Sokoban: Reversed Solving

Frank Takes (ftakes@liacs.nl)
Leiden Institute of Advanced Computer Science (LIACS), Leiden University

June 20, 2008

Abstract

This article describes a new method for attempting to solve Sokoban puzzles by means of an efficient algorithm, a task which has proven to be extremely difficult because of both the huge search tree depth and the large branching factor. We present a way of solving Sokoban puzzles that, using several heuristics, starts from the final state of a puzzle, and from there works its way back to the initial state. This method makes the time-consuming checking for a large portion of the undesired deadlocks unnecessary, giving some interesting results.

1 Introduction

We will start with a description of the game of *Sokoban* and the obstacles that arise when attempting to solve a Sokoban puzzle by means of an efficient algorithm. We will then discuss several solving methods, and finally present a new way of solving these puzzles, eliminating some of the discussed obstacles. The basic idea behind this method is to “reverse” the puzzle: we work back from the final state to the initial state of the puzzle. This article ends with some results of our solving method and several possible future challenges when it comes to fine-tuning our solving method.

This paper is a condensed version of the Bachelor Thesis [7].

2 Sokoban

Sokoban is a single player game that was created around 1980 in Japan. Sokoban is Japanese for “warehouse keeper”, which is a pretty straightforward name judging from the fact that the goal of Sokoban is to push boxes around in a room with obstacles. Other than being a funny game, Sokoban has been an object of study for those in the field of Computer Science and Artificial Intelligence for quite some time. The reason for this interest comes from the fact that humans can often solve these puzzles in a few minutes doing several hundreds of moves. However, solving a Sokoban puzzle by means of an efficient algorithm has turned out to be very hard, because of both the huge search tree depth and the large branching factor.

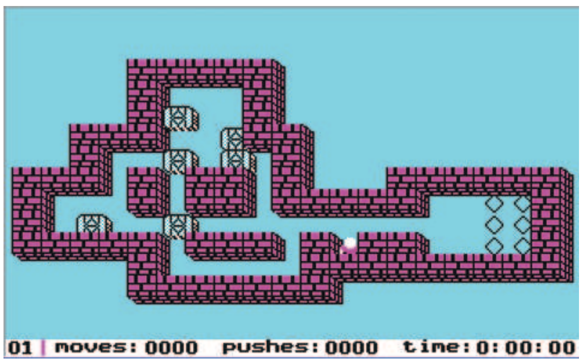

Figure 1: Puzzle 1 from the original set.

2.1 The game

Sokoban has relatively simple rules. The game is played on a two-dimensional field, usually of size 20×20 or smaller. We will describe the field’s cells (squares) with coordinates (x, y), where the top left corner corresponds to $(0, 0)$. A cell contains one of the following things: an empty square, a target square, a wall, a box, the man, the man on a target square or a placed box (combination of a target square and a box). The amount of boxes is always equal to the amount of target squares. The man, of which there is only one, can move in four directions (traditionally up, down, left or right), and he can only move to target squares and empty squares. Additionally, the man has the ability to push one box at a time. Formally, pushing a box from position (x, y) while the man is standing at position $(x - 1, y)$ is only allowed if position $(x+1, y)$ is either empty or a target square. The same of course applies for the y-direction. As one may have guessed, the man cannot be moved through walls, neither can the boxes. Usually the playing field is surrounded by walls, so that we will always be bounded by walls and cannot reach the edge of the field. In each puzzle the man starts at a certain fixed position. Traditionally, the goal of Sokoban is to use the man to push all of the the boxes onto the target squares. While doing that, one could also try to minimize the number of moves, or alternatively, minimize the number of boxes pushed. In this article we will just focus on trying to solve the puzzle.

Upon the its release in 1980, the original game consisted of a set of 90 puzzles. The first puzzle, which is shown in Figure 1, takes the average human probably less than five minutes to solve, while the last puzzle can most likely keep one busy for several hours. While in the easy levels all target squares are grouped together in some seemingly separate room, more difficult puzzles often have target squares all over the playing field, which is one of the factors responsible for an increase in difficulty.

