone positions to the closest pair of goal squares. If the number of stones is odd, an extra vertex is added to the graph and it is connected by an edge to all other vertices in the graph. Each of these edges has a weight equal to the number of moves on the shortest path from the current position of the stone to some goal square. This graph and the respective instance can be seen in the Figure 3. After the graph is constructed we compute a maximum weight matching to find the highest heuristic value.

If the abstraction is composed of more than two stones the problem of computing the highest heuristic value becomes equivalent to the maximum weight exact cover, an NP-complete problem. Since we have to solve this problem for each state in the search we use a fast and simple greedy randomized constructive heuristic. We start by querying the distance of every subset of k' stones in the MPDB and sort these distances in order of non-increasing increments, where the increment is the difference of the value stored in the MPDB and the sum of the distances for each stone to reach the closest goal square. Each of the k stones in the state is allowed to be part of only one selected subset of k' stones. Thus, selecting a subset will disable other subsets which include the same stone. We heuristically generate $k + 1$ partitions of the k stones as follows. The first partition is obtained by greedily choosing the subset of highest increment, until all stones are covered. The remaining k partitions are obtained by greedy randomized strategy. This strategy chooses some random subsets from the first m subsets, and then completes it greedily. We repeat this strategy k times with $m = i\binom{k}{k'}/k$ in iteration $i = 1, \ldots, k$. Thus, in the last iteration we choose randomly among all subsets. The highest heuristic value obtained in all $k + 1$ partitions is used as the heuristic value. Observe that the heuristic obtained in this way is still admissible, since every partition of the stones yields a lower bound in the shortest distance to bring the stones to the goals.

The heuristic function based on MPDB is ineffective compared to other heuristic functions like EMM or IPDB, since the MPDB is built from a set of abstract goal states. Each subset of stones will be assigned to some closest subset of goal squares. This ignores that each stone must be placed on an unique goal square. It also looses information when goal squares are occupied by stones. In this case the heuristic function may assign other stones to the goal squares that are already occupied. In Figure 3 every subset is assigned to the same abstract goal state Figure 2b. The highest heuristic value obtainable by an MPDB-2 for the state in Figure 3 has value 12, while the heuristic value obtained by EMM is 16.

IV. Experimental Results

In this section we present experimental results performed with the proposed approach. We first compare i) EMM, the standard heuristic function of Sokoban; ii) IPDB, the state-of-the-art heuristic function [15], and iii) MPDB. Next we compare the MPDB to the EMM, measuring the effectiveness in detecting deadlocks. Finally, we run several experiments to evaluate the ability of each technique in finding optimal solutions for Sokoban. All experiments have been performed on a PC with an AMD Opteron processor running at 2.34 GHz with 32 GB of main memory. In all experiments no MPDB uses more than 2 GB of memory. All methods use the standard set of 90 instances[2].

TABLE II: Heuristic value on the initial states of the standard set of 90 instances.

