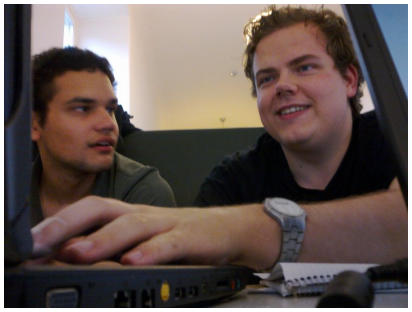
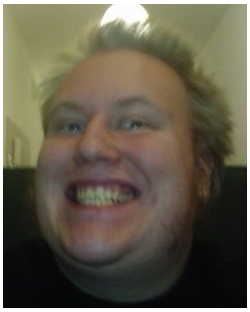
We Called Him Jens

The Solver of Sokoban

JOHANS HJÄLTAR
TIM LENNERYD, RISHIE SHARMA, JACOB NORDGREN, JOHAN ÖHLIN

倉庫番

Abstract

The game of Sokoban has been said to be PSPACE-complete, and it is still possible to write a fairly effective solver for this using several artificial intelligence techniques and heuristics. It was therefore a task given to us as a project for our course in just artificial intelligence.

For us, the task became yet a bit more challenging, since we decided to use the programming language C++, a language we are not too familiar with. But it turns out there are only a few good ways of finding solutions for a majority of the game boards given to us. The A* algorithm together with some good heuristics, such as not putting boxes in the corners from which they can not be moved again, and a forward- and backward search ought to give a good solution.

We did not have the time to implement all of these heuristics but our solver, the one we call Jens, manages to solve a reasonable amount of boards anyway.

Contents

Chapter 1

Background

Sokoban (倉庫番 in Japanese) is a type of transport puzzle originally from Japan that involves pushing boxes into their goal positions on a maze-like board. It was first created in 1981 and has since been used both as a computer game and a testbed for artificial intelligence.

While there have been a number of different implementations and variations of the game, the basics of it remains the same. A character is the same size as the boxes, and the boxes can only be pushed from the four different directions; up, down, left and right. The goal is to get every box into a goal.

If the player is to push the box to the left, he needs to stand to the right of the box. That is, the boxes can not be pulled or pushed diagonally as you can see in Figure 1.1.

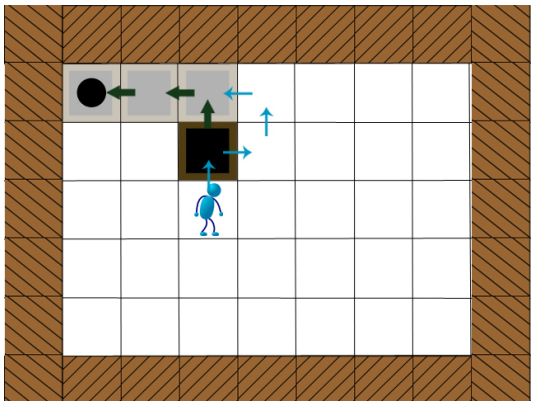


Figure 1.1. Sokoban board with arrows on how to push boxes. Blue arrows are the players moves, the black ones are the box moves.

For a more extensive explanation of the different figures, see Appendix A.

Chapter 2

Theory

2.1 Complexity

The game of Sokoban belongs with a number of other games and puzzles in a family called Motion Planning Problems, or more specifically Motion Planning Problems with Movable Obstacles. The time and space complexity of the games and puzzles in this family vary, some being in P, some being NP. Sokoban however have been shown by for example Joseph C. Culberson to be PSPACE-complete [1] . He has done this by showing that a Turing Machine can be emulated in linear time using a specially constructed infinite version of the puzzle. In this version, there are only a finite number of out-of-storage containers, and he uses a number of special constructions of the puzzle to emulate different parts of the Turing Machine. He then restricts the tape to finite length, which in turn shows that the puzzle is PSPACE-hard. We will not go into this further, since the reasoning for this are fairly advanced. We refer to the report by Joseph C. Culberson instead, which can be found in the references [1].

