Sokoban Solver

A Push in the Right Direction

Simon Kotlinski, Rasmus Göransson, Peter Boström

Peter Boström, 890224-0814, pbos@kth.se
Rasmus Göransson, 850908-8517, rasmusgo@kth.se
Simon Kotlinski, 870428-2931, simonko@kth.se
Artificial Intelligence DD2380
2010-10-17

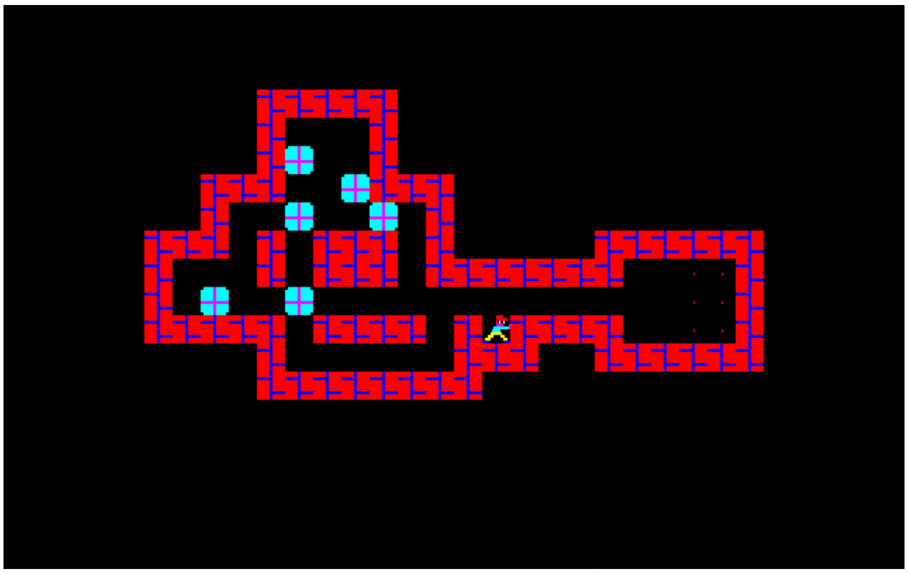
Abstract

This paper is a project result of the course Artificial Intelligence at KTH. The project was about implementing a agent solving Sokoban levels. Sokoban is a video game from 1981, where a player solves a tour puzzle. We describe problems encountered and what methods we used to solve the 136 given levels.

Table of Contents

What is Sokoban?

Sokoban is a Japanese video game from 1981 created by Hiroyuki Imabayashi. You work as a warehouseman and your mission is to arrange big boxes in a warehouse. The rules of Sokoban are simple. Every room contains a number of boxes and you want to move them into specific slots. This may sound as a small issue, but your character has just enough strength to push one box at the time. This means he have no chance pulling one box backward or push two boxes forward. The player cannot move through a box or a wall. A box in a corner cannot be moved in any direction. [1]

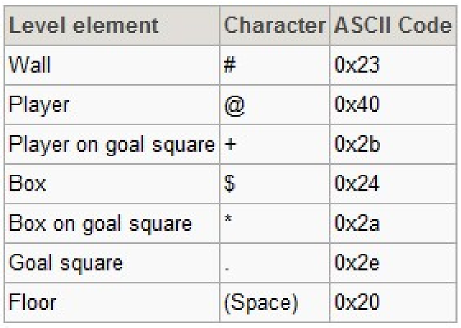

A typical Sokoban level.[2]

The picture above is a Sokoban level. The pale blue symbols are boxes, the red symbols are walls and red dots at the right hand side are the goal.

[1] http://www.sokoban.jp/, Official Sokoban site, October 14, 2010.
[2] http://upload.wikimedia.org/wikipedia/en/6/61/Sokoban_pc8801.png , Wikipedia, October 14, 2010.

Problem

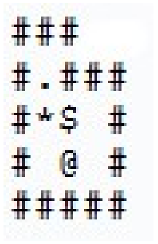
The task is to implement an agent that can play the game of Sokoban. We have access to a Sokoban game server that provides us with levels. Every level is a string of characters. The set of characters are #, @, +, $, * and (Space).