#	E	I	M	U	#	E	I	M	U
1	**95**	**95**	91	97	46	**223**	**223**	191	247
2	129	**131**	117	131	47	**199**	**199**	163	209
3	132	**134**	116	134	48	**200**	**200**	150	200
4	**355**	**355**	311	355	49	**104**	**104**	72	124
5	139	**141**	121	143	50	100	**102**	83	370
6	**106**	**106**	94	110	51	**118**	**118**	84	118
7	**80**	**80**	68	88	52	367	**369**	319	421
8	**220**	**220**	192	230	53	**186**	**186**	164	186
9	229	**231**	206	237	54	177	**181**	160	187
10	**510**	**510**	349	512	55	**120**	**120**	102	120
11	**207**	**207**	174	241	56	**193**	**193**	170	203
12	**206**	**206**	174	212	57	**217**	**217**	183	225
13	**220**	**220**	165	238	58	**197**	**197**	178	199
14	**231**	**231**	206	239	59	**218**	**218**	194	230
15	**96**	**96**	86	122	60	**148**	**148**	127	152
16	162	**166**	118	186	61	245	**249**	197	263
17	**201**	**201**	197	213	62	237	**239**	196	245
18	**106**	**106**	87	124	63	427	**429**	383	431
19	**286**	**286**	254	302	64	367	**373**	341	385
20	**446**	**446**	360	462	65	**203**	**203**	170	211
21	**129**	**129**	89	147	66	**187**	**187**	158	325
22	**308**	**308**	250	324	67	377	**385**	323	401
23	426	**430**	383	448	68	321	**325**	283	341
24	**518**	**518**	414	544	69	**219**	**219**	183	433
25	368	**370**	261	386	70	**329**	**329**	281	333
26	165	**167**	138	195	71	**294**	**294**	261	308
27	353	**355**	295	363	72	**288**	**288**	196	296
28	286	**290**	208	308	73	437	**441**	408	441
29	122	**128**	107	164	74	176	**178**	164	212
30	359	**385**	320	465	75	**263**	**263**	216	295
31	**232**	**232**	176	250	76	**194**	**194**	156	204
32	113	**115**	94	139	77	**360**	**360**	255	368
33	**152**	**152**	119	174	78	**136**	**136**	124	136
34	**154**	**154**	128	168	79	166	**168**	144	174
35	**364**	**364**	316	378	80	**231**	**231**	213	231
36	**507**	**507**	447	521	81	**167**	**167**	145	173
37	**242**	**242**	187	284	82	**135**	**135**	117	143
38	**73**	**73**	60	81	83	**194**	**194**	182	194
39	**652**	**652**	535	672	84	149	**151**	135	155
40	310	**312**	275	324	85	**305**	**305**	238	329
41	221	**223**	166	237	86	**122**	**122**	97	134
42	**208**	**208**	139	218	87	**221**	**221**	201	233
43	132	**134**	116	146	88	**336**	**336**	246	390
44	167	**169**	158	179	89	**353**	**353**	274	379
45	284	**286**	224	300	90	**442**	**442**	244	460
					AVG	241	242	200	262

A. Heuristic Value on Initial States

One way to compare the effectiveness of a proposed heuristic function is to evaluate the heuristic value on the initial states of the standard set of instances. The work of Pereira et al. [15] provides these results for an IPDB built from an abstraction with two stones. Therefore we compare our MPDB with their results and the standard heuristic function of Sokoban. Table II provides these results. Column “#” gives the number of the instance, columns “E”, “I” and “M” report the heuristic value for the EMM, IPDB, and MPDB, respectively, and the last column “U” reports the best known upper bounds on the solution length for each instance.

Clearly the IPDB dominates both the MPDB and EMM. The average heuristic value is one unit higher than that of EMM, and for some instances considerably more (e.g. for #30 the value is 26 better). It is also clear that the MPDB yields poor heuristic values, which are always dominated by the EMM and the IPDB. This is expected since this heuristic function loses much information by the construction of the set of abstract goal states. This also indicates that MPDB is not an effective heuristic function for Sokoban.

B. Deadlock Detection

Sokoban instances are built such that most of the possible movements generate deadlocks, sometimes non obvious ones. A Sokoban solver can spend a great effort trying to solve subsets of states in deadlock. Therefore, the capacity of deadlock detection is crucial for an effective Sokoban solver.

[2]http://www.cs.cornell.edu/andru/xsokoban.html, accessed in June, 2014.

This section evaluates the capability for detecting deadlocks. To this end we generate $10,000$ random states for each of the 90 instances of the standard set. Each state is generated starting with an empty instance. Then we repeatedly place a stone on a random non-dead square, until the number of stones is equal to the number of goal squares. This avoids the creation of obvious deadlocks. Finally we place the man on a random free square. This experiment aims to show how the heuristic function is effective during the search in detecting deadlocks and in heuristic values. The work Pereira et al. [15] does not report this experiment. Thus we only compare our MPDB with two different sizes of abstractions with the EMM.

Table III shows the mean heuristic value of the $10,000$ randomly generated states of EMM and MPDB in columns "HV", and the number of detected deadlocks in columns "DL". Column "#" gives the number of the instance. We can expect that MPDB-2 detects more deadlocks than EMM, since MPDB-2 can detect all deadlocks formed by two stones. We can also expect that MPDB-4 dominates MPDB-2, since MPDB-4 detects all deadlocks formed by four stones.