Dorit Dor and Uri Zwick has concluded that the Sokoban puzzle is in fact NP-hard in terms of time complexity.[2] They use a strategy similar to Culberson's proof above, where they make special constructs within the rules of the game to make the game behave a certain way. This is to be able to reduce the problem to a known NP-hard problem, which in the case of their proof is P3SAT, i.e. the Planar 3SAT problem which is a variation of the original 3SAT problem. They do this first for a generalized version of the game, in where the pusher can push k number of blocks, and pull n blocks all at once. In the case of their first proof, $k = 5$ and $n = 1$. By being able to reduce this constructed Sokoban game to 3SAT they are then able to use this as a grounds to prove that the Sokoban game we are considering here is also NP-hard. This is done by making some changes to the proof and using $k = 1$ and $n = 0$ which then means that the pusher can push one block at the time, and it is not possible to pull boxes. These proofs are however quite advanced, and if more information is wished for, the article used is in the references.[2]

2.2 Unique states and search space

The Sokoban game has a very large search space as well as a very big branching factor, as the complexity analysis referenced above shows. This can be easily shown just by thinking of how one may define a single state. To be able to provide a unique state, as a basis for eliminating repeated states in the solver, all the box positions need to be included. On top of that, the position of the pusher needs to be included as well, since the boxes may need to be pushed from a certain direction for the successive states not to result in a deadlock. If the pusher can get from his current position to that direction without touching a box, that is enough to provide a unique state in terms of the theory.[3] This is a fairly complex thing to keep track of, and it is usually just easier to keep track of the specific position of the pusher as well as the boxes, and call this a unique state.

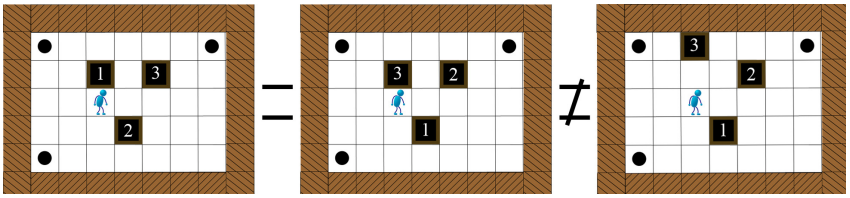


Figure 2.1. Leftmost image shows a state, middle image shows that same state with a different configuration of the boxes, and right shows a unique, but similar state.

This means that the branching factor of a specific board depends on the number of boxes, since every box can be in every position in theory, and also on the number of directions the player can move (which is at most four in Sokoban). A normal game with for example six boxes may then have a branching factor of $4 \cdot 6 = 24$, since every box has at most four possible legal moves depending on the direction it is pushed. This means the search space will be huge, since in level k of the search space tree, there will be 24k nodes.[3]

2.3 Finding solutions

Due to the perfect information single-player nature of Sokoban, an informed search algorithm that is proven to be optimal in such situations can be used, despite the search tree being very big. One example of such an algorithm is A*, an algorithm using best-first search and that will find the least-cost path from an initial node to a goal node. The initial node in this case is the untouched box-positions and the player position, and the goal nodes are the nodes where all the boxes are placed in the goals, regardless of the permutation among the boxes and the position of the player. A* is an extension of the Dijkstra algorithm that achieves better performance by using certain heuristics, and can still be proven to be optimal if certain constraints to the heuristics used are met.[4]

A* considers both the cost to get to a particular node in the tree from the root (the initial state node), as well as the estimated cost to get from the current node to the goal node. Due to the fact that the cost to get from current to goal is only estimated, heuristics must be used to do the estimation, which in turn places certain constraints on the heuristics, as mentioned above. The main constraint is that the heuristic (or combination of heuristics) may not overestimate the cost to get from the node to the goal. This is an important, since this means that the A* algorithm will never overlook the possibility of a lower-cost path. The algorithm is therefore optimal in the sense that it will find the least-cost path in terms of the costs placed upon the node transitions by either the heuristic or the problem itself.

That being said, this does not mean that A* is the best algorithm to use out of the box for Sokoban. Because of the nature of the game the search tree can be pruned in a number of ways by both simple heuristics and quite advanced ones, by finding deadlocks in different ways. This is discussed in more detail below. Many of the heuristics that can be implemented to find deadlocks can however be used with both A* as well as in other search algorithms, since they mostly just prune the tree in different ways. A* is, however, a good choice even after all these heuristics were implemented, since A* is a search algorithm that has very good performance provided a good (more global) heuristic can be found to provide it with estimates. [4] Using a simple Move-Toward-Box and Push-Box-Toward-Goal heuristic could give A* a big edge over algorithms like depth-first and regular greedy-best-first, since greedy-best-first does not consider the distance already travelled.