2.2 Previous work

So far, the best known algorithm, developed at the University of Alberta, is called Rolling Stone [3]. This algorithm uses the IDA* algorithm along with several domain-dependent improvements. The IDA* algorithm on its own is not able to solve any puzzles, but the domain-dependent improvements are responsible for a large increase in performance, enabling Rolling Stone to solve 59 out of 90 puzzles from the original set. An interesting subclass of Sokoban puzzles has been introduced [5], of which all puzzles can be solved by means of a specialized algorithm in a finite amount of time. More about this subclass later. There has also been some research [8] on finding the shortest Sokoban solutions for certain puzzles. Japanese researchers claim [9] to have an algorithm that can solve all 90 puzzles, however, neither papers nor program specifications have been released.

3 Obstacles

3.1 Deadlocks

One of the biggest obstacles any human or algorithm solving a Sokoban puzzle will experience is the presence of deadlocks.

> A *deadlock* is a position that can not result in a correct solution of the puzzle.

We can roughly distinguish two kinds of deadlocks. The first kind of deadlock is related solely to the position of the boxes. For example, the player could push a box next to a box that is adjacent to a wall. Assuming the boxes are not both at a target position, this would be an undesired and irreversible position. Other examples are boxes in a corner, or 4 boxes aligned in a 2×2 position, or other more complex derived positions. Checking for these deadlocks can be extremely difficult for an algorithm, as we may have placed a box at the entrance of a very long tunnel with a dead end, so just looking in a 1, 2 or 3 block radius of the box is not near enough. We can conclude from the above that it is extremely vital to detect these deadlocks as they arise, but considering that this can be very hard, it would be better if there was a way to completely *avoid* these deadlocks. We will present a solution for this problem later.

Other than deadlocks that solely depend on the position of the boxes, one can also imagine states in which the man is at a certain position from which he cannot reach the other still unplaced boxes anymore. Obviously we do not want this second form of a deadlock to occur either, and have to find some way to check for these situations as well, which can be even harder than checking for the deadlocks that are solely caused by the position of one or more boxes.

3.2 Amount of moves

During the execution of an algorithm that is attempting to solve a Sokoban puzzle, we will want to know how close we are to a solution, and whether or not the current state will likely lead to a correct solution. Therefore it would be nice to have an indication of the amount of moves that is necessary to solve the puzzle. If we are trying to find an optimal solution, this would be a nice upper bound to use. Nevertheless, when looking for just *some* solution, we can still use this bound as an indication. It is however extremely hard to derive this upper bound, given some Sokoban puzzle.

3.3 PSPACE-completeness

Sokoban has been proven to be PSPACE-complete [1], which is the hardest set of problems in PSPACE. PSPACE is the set of decision problems that can be solved by a deterministic or nondeterministic Turing machine using a polynomial amount of memory and unlimited time. To put this into context, observe that PSPACE is a superset of NP.

4 Solving Methods

4.1 Single-Agent brute-force

The first idea that comes to mind of any algorithm designer is a brute-force approach. The theoretical branching factor in a single-agent search algorithm for solving Sokoban is 4 (up, down, left and right), but with a little reasoning we can reduce this to an average factor somewhere in between 2 and 3, as we will rarely want to move a step back (unless we previously moved a box, and want to walk away from it again), and cannot move through walls or boxes. If, for the sake of simplicity, we assume that the branching factor is about 2.5, and the length of an average solution is about 200 moves, we would end up with a complexity in the order of 2.5^{200}, an astronomically large number. That is, assuming we find the right solution, as we may just push a box in a corner, cause a deadlock, and then consider another 1000 moves before noticing we did something wrong. Even with some heuristics preventing these rather dumb mistakes, this single-agent approach is obviously not the most efficient because of the huge complexity.