EMM obtains a mean heuristic value of 165 and detects $1,318$ states in deadlock. MPDB-2 has a mean heuristic value of 137. As expected this value is lower than the one provided by EMM, but the MPDB-2 detects $8,168$ deadlocks and the MPDB-4 detects $8,981$ deadlocks. This shows that EMM in average fails to detect about 77% of the deadlocks, and spends a larger amount of the computational effort in states that will never lead to any goal state. This also suggests that we can build a more effective solver if we combine an effective heuristic function like EMM with an effective deadlock detection technique like MPDB.

C. Solving Instances

In this final experiment we compare different approaches to obtain optimal solutions to Sokoban. To this end we implemented an efficient A* algorithm. It is not fair to compare our results directly with the ones from the RS* since it uses an IDA* algorithm and A* has the advantage of being able to perfectly detect duplicates. We use the same set of tie-breaking rules from RS. Thus our base implementation of A* using the EMM as a heuristic function would be equivalent to the RS* if it used an A* algorithm. The results for this version of the solver are shown in column "EMM" in Table IV. These results are better than the ones reported in [15] without using a domain-specific tie-breaking rule. Column "MPDB" shows the results for the solver using the MPDB as the heuristic function for an abstraction with two and four stones. Column "EMM+MPDB' shows the results when using the EMM as the heuristic function and the MPDB only for deadlock detection. Again two sizes of abstractions have been tested. All tests have been performed with a limit of one hour of computation time and 5 million explored nodes. We also limit the size of the abstraction to four, since for some instances is not possible to build the MPDB with an abstraction of five stones in an hour. The reported time always includes the time used to build the MPDB and to solve the instance. The number of nodes does not include the nodes explored to build the MPDB, since in these nodes it is not necessary to compute a heuristic value and they are therefore processed much faster.

Instances which could not be solved within the above limits are marked with the symbol ">". The table shows only instances that could be solved by some of the approaches. MPDB-4 could solve only three instances and MPDB-2 only two. EMM could solve ten instances, which is more than the six instances that RS* was able to solve. This is expected, since RS* was designed to consume less memory, and uses IDA*, which we have replaced by A*. EMM+MPDB-2 solves eleven instances and EMM+MPDB-4 twelve instances. EMM+MPDB-4 explores fewer nodes than the other approaches. Ten of the eleven instances that EMM+MPDB-2 solves, it solves faster than the other approaches. These results show that the proposed approach to use MPDB for deadlock detection is effective even considering the higher preprocessing costs.

V. Conclusions and Future Work

We have proposed an effective approach for optimally solving Sokoban, by combining the use of a good heuristic function such as the EMM with an effective deadlock detection technique. We have shown that MPDB can serve the role of an efficient deadlock detector. We solve three more instances exploring an order of magnitude less nodes compared to the standard heuristic function of Sokoban. A way to improve MPDBs could be to find better heuristics for finding good partitions in the case of abstractions with more than two stones. The existence of an alternative method to apply MPDBs as an efficient heuristic remains another open research question.

References

[1] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[2] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[3] J. C. Culberson and J. Schaeffer, "Searching with pattern databases," in *Canadian Conference on Artificial Intelligence*, 1996, pp. 402–416.

[4] R. E. Korf, "Finding optimal solutions to Rubik's cube using pattern databases," in *AAAI Conference on Artificial Intelligence*, 1997, pp. 700–705.

[5] S. Edelkamp, "Planning with pattern databases," in *European Conference on Planning*, 2001, pp. 13–24.

[6] A. Felner, R. E. Korf, and S. Hanan, "Additive pattern database heuristics," *J. Articial Intelligence Res.*, vol. 22, pp. 279–318, 2004.

[7] R. Zhou and E. A. Hansen, "Space-efficient memory-based heuristics," in *AAAI Conference on Artificial Intelligence*, 2004, pp. 677–682.

