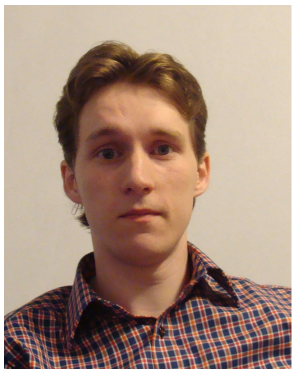
Sokoban Project Report
by FAMA

Version 1.0
October 10, 2012

Abstract

In this report we present an algorithm for solving the Sokoban problem. The algorithm is a Best-First search which uses a heuristic to try and estimate the value of the state of a Sokoban map. We find that our strategy is quite successful on our test maps but we also find that there's room for improvement, mainly in how we determine which goals should be filled first.

Andreas Brytting, 890609-0595



Filip Gauffin, 870218-0657

Anton Holmberg, 891227-0553

Magnus Raunio, 900531-5131

Contents

1 Introduction

Sokoban is a grid-based 2D puzzle game where the player has to push boxes onto goal tiles. You can only push boxes forward. The game is finished when all boxes are on goal tiles. Our assignment was to make a computer agent that can play this game successfully.

The Sokoban problem can be solved, like a lot of other games, by searching the state space of the game until you find a solution. The state space of a Sokoban game can be quite large though so it's necessary have a strategy when searching the state space so that you generally search/move in the right direction and the strategy we select to guide our search in the state space is really what can be considered the AI part of the solution.

The strategy can be something like using a A* search to try and minimize the total distance moved (And thereby assume that the solution should generally be quite simple and not require so many moves) or perhaps to use hill climbing to try and improve the current state by moving closer to a solution.

Our four member team has tried to solve the Sokoban puzzle problem using a search strategy which tries to guide the search algorithm to a solution by selecting the best state to expand, so essentially a Best-First search strategy. Each member of the project and their contribution to the project is listed below.

Andreas Brytting Wrote unit tests, tweaks to the heuristic and helped put all the pieces together.

Filip Gauffin Translating the main pseudo-algorithm to code as well as checks for equality between visited states. Also testing different costs for the heuristics.

Anton Holmberg Created unit tests, helped design the heuristic and implemented the solution visualizer for the game.

Magnus Raunio Worked mainly on dead end detection to allow for early pruning of states during the search. Wrote the main classes State and Board.

2 Problem description

Sokoban has pretty simple rules but the variety of maps and the number of possible moves you can take makes it a pretty difficult problem nonetheless. A Sokoban map (a example can be seen in figure 1) consists of the following:

A Player The player is what you can move around and use to interact with the environment. A player can only move either up, down, left or right and can only move one tile at a time.

Boxes Boxes are object which the player can move by ”pushing” them. A player pushes a box by stepping on the box which is then moved to the tile adjacent to it in the same direction that the player moved onto the box. A prerequisite for pushing a box is that there’s neither a wall or another box already in the tile to which we are trying to push the box.

Walls Walls are tiles on which the player cannot move nor can boxes occupy a wall tile.

Empty tiles Empty tiles can hold a player or a box and players are allowed to move over them.

Goals All goals need to have a box placed on them in order to solve a Sokoban map, once this is done the player wins. Both boxes and players can move over the goals. There’s always as many goals as boxes.

So we can see that a box need to have either an empty tile or a goal on both sides of it either in the y-axis or the x-axis in order for it to be able to be moved. If a box is not movable and doesn’t stand on a goal we are in a dead end and the board cannot be solved. Of course, having two empty tiles/goals either on the x-axis or the y-axis doesn’t guarantee that we aren’t in a dead end, perhaps neither of those empty tiles can ever be reached by the player and then we have a dead end anyway. Note that we mainly use the term dead end to refer to a state of the board from which there is no possible solution no matter what you do but also occasionally use it to refer to tiles on the board from which it’s impossible to push a box to a goal.

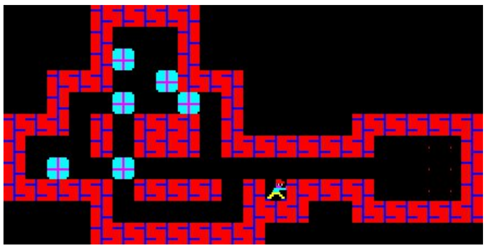


Figure 1: A example map of a Sokoban game. The dots are goals, the blue squares are boxes, the red blocks are walls, the black squares are empty tiles and the little man is the player.

3 Design and Implementation

3.1 The Structure

Our initial approach to designing a solution to the Sokoban problem was to simply try and minimize the total distance between the goals and boxes by moving the closest box to their closest goal until it was on top of said goal. While doing this we would build a search tree through which we could backtrack if we ended up in a bad node, from which no solution could be reached, and try a different branch instead.

