Sokoban: A Survey of State-Space Search Approaches

Dalal Alharthi
dalharth@uci.edu
ID: 51601658

Robert Logan
rlogan@uci.edu
ID: 87482533

Rahul Sridhar
rsridha2@uci.edu
ID: 41608676

December 6, 2016

Abstract

In this paper, we compare the performance of a collection of state-space search algorithms in finding solutions to Sokoban puzzles. We analyze the performance of Breadth-First search as well as informed search algorithms such as A and Iterative Deepening A*. For informed search approaches, we measure the effect of different heuristic functions as well as the incorporation of domain-specific knowledge on search performance.*

I. Introduction

Sokoban is a classic computer game first developed in the 1980s. The game consists of a *single agent* residing on a grid of tiles, whose goal is to push boxes to a set of predefined storage locations. The environment is *fully observable* to the player at all times, and there are only a finite number of possible states for a given puzzle (hence the states are *discrete*). The actions available to player at any given point in the game are to move up, down, left, or right to an adjacent tile. If the adjacent tile is empty or a storage location, the player is always able to move into it. If the adjacent tile is a box and the tile behind the box is empty or a storage location, then the player can move into the adjacent tile, pushing the box into the tile behind it. If the adjacent tile is a wall or a box which cannot be pushed then the player may not move into it. Accordingly, the above rules demonstrate that these actions are *deterministic*. A solution to the problem consists of a sequence of moves such that when the final move is performed, all of the boxes are in storage locations.

Sokoban is an interesting game to AI researchers because it is PSPACE-complete, and is considerably more difficult than similar problems such as the sliding box puzzle due to its large branching factor and search tree depth [1]. Previous work on the problem has established that state-space search algorithms tend to perform very poorly out of the box; the most successful solvers rely heavily on incorporating domain specific knowledge to help reduce the size of the state space [2]. For example, it is possible for boxes to be pushed into locations which prevent them from ever being moved again - states containing such a configuration are known as *deadlock* states. Once a deadlock state is encountered during search, subsequent branches can be pruned from the search tree since it is impossible to reach a solution. In this paper we will analyze the performance of different state-space search algorithms in solving Sokoban, as well as demonstrate the effectiveness of including domain-specific knowledge such as the deadlock check mentioned above.

II. Methodology

Problem Representation

In our implementation, we chose to represent the game's *state* as a tuple of player coordinates, along with a 2D array of integers called `tiles` which describes the elements contained in a certain location

in the game board. For example, `tiles[0][0]=1` indicates that there is a wall at coordinate $(0, 0)$ on the board, while `tiles[1][2]=2` indicates that there is a box at coordinate $(1, 2)$. While this representation of a state consumes more memory than storing all of the coordinates of the different board elements in lists, it allows the elements in a particular location to be looked up or changed in constant time.

As described in the introduction, there are 4 potential *actions* available to the player in a given state - they can move up, down, left or right. In our implementation, we consider any move that is blocked (such as moving into a wall) to be illegal when expanding the current space. A resulting state is produced by the *transition function* by altering the player coordinates, and potentially the `tiles` array if a box was moved. For example, if a player located at $(1, 1)$ moves right into an adjacent tile containing a box (i.e. `tiles[2][1]=2`), then the player coordinates in the resulting state will be $(2, 1)$ and the tiles array will be changed so that `tiles[2][1]=0` (an empty tile) and `tiles[3][1]=2` (a box). We check that a state passes the *goal test* by checking that each storage location contains a box. Since storage locations do not depend on the state, we store the coordinates of the storage locations in a list when initializing the problem so that they can be easily examined during search.

Search Algorithms

In our experiments, we employed the following search algorithms:

- Breadth-first search
- A* search
- Iterative deepening A* (IDA*) search

To prevent A* and IDA* from exploring sub-optimal paths, their implementations were modified to include hash tables of explored states and the lowest value of the evaluation function encountered for those states . This incurs an additional computational cost in each iteration of the algorithm, but the time gained by the resultant pruning of the search space outweighs the time spent in maintaining the hash tables. Also, given that the heuristics we used are consistent, the optimality of the solution found won't be affected.