[8] S. Edelkamp, F. Informatik, and A. Lluch-lafuente, "Abstraction in directed model checking," in *ICAPS-Workshop on Connecting Planning Theory with Practice*, 2004.

[9] J. C. Culberson, "Sokoban is PSPACE-Complete," in *Fun With Algorithms*, E. Lodi, L. Pagli, and N. Santoro, Eds., 1999, pp. 65–76.

[10] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowl-

TABLE III: Average value of the heuristic function and number of deadlocks over $10,000$ randomly generated initial states for the standard set of instances.

#	EMM		MPDB-2		MPDB-4	
	HV	DL	HV	DL	HV	DL
1	60	76	58	5,669	59	6,434
2	76	142	68	6,545	69	7,764
3	80	9	69	7,153	71	8,173
4	224	2,463	196	8,612	199	9,194
5	92	4	79	7,619	82	8,456
6	63	2,491	56	6,897	59	7,928
7	56	0	48	5,779	53	8,079
8	141	37	120	7,056	124	8,377
9	144	3,935	129	8,685	133	9,106
10	335	1,315	216	8,218	232	9,405
11	149	0	126	7,959	135	8,792
12	144	205	120	7,199	126	8,380
13	151	0	118	7,564	126	8,887
14	162	3,518	143	9,575	149	9,867
15	83	0	73	7,985	79	8,922
16	115	2,623	85	8,506	92	9,642
17	124	3,890	121	8,254	123	8,387
18	87	4,930	76	7,503	83	8,757
19	186	3,245	162	9,020	167	9,467
20	285	0	251	8,380	263	9,044
21	102	0	75	8,196	80	8,923
22	221	0	182	8,826	194	9,569
23	303	0	269	9,262	278	9,658
24	400	3,460	324	9,613	333	9,875
25	252	4,004	179	9,290	187	9,775
26	106	444	91	8,426	100	9,303
27	246	64	205	8,869	213	9,587
28	193	2,687	142	9,215	153	9,809
29	107	0	91	9,524	100	9,952
30	263	0	233	9,651	252	9,927
31	159	0	122	8,802	131	9,645
32	80	2,526	63	8,457	69	9,359
33	100	0	77	8,439	84	9,508
34	104	14	87	7,140	94	8,774
35	230	0	199	7,639	209	8,641
36	312	28	272	8,968	283	9,724
37	163	39	127	8,252	139	9,186
38	49	0	42	7,942	46	8,775
39	420	3,625	344	9,498	364	9,896
40	206	9	180	9,185	187	9,543
41	147	0	111	8,826	123	9,740
42	135	2,465	96	9,066	106	9,638
43	96	0	83	6,874	89	8,030
44	108	3,030	101	7,595	105	8,414
45	186	0	145	8,167	153	9,044
46	138	2,063	116	7,754	120	8,752
47	139	0	115	8,365	121	9,119
48	192	0	140	9,808	145	9,964
49	77	3,848	59	9,079	66	9,707
50	138	8,268	123	9,616	132	9,768
51	80	0	57	6,792	64	8,296
52	269	62	234	9,479	239	9,765
53	144	1	127	7,485	131	8,278
54	121	0	106	8,147	112	8,900
55	98	8,072	84	9,087	91	9,520
56	130	7,922	108	9,626	116	9,819
57	139	0	112	5,643	118	6,845
58	129	2,853	114	8,070	119	8,683
59	158	1,979	139	7,758	144	8,890
60	100	5,180	85	8,274	90	8,917
61	169	4,218	134	9,044	142	9,591
62	165	8,721	135	9,763	141	9,877
63	279	13	243	5,820	250	7,152
64	241	1,671	218	8,327	222	8,861
65	158	0	135	6,479	141	8,311
66	136	2,003	115	7,170	124	8,400
67	263	2,554	221	8,168	226	9,219
68	234	35	206	6,986	213	8,736
69	165	29	142	7,435	153	8,700
70	229	2,271	195	9,006	200	9,585
71	191	2,370	171	8,706	182	9,601
72	173	1,100	123	8,326	136	9,405
73	268	2,958	243	7,376	250	8,313
74	125	2,116	110	8,341	118	9,441
75	175	28	143	8,324	150	9,487
76	148	1,906	121	9,059	128	9,651
77	232	1,701	175	9,147	204	9,810
78	79	195	72	5,134	74	5,863
79	100	153	84	4,005	89	4,935
80	129	1,939	118	5,656	120	6,080
81	105	130	89	6,402	93	7,320
82	99	17	87	7,016	90	7,854
83	129	426	122	6,227	124	6,828
84	105	1	94	5,597	96	6,122
85	191	5,701	150	9,040	159	9,749
86	80	68	66	7,315	72	8,338
87	149	3	135	8,223	141	8,907
88	231	0	173	9,413	187	9,775
89	257	2,453	206	9,529	221	9,876
90	269	959	174	9,665	190	9,952
	165	1,318	137	8,168	144	8,981