This was the initial idea and to do this fast and efficient we had to consider the structure of the search tree, how to find loops in the search tree as well as how to do early dead end detection to avoid expanding dead branches of the search tree, i.e. pruning. We could probably improve on the heuristic for measuring distance to a solution as well other than simply taking the total distance between goals and boxes. So basically there's two parts to our search strategy, firstly we need to create a good structure for the search tree and pruning it and secondly we need to have a good heuristic to tell us if a state is close to a solution so that we know which nodes in the search tree we wish to expand.

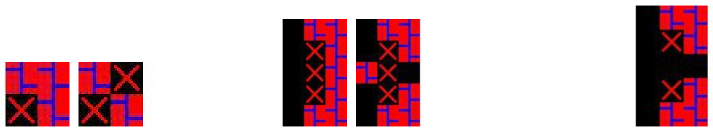
So to limit the size of the search tree we broke the Sokoban board up into two subcomponents, a **Board** class and a **State** class. The State objects are our nodes in the search tree and the Board object represents a static environment of the Sokoban map. We also use a **Action** class which represents a movement of a box by the player, each Action takes us from a State object to another State object.

The Board component is a object which is basically a matrix that tells us the layout of the Sokoban board, i.e. goals, empty tiles and walls. The board also has dead ends marked which means that if we push a box into a marked dead end it will be impossible to solve the Sokoban board.

The dead end positions on the Board are either corners without goals or long stretches of empty tiles for which both ends are marked as dead ends as well as it's impossible to move the box sideways on that stretch. In figure 2(a) you can see corner dead ends. In figure 2(b) you can see a long stretch of dead ends where both ends are dead ends as well as it being impossible to move the box sideways. The long stretches of dead ends are found by traversing from each corner dead end and see if we end up in another dead end before we find that we can push the box sideways, like we can do in figure 2(c).

The Action component consists of a position on the board and a direction (Up, Left, Right or Down). The Action symbolises from which position we pushed a box and in which direction we pushed the box. Basically it tells us where the player were and in which direction the player moved, but an Action also always results in a box being moved.

The State component consists of a list of box positions, a list of possible Actions to take in the state as well as a pointer to the previous state


(a) Corner dead ends.

(b) Dead ends detected inbetween corner dead ends

(c) A example of when we don't mark a tile as a dead end when it's between two corner dead ends

Figure 2: The red X:s mark dead ends for boxes which we detect on the board.

we were in to get here and the Action we took in that state to get to this state. It also stores a distance value which we use to estimate how close to a solution the state is. The pointer to the previous state is necessary so that once we've found a solution we can backtrack through the search tree and find our path to the solution by checking each Action taken to get to each state. You may note that we do not store the players position in the State class, this is because we can easily calculate this by checking the Action to get to this step.

The list box positions simply tells us where on the Sokoban map we have boxes. The list of possible Actions we can take in a State is created by cross referencing the box list with the static Board object. We do this by doing one BFS from the players current position to each empty tile next to each box, making sure not to cross any tiles blocked by walls or boxes. Note that we only do one BFS for the whole state to find all possible Actions for that state.

Once all Actions have been found we wish to filter out the Actions which will lead to dead ends. Firstly, we can check the Board object to find which Actions lead to dead ends, but we also need to check for dead ends caused by boxes. If we for example have four boxes in a square and not all of them are on a goal, we are in a dead end, see figure 3.

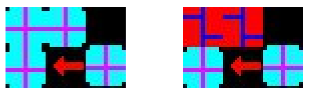


Figure 3: Typical situations where boxes may cause dead ends when placed next to each other.

We will describe the heuristic to calculate the distance estimate later since knowing how it works isn't really important to understanding the structure of our algorithm.

Now that we've described all the components which are used to construct the search tree let's go into detail on the algorithm itself. In figure 4 we can see pseudo code for the algorithm. It's quite easy to summarise though, basically we have a queue with the best states first (The ones with the lowest distance to a solution) which we loop through and expand each States' Actions into new States which we add to the queue until we find a solution. We also track already visited states to avoid loops.

```
# rootState is the starting state for the Sokoban board
# board is a Board object representing the Sokoban boards layout, walls and goals
# q is a priority queue where the state with the lowest distance value is queued first
# visited is a hashset for visited states to allow for loop detection in our search tree
solve(rootState, board):

  # Add the root state to the queue with the cost 0
  q.add(rootState, 0)

  # We search for a solution until weve exhausted the search tree
  while(!q.isEmpty()):
    currentState = q.next()
    visited.add(currentState)

    # Expand each action into a new state in the search tree
    for action in currentState.actions:
      nextState = createState(currentState, action, board)

      # Check if the state is a solution
      if nextState.boxPositions.equals(board.goalPositions):
        # Reconstruct the steps taken to reach it and return these steps
        return backtrack(nextState)

      # Add the state to the queue if we have not visited it already
      if !visited.contains(nextState):
        q.add(nextState, nextState.cost)

  # In case we dont find a solution, return null
  return null
```

Figure 4: Pseudocode for our algorithm when searching for a solution for the Sokoban board

So while the algorithm is quite simple it really relies on having a decent distance measure for each state so that we don't expand "bad" states. When we create each new State we also calculate a distance value using a heuristic.