Distance Based Heuristics

There are quite a few heuristics we came up with which can be applied to this problem. The most obvious and naive is to sum up the minimum Manhattan distances between boxes and storage locations, which relaxes every restriction in the problem except that a solution consists of boxes in storage locations. The main benefit of this heuristic is that its value for a box at a given location is completely independent of the location of the player or other boxes - so we can compute upfront the value for every location on the board at the beginning of the search and store these in an array so that they can be looked up in constant time. In addition, this heuristic is easily shown to be *consistent* and hence guarantees that A* and IDA* will always produce an optimal solution. Consistency follows from the fact that the only situation where the heuristic value decreases is when a block is pushed closer to its closest goal, in which case it is reduced by one which is equal to the cost incurred of performing this move. Hence $h(n) = h(n') + c(n, n')$. In all other cases, the heuristic value will either increase or stay the same, so $h(n) < h(n') + c(n, n')$.

A simple improvement to the Manhattan distance heuristic can be made by including back the restriction that boxes cannot pass through walls when they are being moved to goals. We will refer to this as the *wavefront heuristic*. Similar to the Manhattan distance metric, this heuristic's values depend only on the static elements of the game (i.e. walls and goals) and so it can also be computed

exactly once for each goal location at the beginning of the program and stored in an array for later use. The distance values are computed using the wavefront algorithm, whose runtime is linear in the size of the game board. To evaluate the heuristic for a given state, one needs only to sum up the values in the array corresponding to the boxes positions on the board, which can be done in constant time. As was the case with the Manhattan distance heuristic, the wavefront heuristic also turns out to be *consistent*. The proof of this statement follows the exact same argument that was presented above.

As it turns out, both of the heuristics presented so far have limited application due to one serious flaw - they do not enforce the restriction that a storage location can only hold one box in the final solution. This is a major issue since it can cause algorithms to explore states where there are way too many boxes concentrated around a few storage locations. To counteract this we need to find some way of assigning boxes to goals, preferably in a way that minimizes the cost of reaching those goals.

We attempted to address this problem by computing the minimum cost assignment from boxes to goals where the entries of the cost matrix are the Manhattan distance values, we call this the *goal priority* or *minimum cost matching heuristic*. For small numbers of boxes and goals we use an algorithm which performs the first two steps of the Hungarian algorithm which introduces zeros to the cost matrix followed by Branch and Bound algorithm. Performing the first two steps of Hungarian algorithm is a minor optimization of B&B because it lowers the variance between the different assignment sums, which will make all the lower bounds closer to each other. For large number of boxes, it became too slow to use. The full Hungarian algorithm might work better in this case.

Deadlock Detection Heuristics

While the distance based heuristics discussed so far are useful in that they help the search prioritize moving blocks towards goals, they can be incredibly short sighted in the sense that they spend large amounts of time exploring parts of the search space which cannot possibly lead to a solution. A specific case where this happens is when one or more blocks become immovable, which is referred to in the Sokoban literature as a *deadlock* state. For example, a deadlock state occurs when a box is pushed into a corner which does not contain a goal. Significant gains to performance can be made by detecting when the search has encountered such a deadlock state and pruning all subsequent nodes from the search tree. In the context of heuristic search, deadlock detection algorithms can be viewed as a form of heuristic function which takes on a value of infinity when evaluated in a deadlock state.