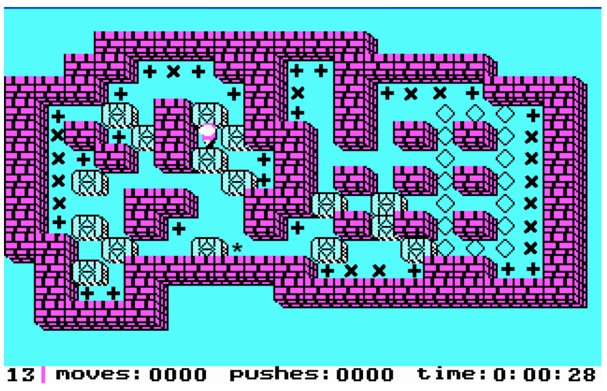

Figure 2: Puzzle 13 of the original set, with unsafe box positions ($\times$ and $+$).

4.2 Multi-Agent brute-force

We can also look at the game in a multi-agent way, an approach that is already quite a bit smarter. We see each box as an individual that is trying to move towards a target square. In order to be able to move, the boxes need to get the man to move behind them, and get the man to push them towards their target positions. Ignoring the checking for whether or not the man can actually reach the box (and how he can reach it), with n boxes, this method actually increases the branching factor to $4n$ (or $2.5n$), as at any time during the solution we may want to start moving another box, or just keep moving the box we previously moved. With an average number of moves of 200, of which maybe 50 are box moves, this approach would lead to a complexity of about $(2.5n)^{50}$, which would still not be near good enough. This major branching factor and search tree size obstacle clearly also rules out a pure brute-force approach. We will have to do better.

4.3 Heuristics

While solving a puzzle, several heuristics can be applied to both the single-agent and multi-agent approach. One of these heuristics is marking unsafe positions. Places where we never want a box to be placed, can be marked using a simple algorithm, which starts by marking corners. Note that a corner is defined by its two direct neighbours, a position (x, y) is a bottom-left corner if positions $(x-1, y)$ and $(x, y-1)$ are walls. We can mark these corners in advance, we do this with a $+$ symbol in Figure 2. For each pair of marked corners, we can now check if the squares on the line between these corners are positioned along a wall. All positions along this wall, assuming they are not target squares, are also dangerous and marked with an $\times$ symbol in Figure 2.

Never considering the positions discussed above is an obvious improvement. However, consider the position marked with the $*$ symbol in Figure 2. For the current setup of boxes, moving a box to that position will lead to a deadlock. We can however not detect this in advance, so during the execution of an algorithm there will have to be frequent checks for these deadlocks to make sure they do not occur. Other heuristics for Sokoban that have been introduced are pattern search, move ordering, deadlock tables (to quickly on the fly detect local deadlocks), and macro moves. All of these have been implemented in the Rolling Stone [3] algorithm.

5 Multi-Agent Reversed Solving with Heuristics

As described in the previous sections, deadlocks can be extremely annoying and are therefore never desired. The solving methods above all have to deal with these deadlocks, and have to apply some kind of deadlock detection. Our method, Reversed Solving, reverses the game, working from the solution back to the original puzzle. The man no longer pushes boxes, but pulls them instead. *Pulling* is defined as follows:

> A box at position $(x-1, y)$ can be *pulled* to (x, y) if the man is standing at position (x, y) and position $(x+1, y)$ is either

empty or a target square.
A similar condition of course applies for pulling in the y-direction.

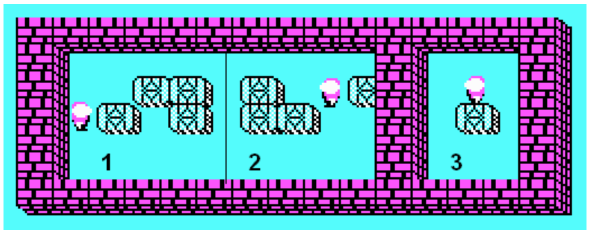

Figure 3: Pushing and pulling in three situations.