3.2 The Heuristic

The heuristic we used to evaluate a state consist of three different measures which are added together. The output of the heuristic function is called the heuristic value in the sections below. Also note that a low heuristic value means that a state is preferable.

Distance from box to goal

The first part of the heuristic is based on the simple observation that boxes close to goals is generally a good idea. This idea is implemented by finding the box which is closest to a goal and adding this distance to the heuristic value. Boxes that already are on a goal tile are excluded.

The distance from a box to a goal is approximated by the following formula:

distance = abs(goal.x - box.x) + abs(goal.y - box.y)

One question that might arise is why only the minimum distance to a goal is used instead of the average distance? This is because it makes the search algorithm focus on one box at a time, which we found is generally a good idea.

Boxes on goals

The second part of the heuristic is based on the observation that it is generally good to have boxes on goal, especially if there are a large number of adjacent walls to the goal tile. This is implemented by letting each box on a goal give a large negative addition to the heuristic value. The size of the addition increases with the number of adjacent walls to the goal tile according to the following formula:

heuristicValue -= 20 * (1 + numberOfAdjacentWallElements)

In this context we also consider other boxes to be walls.

Search depth

The third and final part of the heuristic is based on the observation that it's often good to keep solutions short.

This might not be apparent at first glance as we did not have any restriction on the length of the solution in the assignment. But this is important because dead ends can occur when searching even if we try to detect these in the main search algorithm. If we continue to explore that part of the search tree we will never find a solution and we often have to search VERY far before we exhaust that whole branch of the tree. By penalizing the search depth we decrease the chance of exploring branches of the search tree that appear promising, but in reality we have already reached a dead end a long time ago. This is implemented in by letting each state have a number which indicates its depth in the search. this number is simply added to the heuristic value. A example of a dead end which we don't prune in the search tree can be seen in figure 5.

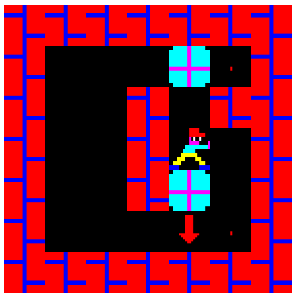


Figure 5: The figure shows an example of a dead end which we don't detect.

3.3 Solution visualizer

Finally we also had a tool which we call the solution visualizer which basically let us play a solution to observe how our algorithm did. This allowed us to get a better understanding of our algorithms strengths and weaknesses and helped us improve the algorithm.

3.4 Quick summary

So to summarize, our algorithm relies mostly on early pruning of dead ends and figuring out what state to explore next by using a heuristic function based on distance from box to goal, number of boxes on goal and the current search depth.

4 Results and analysis

When we allow our algorithm to try and solve 100 test boards we find that it managed to solve a total of 70 of them. The results of the test can be seen in figure 6 and as we can see most of the maps are solved quite quickly (under 30 seconds) or they are not solved at all. Since we assume we should have a quite even/natural distribution of map difficulties so we should really see a more homogenous distribution of the time it took to solve maps or perhaps something that look like an exponential decline. What we should see at least is that it takes progressively longer time to solve maps but what we see is that it pretty much takes stop at one point. Perhaps this could mean that there's certain characteristics for the unsolved maps which make it hard for our algorithm to solve them.

When we look at some of the maps which the algorithm couldn't solve under one minute we find that what they all seem to have in common is that

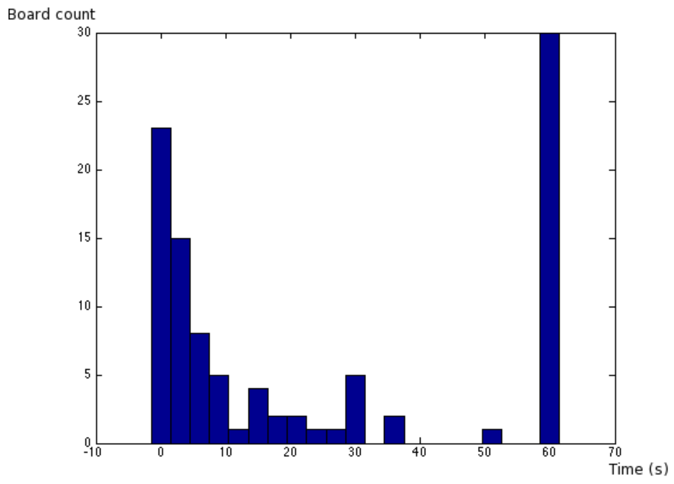


Figure 6: The histogram shows the distribution of running times for the 100 test maps we tried to solve. All maps which had a running time of 60 seconds or over can be considered as failures and has been collected in the 60 seconds bar.

it's possible to block access to multiple goals by pushing a box onto a goal. Examples of maps which the algorithm couldn't solve can be seen in figure 7

What probably happens right now is that the algorithm blocks off access to a bunch of goals and continues to do a lot of work and then end up having to backtrack a lot. It could be seen as a type of dead end which we don't detect properly really.This is somewhat countered by the fact that we try to keep the search depth shallow but perhaps it's not enough.