There are two classes of deadlocks we seek to identify in this paper. The first is the class of *simple deadlocks*, which occur when a single box becomes immovable - such as in the corner example given above. Since these deadlocks do not depend on the positions of other movable game elements such as the player or other boxes, they can be readily identified at the initialization of the problem. A simple algorithm for doing so works as follows. Remove all of the boxes from the game board. Place a single box on a given goal. Iteratively determine all of the tiles which the box can be pulled to from this goal and mark them as non-deadlock tiles (note: since pulling and pushing are opposite maneuvers these are precisely the sets of tiles from which a box can be pushed to the goal). Do this for all goals. Mark all remaining unmarked tiles as deadlock tiles. This algorithm can be made slightly more efficient by foregoing the pulling procedure for goals which are already marked as non-deadlock tiles. Now whenever an action would cause a block to be pushed into a deadlock tile we consider it to be invalid.

The second class of deadlocks we consider are called *freeze deadlocks*. These occur when a cluster of boxes is configured in such a way that they all block each other from moving. Since freeze deadlocks depend on the location of other boxes, they must be checked for whenever an action would cause a block to move. A simple algorithm for detecting freeze deadlocks works as follows.

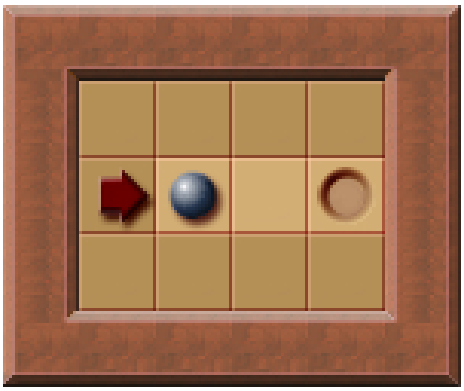


Figure 1: *Shaded squares are simple deadlocks [3]*

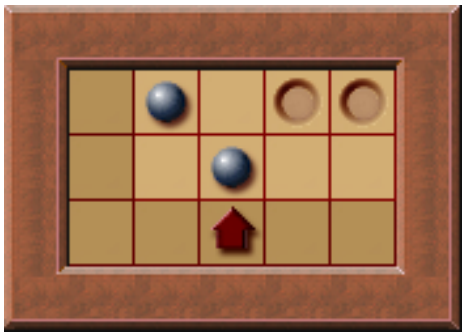


Figure 2: *Performing indicated push will cause a freeze deadlock [3]*

Whenever attempting to push a block, consider what the resulting state looks like. We say that the pushed box is blocked horizontally if it is bordered by either a wall or two deadlock tiles on its right and left. Similarly, we say that the pushed box is blocked vertically if it is bordered by a wall or two deadlock tiles from above and below. If the pushed box is blocked both vertically and horizontally then there is a freeze deadlock, if the box is not blocked in either direction then there is no freeze deadlock. If the box is blocked in a single direction, and bordered by another box in the unblocked direction, we must then recursively check whether that bordering box is blocked in both directions. To prevent these checks recurring infinitely, we temporarily consider the already examined block to be a wall. Also note that the notion of a freeze deadlock is completely useless if the blocks are located over storage locations, since they could very well be a part of the solution - in this case we say the freeze deadlock check fails even if the described algorithm finds one.

III. Experimental Results

Given all of the well motivated heuristics we have just discussed, we would like to know how much of an impact they have on the performance of our search algorithms. Specifically we are interested in how well they prune the search space as well as their impact on empirical runtimes since many of the heuristics discussed require a substantial amount of overhead. For our experiments, we chose to run the following algorithms:

1. BFS
2. BFS w/ Deadlock Detection
3. A* w/ Deadlock Detection and Manhattan Distance Heuristic
4. A* w/ Deadlock Detection and Goal Priority Heuristic
5. IDA* w/ Deadlock Detection and Manhattan Distance Heuristic

Given the complexity of the Sokoban problem and the very few algorithms in popular literature that have been successful on highly difficult puzzles, we felt that the relatively simpler algorithms and heuristics we have formulated would not perform efficiently on highly difficult puzzles. We hence spent most of the time designing our search algorithm to optimize solving the first 99 problems from the microban suite available at *http://www.onlinespiele-sammlung.de/sokoban/sokobangames/skinner/*, the difficulty of which varies from easy to hard. The reader can readily observe that these problems

are much simpler than the course benchmark problems, which heavily skewed our perspective of which algorithms were most efficient.