Level element	Character	ASCII Code
Wall	#	0x23
Player	@	0x40
Player on goal square	+	0x2b
Box	$	0x24
Box on goal square	*	0x2a
Goal square	.	0x2e
Floor	(Space)	0x20

Characters translation[3]

From now on we will use the characters rather than pictures describe levels, situations and algorithms. The player can move in four directions *U(up), D(Down), L(left), R(right).* A solution to a level is a sequence of directions ending up with all boxes on a goal square.



```
###
#.###
#*$ #
#  @ #
#####
```

A simple Sokoban level

One correct sequence for this level could be: *"L U D R R U L"*.

[3] http://www.sokobano.de/wiki/index.php?title=Level_format , Sokoban Wiki, 2010-10-15

Method

The following is an overview of the strategy used to solve Sokoban boards. This alone is not (nearly) enough to solve most of the boards. We also use a lot of optimizations to deal with Sokoban's huge depth and branching factor. The first one we figured we had to put in there even to solve the simplest of boards was to only consider box moves as new states and not player moves.

1. Fetch the board from the server. Translate the data from the server to a level object. The level object contains information about the initial board and its initial state.
2. Create the first board-state which holds box and player positions.
3. From this initial state, we create states for all possible box pushes that can be done from this state. That is, each side of each box that the player can reach and push forward gives birth to a new state. Every new state remembers which state created it to enable backtracking to find the solution string. These states are put into a queue.
4. We repeatedly pop the state with the lowest cost from the queue and expand its possible moves into new child states that are also put into the queue. We traverse the solution space this way and finally when pop a state where all boxes has reached a goal we have our solution.
5. The solution string is created by recursively backtracking through all the states parents and concatenating each move sequence. The solution string is a path containing every move needed to reach the solution.
6. This path, represented by character movements, U, D, L, R will finally be sent to the game server for approval.

Optimization

Sokoban is a board game with a branching factor and search depth comparable to chess. We have had to make a lot of optimizations to be able to solve even simpler boards. Solving non-trivial boards by traversing the whole board within reasonable time is simply impossible. Some states are deadlocked and cannot possibly lead to a solution. Therefore, checking any further movements made from this state is a waste of time.

Repeated states

Just like a regular graph-search algorithm, not detecting revisits of old nodes, or states, can be devastating to performance. Even if we try making the same move a million times, the outcome of that specific move will always be the same. Therefore we need to know which states we have already visited. We use a hash set to allow fast insertions and lookups.

Calculating and storing hash keys for states can be very efficient. Hashing a state means that we get an almost unique key for each possible state in the level. Using these keys, we can find out where the states are in the hash set and see if a state has been visited before in almost constant time.

We create the hash key from the position of all boxes in the level. Even if two boxes switch positions, we store them in sorted order. If we wouldn't, the hash function we're using would give a different value for the two identical box configuration.

Note that the player position does not affect the hash-value for a state. This is important, because a state could be equal to another even if the player position differs. Two states, a, and b, are completely equal if their boxes are placed in the same positions and the player in state a can reach the player in state b without moving any boxes.

```
####### #######
#@ $  # #  $ @#
####### #######
```

Two different states with the same hash-value

Prioritizing states

Some states simply show to be more promising than others. One which has many boxes placed on goals is more 'likely' to quickly reach a solution. Therefore we want to spawn states from these first, without completely disregarding the others.

We've constructed a priority queue structure with a hash set built into it. This guarantees that no duplicates are inserted into the queue and that we pull the most promising states first. Initially we generate the first 32 possible states without ordering by cost, to have a more balanced set of moves to start searching from. This prevents the solver from taking a long detour before checking the other possible initial movements.

Assigning State Cost

The cost of a state is given by the minimum number of pushes the boxes needs to be pushed in order to reach a goal disregarding blocking boxes plus 100 for every box not on a goal. By adding a cost for every box not on a goal, we get a higher priority for placing boxes on goals.