This method completely eliminates the need for box deadlock detection and prevention, as undesired states related to the position of the boxes can never be reached. In Figure 3, in the first situation, the man can push the box to the right and create a deadlock. Imagine in the second situation that the man can only pull. He can pull the box to his right to the left, but no further. The 2×2 deadlock can never occur. The third situation illustrates how, when the man can only pull, the box can never be positioned next to walls or in corners. It is of course still possible to lock the man at some position. This is often easily detected as in the next state(s) there will not be any more feasible moves.

Our algorithm, *ReversedSolving*, proceeds as follows. First the Sokoban puzzle is loaded, and "reversed": all boxes are placed at target positions. While the boxes are not all back at their original position and the man cannot reach his original starting position, we repeat:

```
While Condition $X$ is not satisfied
    Pull the box to unvisited positions.
Change to another box, guided by Condition $Y$.
```

This algorithm is still rather general, it just states that we are solving the puzzle in a reversed order. Condition X and Y together define the complexity of the algorithm and also determine which puzzles can and which puzzles can not be solved using our algorithm. In the next two subsections we will give an overview of the conditions that could be used in our algorithm, after which we will discuss how to use these conditions together to create a good algorithm.

5.1 Condition X: When to stop moving a box?

We can define several possibilities for determining when to stop moving a box:

X_1 After *each* step.

X_2 After n steps.

X_3 Until a box is at a final position.

X_4 Until a box is k steps away from a final position (where k is any integer between 0 and n, and n some integer that defines how complex this condition is).

X_5 After a random number of moves.

5.2 Condition Y: Which box is next?

After deciding to stop moving a certain box, we will want to pick a new box to start moving. We again present several possible choices:

Y_1 Every box.

Y_2 Every unplaced box.

Y_3 "Serve" the boxes in a lexicographical order.

Y_4 "Serve" the boxes in some predefined order, for example determined by the sum of their current distances to the target squares.

Y_5 The box that is currently closest to some target.

Y_6 A random box.

5.3 Combining the conditions

If we take X_1Y_1 (we denote the combination of X_i and Y_j by X_iY_j) as condition, we clearly end up with a brute-force multi-agent approach, but in this case by pulling the boxes instead of pushing them. This approach should theoretically always lead to correct solutions, however, especially for larger puzzles, this approach would still be way too complex.

A special subclass of Sokoban puzzles, referred to as the *Lishout subclass*, and defined in [5], allows solutions where boxes can be moved to their targets one by one. These puzzles can exactly be solved by taking condition X_3Y_2. An example of such a puzzle is the one in Figure 4, referred to as "the" Lishout puzzle.

We found $X_4(n)Y_2$ to be a very interesting condition. Several values can be used for n. If we take $n = 0$, we get $X_4(0)Y_2$ and are just doing the same as X_3Y_2, we are solving puzzles in the Lishout subclass. If a puzzle is almost in the subclass, meaning that we will have to keep one or more boxes one step away from its target position, then consider some other moves, and then put the box at its target, the puzzle will be solved easily with $n = 1$. Even though complexity will increase when we increase the value of n, and for large values of n (formally $n \geq S$, where S is the largest possible distance between any two squares) results in a brute-force approach, this condition appeared to perform quite well.

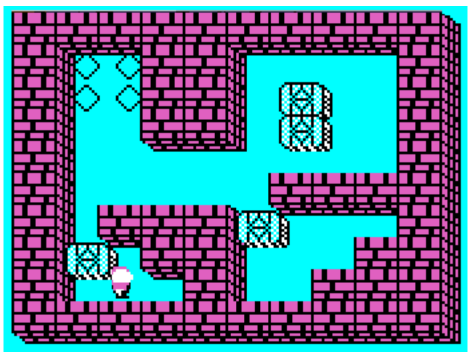

Figure 4: *Lishout* puzzle.

6 Experimental Results

We implemented Reversed Solving in C++. All game-dependent operations such as checking what a square contains, checking if a box can be moved, moving a box, finding a box or moving the man, are implemented in $O(1)$, meaning these operations do not depend on the size of the puzzle or the amount of boxes. A relatively small amount of time is spent checking if the man can reach a certain position. This operation is often needed when switching to another box, as it of course has to be possible for the man to reach it.