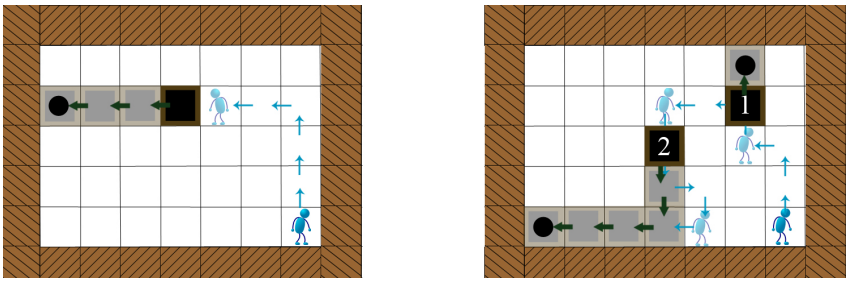


Figure 2.2. Move-To-Box heuristic together with a simple heuristic to push a box toward a goal. Makes no allowances for the possibility to actually reach the goal, but works for simple cases as above.

2.4 Finding deadlocks

A deadlock in this game is a state in the game from which it is impossible to win. In the most obvious case, this is when a box is pushed into a corner and therefore cannot be pushed from any meaningful direction. But deadlocks can be much more subtle than that, containing a number of walls in different configurations as well as boxes. As soon as a deadlock is found, that part of the search tree ought to

be pruned, which means that every single deadlock found will help in the effort of reducing the size of the tree.

Let us do an example. If there are two boxes on a 10x10 board, then each of the boxes can be in one of a 81 different positions (they can not be in the wall). The total number of positions for the two boxes are then $81 \cdot 80 = 6480$. Add to that the pusher, who can be in any one of the free positions, and we have 511 920 unique states. Now, this is a completely constructed example, where the whole 10x10 board is open with no walls except around the edge. If we only consider the 4 corners to be deadlocks, this means that we have $77 \cdot 76 = 5852$ positions for the boxes, and 79 for the pusher, to a total of 462 308 unique states. Due to removing the 4 deadlocks we had considered, we have removed 511 920 - 462 308 = 49 612 states.

In theory the tree could be reduced to only having valid, even very long in some cases, solutions, if all the deadlocks were to be found. Doing so would, however, solve the board and thus it has the same difficulty. Therefore, this is not reasonable. The most naive and intuitive solution for finding deadlocks is to sit with a pen and paper and write down all the different deadlocks that one can think of. Be it corners, two blocks beside each other pushed up to a wall etc., but only a fraction of the deadlocks possible can be found in this way. With that said, every single deadlock pruned away helps when it comes to finding a solution, but there are other ways. If one uses a pattern based method, deadlocks can be saved as patterns to a database or table beforehand, which will allow for offline calculation of said deadlocks in for example a brute force manner.

Using brute force prevents larger patterns due to the time complexity of brute force calculations. Smaller patterns can then be saved to a database and would then allow for a simple look-up where the positions of the walls, boxes, goals and the pusher within the area are being considered, and returns whether the position results in deadlock or not. Size-wise these saved locks should be at least 2x2, but the upper limit depends on the time and memory one can spare.

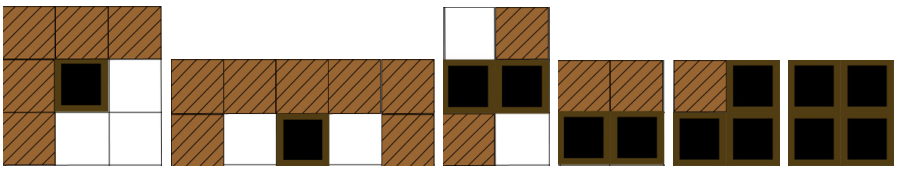


Figure 2.3. There are many different types of deadlocks, but also permutations of the same type of deadlock.

By using the above ideas, we decided to go through the board before we start the search for a solution. We managed to mark all the spots where boxes would not be able to reach a goal from, by applying a number of conditions to a series of floor tiles. If all those conditions were met, the tiles were all illegal in terms of box placements. Figure 5 below shows the conditions and the resulting markings on the board.