The distance from every position is calculated by doing a backwards breadth-first search pulling boxes from all goal positions simultaneously and marking distance along the way. By doing this, any position that has not been assigned a value, must be a dead position, from where a box could never reach a goal. Any move to these positions would lead to a *deadlock*.

```
##########
# 210123 #
##########
```

The 0 is a goal and 1,2,3 are the distance to the goal. The spaces are areas which cause deadlocks.

Deadlocks

A deadlock is a configuration of boxes that cannot lead to a solution. There are many ways to end up in a deadlock. During the development of our solver we have printed states during the solving process. This way we have learned about different kinds of deadlocks and have created tests to avoid many of them.

```
$$    ##       #       $$           #            #########
$ $   $ #      $$      $$   $$      $#  #  $ #   # .** $
 $$    $#       #           ##          ######   #########
```

Examples of deadlocks

It's however important to consider that most of these cases need to be handled differently if the boxes are placed on goals or there are goals in the middle of them. It's important to not get false positives, so that each board still can be solved. Deadlocks however are important to detect, so it's important to get as many right as possible.

Future Deadlock Work

One deadlock case we did not have time to implement is a check for is the so called corral deadlock. A corral deadlock is an area which the player can't reach without causing another deadlock. This check could be done by inspecting each area and its surrounding boxes only, to see if the player can break into that region without creating deadlocks.

```
##########
#.  @ $  #
#.    $  #
##########
```

A corral deadlock; the player cannot reach the area to the right without causing a deadlock. The boxes are dead, and the state is therefore also dead.

Tunnel warping

Inside a tunnel without goals, there can only be one box without having a deadlock. Two boxes inside means that you can only push the boxes together and never apart. Therefore when a box is pushed into a tunnel, we keep on pushing it to the far end without stopping, except if it reaches a goal, then we spawn a state from there. This lets us cut some of the states from the tree, as a tunnel of many squares can be seen as three states outside from one side, outside from the other, and inside. Where it is located inside doesn't really matter. This leads to further cuts in the tree both from less states and detecting deadlocks when multiple boxes are inside tunnels without goals.

Converting Immovable Boxes on Goals into Walls

States with immovable boxes that are placed on goals can make what looks like a deadlock into a valid position. As these boxes cannot be moved however, they can be treated the same way as walls. Therefore we create a more restricted version of the same level for the state, as soon as these box placements occur. The position costs are recalculated with respect to the immovable boxes. This lets us detect deadlocks where other boxes stops being able to reach any goal. This proved to be effective and lead to another batch of boards solved in time. One board (#35 on the test server) was even reduced from having to do a few hundred thousand expands into requiring less than 100.

Partial bipartite matching

We continuously make sure that every box can reach a goal by looking at the position cost but we also make sure that all goals have at least one box being able to reach it. Further improvement would be to do a full bipartite matching to make sure that there is a way to match every box to a single goal without any boxes sharing the same goal.

Compiler Optimizations

One thing we learned is that compiler optimizations can also be very effective. Not to improve algorithmic detection, but to make the program more effective so we have time to search deeper. Finally adding the highest-level optimization flag ”-O3” to g++, decreased our running time (on 136 boards) from 19 minutes, of which we didn't finish 7 boards in time, to 7 minutes in total. This let us pass all boards, and some of the ones we used to fail were passed within 25 seconds.

Results and Conclusions

The solver has a hard timing many of the original Sokoban levels. These contain situations which lead to many corral deadlocks and dead regions. We don't have those checks implemented. They are built in such a way that you're supposed to solve how to push one box into the goal region then repeat that for each box. This usually leads to a huge solution string and our solver, due to the branching factor, has a hard time coping with solving these.

We believe that this could be addressed by attempting to move a box ”as much as possible”, to all states where it could go before pushing another box. This means that a state where a box has been pushed all the way into the goal region (which gets quite high priority) becomes available way quicker. This could lead to a quicker solution but could also lead slower solutions because of the increased branching factor.

As a conclusion: Sokoban is a very complex puzzle game with a vast amount of tough cases. The test levels however work quite well. We are able to solve all 136 boards in around 4m 21s, as of our final test. Some of these levels work way better than others.