The complexity of the algorithm comes from the amount of generated states, and constantly checking if these states have not been explored before. Therefore we think the amount of inequivalent generated states is a good measurement for the performance. We can define state equivalence as follows:

> One state is *equivalent* to another state if the positions of the boxes are equal, and the man is in the same part of the reachable space, meaning the man in the one state can walk to the position of the man in the other state without moving any boxes.

In Figure 5 the man is in the space marked with number 1. Leaving the boxes at their current positions, but moving the man anywhere within this space (over empty squares or target squares) remains the same state. However, if the man were to be located somewhere in the space marked with number 2 or 3, then these would be actual different states. This brings the total amount of possible states for this configuration of the boxes to 3, as there is no other isolated space in which the man can be located. A hash function is used to quickly compare states.

We have determined how many *possible set-ups* exist for a certain puzzle: the possible inequivalent configurations (states) that could have been initial positions. This amount was easily obtained by running our algorithm with condition X_1Y_1, without any stopping condition when the puzzle would normally be back in its original position. The table below shows this value and also gives an overview of which puzzles were solved (denoted by Y, unsolved: N), as well as the amount of generated states for a brute-force approach (Condition X_1Y_1), Lishout's approach (Condition X_3Y_2) and Condition $X_4(n)Y_2$. In this last condition, we used the largest possible value of n to theoretically always obtain a solution.

	Set-ups	X_1Y_1	X_3Y_2	$X_4(n)Y_2$
Lishout	28276	2212 Y	54 Y	54 Y
Orig. 1	148501	82922 Y	6331 N	11001 Y
Orig. 78	30+ min	30+ min	2197 Y	2197 Y
micro1	39	33 Y	19 N	33 Y
micro7	2103	786 Y	13, Y	13 Y
micro10	374	323 Y	33 N	208 Y
micro25	96	77 Y	34 Y	34 Y
micro35	6721	920 Y	757 N	920 Y
micro75	1625	204 Y	114 N	204 Y
micro78	11270	2441 Y	114 N	215 Y
micro106	8466	5157 Y	25 N	1296 Y

As we expected, the Lishout approach (X_3Y_2) solves the Lishout puzzle (see Figure 4) quite easily, but for example not Puzzle 1 from the original set (see Figure 1), because simply not all states are generated. Condition X_1Y_1 also solves the Lishout puzzle, but generates a considerably larger amount of states in the progress. The brute force approach also solves Puzzle 1, but again generating a lot of states in the process. Even though all this is still rather complex, it seems nearly impossible to solve Puzzle 1 non-reversed without any heuristics, because of all the possible deadlocks that can occur. A relatively big puzzle from the original set, Puzzle 78, which is shown in Figure 5, was solved quite fast with condition X_3Y_2, as it was in the Lishout subclass, while a brute force approach fails, analyzing over 200,000 different states, running over 30 minutes on a 2.4GHz machine. This shows how seemingly complex puzzles can be solved quite fast with the Lishout approach, but take way too much time with the brute force approach. The drawback is of course that it does not solve all puzzles, as opposed to condition X_1Y_1. We will have to do a little better.

Quite some puzzles appear to be *almost* in the Lishout subclass, e.g., Puzzle 1 from the original set (Figure 1), again shown in Figure 6, after pushing two boxes once. The puzzle is now in the Lishout subclass, and can easily be solved with condition X_3Y_2. Condition $X_4(n)Y_2$ comes in handy: it tries to put puzzles in the Lishout subclass. Observe that this method does solve Puzzle 1 in a reasonable amount of time.

The numbered micro-puzzles (puzzles from a big set of puzzles called Microban [6]), all solved in less than a second with $X_4(n)Y_2$, were added to illustrate the difference between the discussed approaches and how well the conditions perform compared to the amount of possible set-ups.