TABLE IV: Number of explored nodes and computational times in seconds for different Sokoban solvers.

#	MPDB				EMM		EMM + MPDB			
	2		4				2		4	
	Nodes	Seconds	Nodes	Seconds	Nodes	Seconds	Nodes	Seconds	Nodes	Seconds
1	319,196	14.41	186,366	42.13	160	0.16	160	0.20	**153**	2.89
2	$>$5,000,000	674.52	$>$729,842	3,600.00	161,834	21.29	86,840	10.96	**68,927**	222.46
3	$>$5,000,000	625.91	$>$701,471	3,600.00	1,187,486	103.74	950,950	97.54	**22,268**	41.89
6	$>$5,000,000	594.04	$>$839,620	3,600.00	1,354,432	152.41	366,955	35.52	**1,641**	6.80
7	$>$5,000,000	567.89	$>$725,610	3,600.00	$>$5,000,000	427.99	223,956	19.62	**127,840**	275.92
17	1,744,788	85.73	1,108,090	334.11	1,471,533	77.38	19,633	1.23	**775**	7.01
38	$>$5,000,000	347.22	1,863,317	1,257.23	93,423	27.01	24,196	1.76	**8,919**	6.20
49	$>$5,000,000	569.64	$>$518,835	3,600.00	1,596,896	173.13	**1,381,596**	147.03	$>$900,425	3,600.00
51	$>$5,000,000	880.18	$>$142,144	3,600.00	$>$5,000,000	905.20	$>$5,000,000	929.33	**223,106**	3,067.88
78	$>$5,000,000	477.79	$>$1,485,294	3,600.00	8,544	1.88	8,387	1.46	**7,646**	39.14
80	$>$5,000,000	412.42	$>$225,445	3,600.00	27,708	38.26	10,594	1.89	**480**	109.12
81	$>$5,000,000	734.27	$>$177,378	3,600.00	$>$5,000,000	1,055.31	$>$5,000,000	1,195.76	**198,935**	2,365.23
83	$>$5,000,000	316.25	$>$1,249,048	3,600.00	559	20.70	362	0.61	**305**	38.54

edge," *Artificial Intelligence*, vol. 129, no. 1–2, pp. 219–251, 2001.

[11] D. Dor and U. Zwick, "Sokoban and other motion planning problems," *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999.

[12] A. Botea, M. Müller, and J. Schaeffer, "Using abstraction for planning in Sokoban," in *Computers and Games*, 2002, pp. 360–375.

[13] J.-N. Demaret, F. V. Lishout, and P. Gribomont, "Hierarchical planning and learning for automatic solving of Sokoban problems," in *Belgium-Netherlands Conference on Artificial Intelligence*, 2008, pp. 57–64.

[14] P. Haslum, A. Botea, M. Helmert, B. Bonet, and S. Koenig, "Domain-independent construction of pattern database heuristics for cost-optimal planning," in *AAAI Conference on Artificial Intelligence*, 2007, pp. 1007–1012.

[15] A. G. Pereira, M. Ritt, and L. S. Buriol, "Finding optimal solutions to Sokoban using instance dependent pattern databases," in *Symposium on Combinatorial Search*, 2013, pp. 141–148.