The first thing we sought to establish in our experiments was the effectiveness of deadlock detection heuristics. To do this, we compared the performance of uninformed breadth-first search, to breadth-first search with deadlock pruning (note: this can be seen as a form of A* search where the heuristic function is 0 if a state is not a deadlock and infinity if a state is a deadlock). Averaging across the microban test suite, we saw that BFS typically took about 7.8 seconds to solve a problem and expanded around 158,000 nodes. When deadlock detection was included, the algorithm took 1.3 seconds on average to find a solution and expanded around 24,000 nodes. Thus, the inclusion of deadlock detection increased performance by a factor of 6-7, which was the most significant boost to efficiency out of all of the heuristics we tried. For this reason, we chose to incorporate deadlock detection into all of the other algorithms we investigated.

The next thing we sought to establish was the effect of distance based heuristics on performance. To do this we compared the performance of A* using the relatively simple Manhattan distance heuristic and the more complicated goal priority heuristic. Results here were much less inspiring. Across the microban test suite, we saw that A* with the Manhattan distance took on average 6.2 seconds to solve a problem, whereas A* with the goal priority heuristic took an average of 6.8 seconds. A* with the other heuristic however was more successful at pruning states from the search, expanding roughly 23,000 nodes per problem as compared to the 24,000 expanded by A* with the Manhattan distance heuristic. These results led us to believe that there is very little benefit to including distance based heuristics, since both methods significantly increased the runtime of the search relative to BFS w/ deadlock detection, while barely affecting the number of nodes pruned.

The final algorithm we tested was iterative deepening A* w/ the Manhattan distance heuristic. This was largely due to the popularity of IDA* in the Sokoban literature. Results here were also not great, as the algorithm took a whopping 20.8 seconds on average to solve a problem and typically expanded around 121,000 nodes. This of course was expected since IDA* repeats the search each time the depth-bound is increased, thus it tends to repeat a lot of work. The main conclusion we drew from this was that memory is not as much of a constraint today as it was in the past - the low memory consumption of IDA* did not seem necessary as our solver had very few memory issues on the problems in the microban suite.

Accordingly, we were in for quite a surprise when we saw the benchmark problems for the course which involved boards much larger than what we had come across in the microban test suite, and containing an order of magnitude more goals. Unfortunately, BFS with deadlock checking, the solver we had deemed to be most efficient according to the research above, performed very poorly on these as is demonstrated in the table in the appendix. Our algorithm was only able to solve the first two benchmark problems in a reasonable amount of time, and ran out of memory for all subsequent problems.

IV. Conclusions

In summary, it is clear that there is still quite a bit more work that could be done to improve our solver. One area worth investigating is whether using a more compact state representation such as lists of the coordinates of the boxes and player would help conserve memory. We also expect gains to efficiency could be made by implementing more of the techniques we came across in our research such as more sophisticated forms of deadlock detection (such as detecting corral deadlocks), macro moves for pushing blocks to goals and moving through tunnels, as well as techniques which decompose levels into separate subproblems. Given that the latter benchmark problems were composed of what appeared to be separate rooms we expect that subproblem decomposition would have been particularly helpful.

Having said that, the research presented in this paper still has merit regarding the topic of solving

smaller Sokoban problems. We have shown that when memory is not a constraining factor, simpler heuristics and algorithms tend to have better performance characteristics. Distance based heuristics in general appear to be relatively expensive to compute and do not effectively prune the search tree. Meanwhile, domain specific heuristics such as deadlock detection are incredibly effective. This suggests that algorithms which perform well on Sokoban will probably not generalize well to solving other problems. Overall, this problem proves to be extremely interesting and challenging from an AI perspective as it provides a good example of how difficult it is to make state-space search algorithms efficient when applied to problems with large search spaces.