A possible solution to this could either prioritize the goals so that the goals in corners are filled first and the ones in corridors are filled last. We have a current prioritizing between the goals by looking at the number of blocked tile next to a goal which makes goals with many blocked tiles next to them having a higher priority. This helps to fill the corners first but it does not help us from blocking corridors to goals which are in the middle of a room with no blocked tiles next to them.

Another possibility is also to allow the algorithm to sometimes make "bad" choices, i.e. push a box off a goal, since it seems like the box just needs to be pushed further to arrive at another goal eventually. We could also make random choices sometimes in order to try and clear corridors. The idéa of prioritizing the goals seems more likely to succeed though since it gives a more precise strategy to follow and it could more easily be incorporated with our current design. Further we could also have tried a variation where the player push boxes to the goal furthest away, but on boards where the

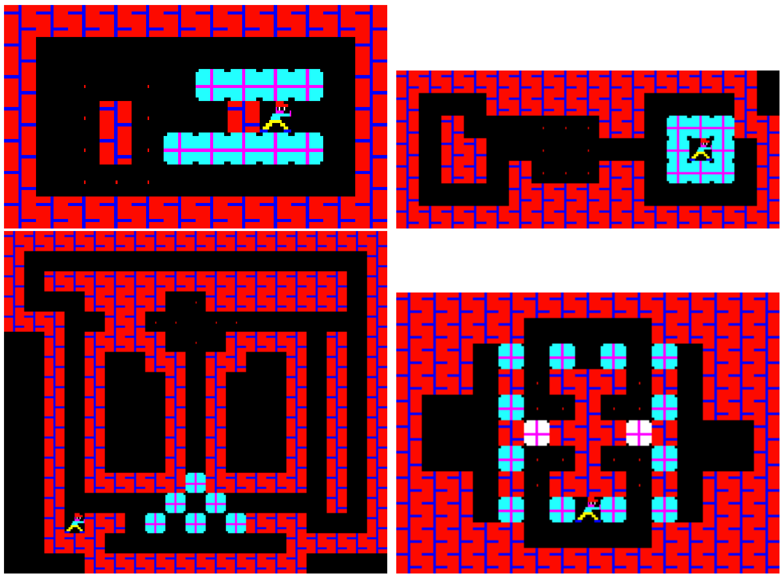


Figure 7: A snapshot of maps which the algorithm couldn't solve in under one minute. What they all seem to have in common is that it's easy to block access to other goals by filling another goal which is closer to the boxes. White boxes are boxes allready on goals.

goals are in different places this might lead to boxes being pushed in the wrong direction.

5 Reflections

When we first started working on the project we all sat down together and wrote pseudo code for the program. We made pseudo code for the whole solution, except for the heuristic which proved to be a little bit trickier. Having the pseudo code was very helpful and it made it easier to write the real code since everyone knew more or less how the different parts would fit together.

For the heuristic we tried to figure out what would be a "good" or "bad" move and how to translate that so the agent would understand and be able to make the right choices. We designed our solution so that it would be easy to swap the kind of heuristic we wanted to use since this would be one of the features that would need the most tweaking.

We choose to start with the simple heuristic of calculating the distance as explained in section 3.2. We could then use the solution visualizer to easily observe how the final solution looked like and see what it did that was stupid and how to improve it. This was a great aid in developing the heuristic further as well as finding bugs that would have been difficult to find if we were just looking at the solution as a string of moves.

Once we had the algorithm running with the basic distance heuristic we added different values for different goals to try and prevent a box from blocking goals if it is not pushed close to a wall. Finally we added the depth-value limiting how advantageous it is to go deep in one direction of the search-tree.

During testing we would look at the length of the solution, that is how many steps that were taken to win, and the time it would take to find that solution. Tweaks to our algorithms would sometimes make the solution shorter but also increase the time it took to find which was a problem since the time we had was limited.

So while we managed to solve 70% of the test boards there's still room for improvement. As suggested in section 4 we could have a more advanced goal prioritization. You could also easily try and experiment with different heuristics as well and see if they improve things but we are quite happy with the results we got and really the only thing we would consider changing to improve the algorithm is the heuristic for the distance to a solution.