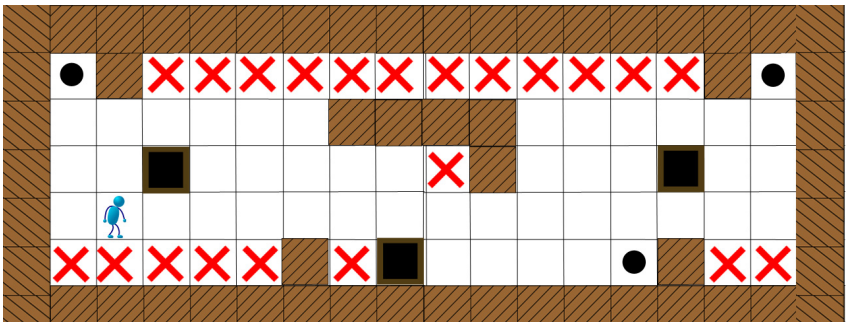


Figure 2.4. Unwanted positions for any box are marked with X. These positions are not considered valid by the application as they will result in deadlock situations.

Chapter 3

Implementation

3.1 Algorithm Basics

The fundamental part of the algorithm is a depth first search that from each node, loops through the possible directions and chooses a direction based on two different criteria. The first criteria that is checked is whether it is possible to move in the direction at all. This is done by enforcing the rules that are required for the move to be possible without violating the rules of Sokoban. This criteria also checks that the exact same position have not been evaluated by the solver previously to make sure that any repeated states are removed.

The second criteria evaluated is the cost to move in the requested direction. This is done by applying a series of heuristics and local optimizations to give a cost based on the attractiveness of a certain position. As an example, pushing a box into a corner is a very unattractive move as every move after that will be a dead end, the box can no longer be moved out of the corner and therefore the game can not be won from that state.

3.2 Architecture

We have tried out a few different architectures and data structures for our application. Early on, we decided that we wanted some type of hash to provide us with a quick (preferably constant time) way to tell us whether we have been in the specific state before. As mentioned above, we consider a specific placement of the boxes as well as the pusher to be a unique state. While this is not optimal, as mentioned by Junghanns and Schaeffer in their article[3], it is an easy definition to implement.

The data structures we have tried for holding the states have been the Boost C++ library versions of a Bloom filter and a hash-map. While the Bloom filter was the most effective in terms of time and space, it had problems when there were collisions, since then a path of the tree was pruned when it was not meant to. After some testing we noticed the Bloom filter had a collision rate at 0.9% with the setup we had, which was too great of a value. We then decided to skip it in favour of

a hash-set to be able to rule out collisions with some extra logic. This was slower overall, and it did not give us the constant time and the constant (if rather large) space complexity that Bloom filter provides, since a hash-map saves the whole node, and needs to enlarge itself at times.

Chapter 4

Result and performance

During the project we now and then made ran through all 136 boards from the server. Some of the runs check the solution against the server to see if it is correct, and some do not and just tells us that it has finished the solution within the time limit. We made a script that ran through the boards and killed it after one minute, since that is the time limit we would have against the server. These runs were not made on regular intervals, and became more frequent at the end. You will find the results in Table 4.1. Note that the server changed the boards over the time period, but they are approximately the same.

Run	# solved boards	Date
1	6	24 September 2010
2	5	24 September 2010
3	6	25 September 2010
4	27	28 September 2010
5	29	12 October 2010
6	38	12 October 2010
7	38	14 October 2010
8	41	14 October 2010
9	50	15 October 2010
10	49	15 October 2010

Table 4.1. Table with results collected in run files using a bash script that kills after the one minute time limit. These files can be found on our SVN server as RUNX.txt[5].

At the beginning of our development, the only thing we had implemented was a DFS, together with a check that Jens did not end up in previous states by way of using a bloom filter. We did however note big problems with our Bloom filter, in that we had collisions which meant we missed parts of the search trees.

After several tests during run 2 and 3, we decided we could not solve the problems we had with the Bloom Filter and went for a more conventional method by using a hash set instead. During this time frame we also experimented with letting the pusher walk in random directions instead of a set order, and the result of that can be seen between run 2 and 3 as well. We also added our first deadlock detection around this time, that only detected corners. This mostly improved on the number of iterations we had to do, rather than the number of solved boards. The third board was for example solved in 1 406 iterations at our 3rd run, and in run 4 only 323 iterations were required. This seems to be a very good performance improvement, and it is, if one were to stay solely with the DFS solution as the major algorithm.