V. Appendix

		P0	P1	P2	P3	P4	P5
BFS	Nodes Generated	1	58,807	-	-	-	-
	Runtime	81 ms	23 s	-	-	-	-
BFS w/ Deadlock	Nodes Generated	1	56,489	-	-	-	-
	Runtime	74 ms	2 s	-	-	-	-
A* w/ Manhattan	Nodes Generated	1	56,372	-	-	-	-
	Runtime	157 ms	11 s	-	-	-	-
A* w/ Goal Priority	Nodes Generated	1	49.861	-	-	-	-
	Runtime	244 ms	12 s	-	-	-	-
IDA* w/ Manhattan	Nodes Generated	1	-	-	-	-	-
	Runtime	295 ms	-	-	-	-	-

Table 1: *Course Benchmark Results*

Problem Solutions

Sokoban0.txt :

Number of moves to the goal = 1

Path to goal = D

Sokoban1.txt :

Number of moves to the goal = 80

Path to goal = U L L L D L L U R U U L U U R D D U R R U R R D L L L D L D D L D R U R R R U U D D D R U L L L L U U L U U R D R R U R R D L L L L D L U R R R R D D D L L L U

	Avg. Nodes Generated	Avg. Runtime
BFS	158k	7.8 s
BFS w/ Deadlock	24k	1.3 s
A* w/ Manhattan	24k	6.2 s
A* w/ ???	23k	6.8 s
IDA* w/ Manhattan	122k	20.9 s