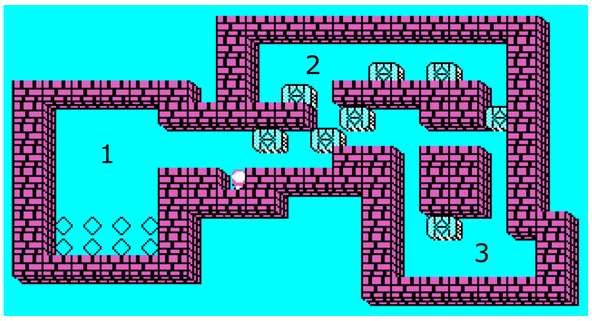

Figure 5: Puzzle 78 from the original set.

7 Conclusion and Future Work

Sokoban puzzles are an extremely interesting subject for the field of Game Theory and Artificial Intelligence, as a perfect algorithm has never been — and can probably never be — found. In this thesis we have presented a new method for solving Sokoban puzzles, called Reversed Solving. The basic idea behind this method is already a big improvement compared to a regular brute force approach, but it has to be adjusted with smart heuristics to give decent results. We have defined two conditions that determine both the solvability and the complexity of the algorithm, and can be used to specify heuristics. We experimented with several conditions and have shown how they perform, and found one of particular interest, a method which basically "tries" to get the puzzle into the Lishout subclass, a quickly solvable type of puzzles.

For our algorithm, the complexity lies within checking whether or not states have been visited before, and the amount of states is therefore a good complexity indication. An interesting piece of future work would be to speed up this checking process, to make the algorithm run faster. Currently, the biggest open problem lies within the fine-tuning of Condition X and Y. What is the best heuristic, where lies the best trade-off between solvability and complexity?

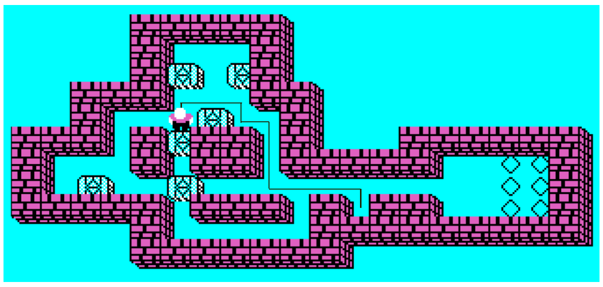

Figure 6: Puzzle 1 from Figure 1, after moving up, left, left, left, up, up, up, left, up, left, left, down.

Acknowledgements

This article (and the corresponding Bachelor Project) was written under supervision of dr. W.A. Kosters from LIACS, Leiden University. Many thanks to Walter for all the support.

References

[1] J. Culberson, Sokoban is PSPACE-complete, Proceedings in Informatics 4, Fun with algorithms, pp. 65–76, Carleton Scientific, Waterloo, 1999.

[2] D. Dor, U. Zwick, Sokoban and other motion planning problems, Computational Geometry 13 (215–228) 1999.

[3] A. Jungmans, Pushing the limits: New developments in single-agent search, PhD Thesis, University of Alberta, 1999.

[4] A. Jungmans, J. Schaeffer, Sokoban: A challenging single-agent search problem, Proceedings, IJCAI-97, pp. 27–36, 1997.

[5] F. van Lishout, Single-player games: Introduction to a new solving method, Master Thesis, University of Liège, 2006.

[6] D. Skinner, Microban [accessed May 9, 2008], `http://members.aol.com/SokobanMac/levels/microbanText.html`.

[7] F. Takes, Sokoban: Reversed solving, Bachelor Thesis, Leiden University, 2007.

[8] W. Wesselink, H. Zantema, Shortest solutions for Sokoban, Proceedings 15th Netherlands/Belgium Conference on Artificial Intelligence (BNAIC '03), pp. 323–330, 2003.

[9] Sokoban, Wikipedia [accessed May 9, 2008], `http://en.wikipedia.org/wiki/Sokoban`.