All of this was later scrapped in favour of a heuristic that chose the direction based on the closest box. This, together with some refactoring of the whole code and detection of some simple deadlocks made big improvements in the number of solved boards, as can be seen in run 4, which solves 27 boards.

Around this time we noted our memory issues as well as started to profile our code a bit. This made us commit big efforts to improve the hash functions we used. This allowed us to increase the maximum iteration limit for the program a fair bit. We also implemented a more advanced deadlock detection of locked boxes with hard-coded 3x3 blocks that we then checked the pushed box against. These improvements made bumped our board number up to 38 for run 6. Just as a note, around this time the boards from the server changed, so we can not make exact comparisons between earlier runs.

After the optimizations above we put our effort into implementing the A* algorithm, with a more advanced version of the Move-Box-To-Goal heuristic we spoke of earlier. Since A* requires some sort of heuristic to estimate the cost of getting from the current node to the goal, we calculated the distance between the pusher's current position to the closest box, and from the closest box to the closest goal. Then, taking the goal position, we again find the closest box and calculated the distance to the now closest goal (excluding the previous box and goal) . This allowed us to calculate a cost based on the distance between the boxes, the distance between a box and its ideal goal, as well as the pusher's distance from any box. This resulted in 50 solved boards in run 9.

After having implemented A*, we spent some time trying to implement a bi-directional search. Due to the architecture of our code, this was not trivial, and we were also low on time. It also turned out to eat even more memory, so it was not feasible to use that code. We now have a functioning bi-directional search, but we have not been able to fit it together with our working A* solution in a way that solves more boards for us.

It should also be noted that any variation could also be cause of differences in

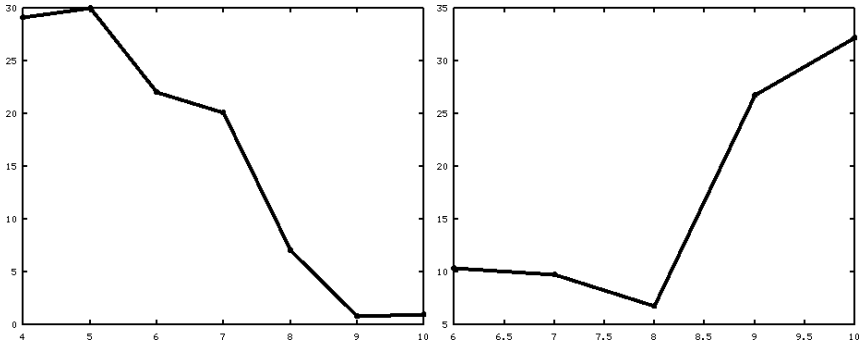


Figure 4.1. How the run time varied on board 44 (left) and board 50 (right) over time. Note it is not the same scale on the two axes.

memory allocations and running processes. Which could make the program run somewhat slower on some boards.

Bibliography

[1] J. C. Culberso, "Sokoban is PSPACE-complete," 1997. Found on `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.41`.

[2] D. Dor and U. Zwick, "Sokoban and other motion planning problems (extended abstract)," 1995. Found on `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.585`.

[3] A. Junghanns and J. Schaeffer, "Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock," in *Advances in Arti Intelligence*, pp. 1–15, Springer Verlag, 1998. Found on `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.6096`.

[4] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson, third ed., 2009.

[5] T. Lenneryd, J. Nordgren, R. Sharma, and J. Öhlin, 2010. Our SVN server: `http://korvmedmos.googlecode.com/`.

Appendix A

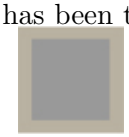
Figure explanations

Here follows explanations for the images used to describe boards.

 - The pushers initial position.

 - A temporary stop for the pusher. Used to indicate pushes on a box from a certain direction.

- A numbered or unnumbered box, the initial position of the box before it has been touched.

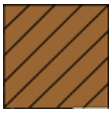
 - Temporary position of a box. Used to indicate position after the box has been pushed somewhere.

 - Arrow showing the movement of the pusher as he is moving around the board.

 - Arrow showing the movement of a box as it is being pushed.

 - A goal. Stays still, may be covered by a temporary box position.

 - A position resulting in a deadlock if a box is pushed to it. These positions are not considered valid by the solver.

 - Wall. Does not move. No other function than to block the path.