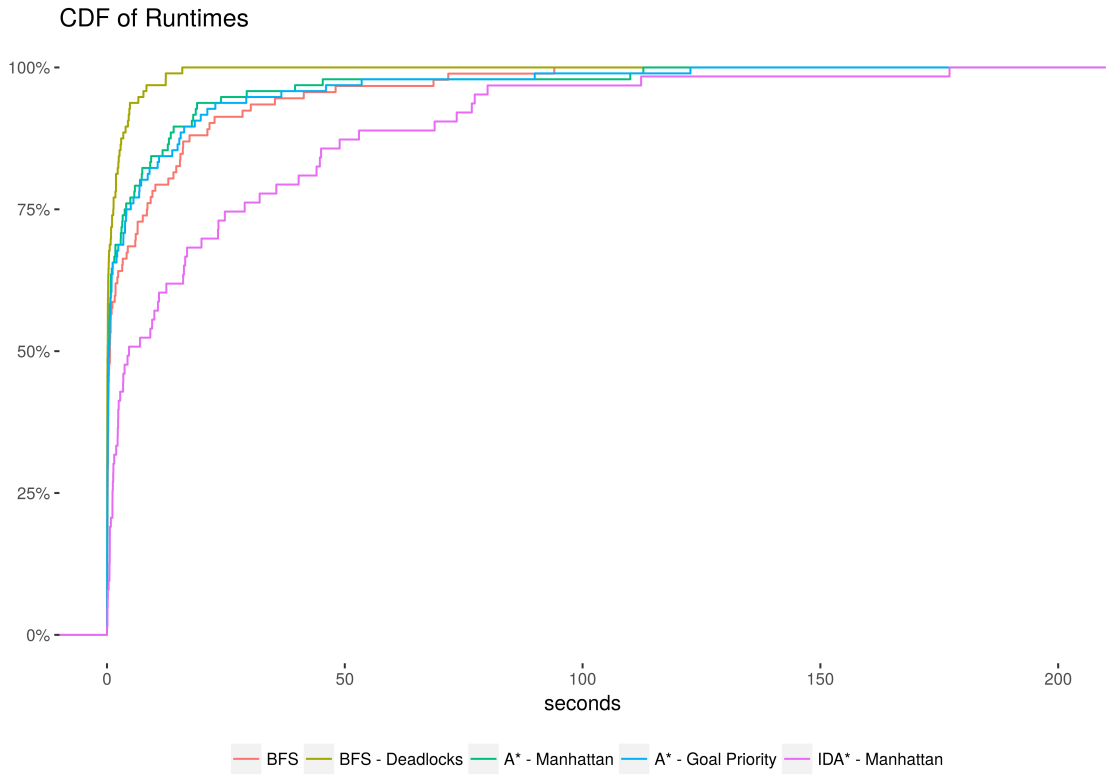
Table 2: *Microban Benchmark Results*

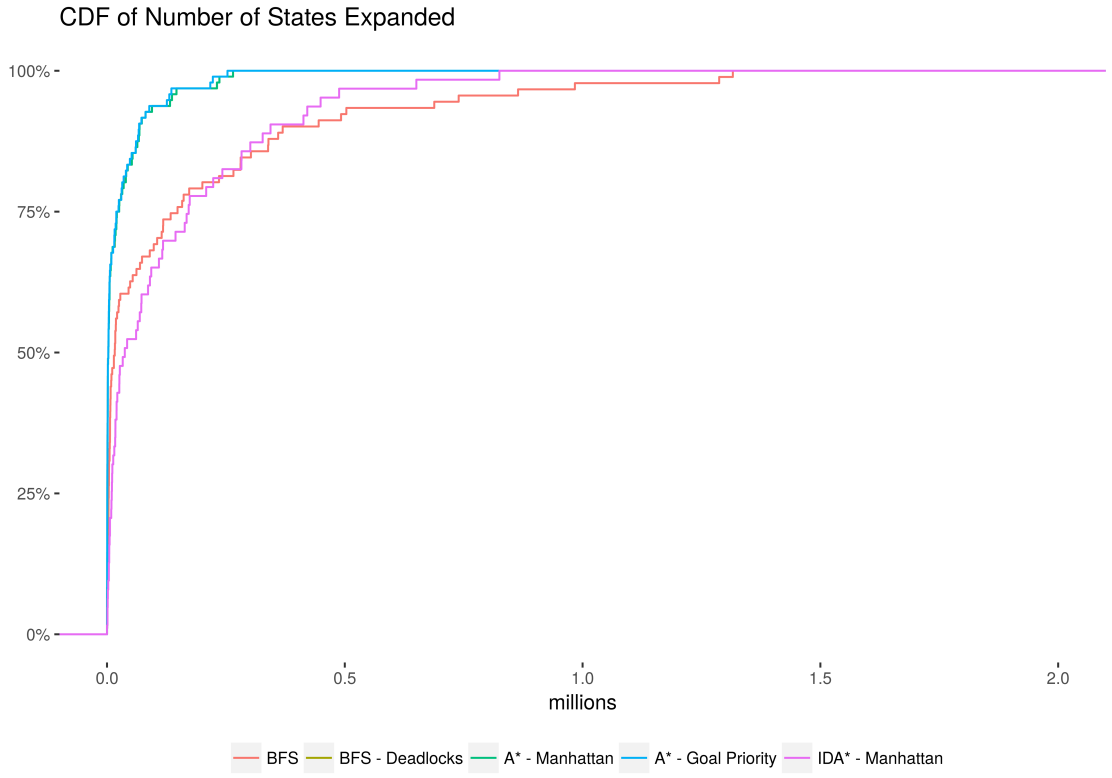
References

[1] Sokoban Is Pspace-complete, Joseph C. Culberson, and Joseph C. Culberson. Sokoban is pspace-complete, 1997.

[2] Andreas Junghanns and Jonathan Schaeffer. Sokoban: A challenging single-agent search problem. In *In IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch*, pages 27–36. Universiteit, 1997.

[3] Sokoban wiki: Deadlock detection in sokoban.


CDF of Runtimes
100%
75%
50%
25%
0%
0
50
100
150
200
seconds
BFS
BFS - Deadlocks
A* - Manhattan
A* - Goal Priority
IDA* - Manhattan


CDF of Number of States Expanded
100%
75%
50%
25%
0%
0.0
0.5
1.0
1.5
2.0
millions
BFS
BFS - Deadlocks
A* - Manhattan
A* - Goal Priority
IDA* - Manhattan