Sokoban solver
DD2380: Artificial Intelligence

Lovisa Skeppe 1988-11-12
Anton Lundquist 1989-01-01
Sam Lööf 1989-07-09
Hobrik Wärnegård 1988-11-05
Abstract
In this paper we have described our implementation of an AI agent that plays the game of Sokoban. We used an A* search in both game space and state space and have implemented some protection against both static and dynamic deadlocks. At the end of the project we also implemented macros for tunnels which helped us solve a few more boards. The final result was an agent that could solve almost half of its given set of boards.
1 Introduction

Sokoban is a Japanese puzzle created by Hiroyuki Imabayashi in 1981 and distributed as a computer game in 1982 by the computer game company Thinking Rabbit.

Sokoban can be seen as a simplification of an automated robot organising storage units in a warehouse. It is therefore a great problem to deal with since it is easy to understand, hard to solve and has a real world connection. The complexity of solving a Sokoban game has been proven to be NP-hard and PSPACE-complete. This is due to its large branching factor and solution depth. When searching for a solution it is important to have good heuristics for which branches to expand thus minimising the required computations.

1.1 Sokoban rules

The rules of Sokoban are fairly simple. You have a game environment which is a plan of a room or a floor (see figure 1) and is fully observable during the whole game. The game involves moving boxes to certain goal squares placed in the environment in least amount of time or least amount of movements. To move the boxes around there is a warehouse keeper that is operational by the player. The warehouse keeper can only move in horizontal and vertical directions and on empty squares. The keeper can walk onto a square with a box on it – and thus pushing the box in front of him – if the square behind it is empty.

![Figure 1: Screenshot of a Sokoban program](image)

2 Problem

To be certain that an implementation finds a solution it needs to search through all combinations of moves. The main problem when implementing a Sokoban solver is that the branching factor in combination with the solution depth quickly gives a large amount of combinations. If the implementation is to be successful, the implementation needs to discard moves that will not lead to a solution as quickly as possible.

3 Approach

3.1 Main idea

We are basing many concepts of our Sokoban solver on the work of Timo Virkalla’s master thesis. To briefly explain our design we are building a state graph and then the we do an informed search in the graph to find the solution path to a goal state (all the boxes are on goal tiles).
We define a state as a representation of the map where the positions of the boxes in combination with the places that the player can reach by moving around (not pushing any boxes) are unique. So states change when boxes are moved, not when the player moves. Since a player can move around in an open space of the board without doing any box movements, we have not really come any closer to a solution. So by only regarding box movements we can heavily minimize the branching factor in the state graph.

We will do two types of path searches. One of them is finding a path in the state graph or also called space state, where the search goal is finding paths for boxes to goal and paths for player to boxes. The other one is done in game space. This is where we will try to move a box in all directions and adding that change as a state to the state graph. By adding all the possible box movements to the state graph we ensure that a solution will be found. However, we cannot assure that this path will be found in polynomial time because the problem is proven NP-hard.

By adding all possible movements we will get an enormous state graph. This will take both time and memory to create and traverse. Therefore we will prune away states that cannot possibly lead to a solution. There are several techniques for doing this, we decided to implement checks for dead spots, deadlocks and tunnels.

3.2 Informed search in game space

The state space search is implemented with an A* search. Every state is asked to provide future states that are added to a queue. The queue is a priority queue, which is a sorted queue that takes O(log n) time to perform insertions and constant time for polling the first element. The elements are sorted according to our heuristic: Sum of Manhattan distance + number of steps.

The Manhattan distance is the distance between two points on a grid calculating the path of the x-axes + y-axes (the blue, red or yellow line in figure) instead of the euclidean distance which takes the linear distance (see green line in figure 2). Since we often have more than one box, the sum of Manhattan distances is calculated with a bipartite matching between the boxes and the goals. The combination that has the lowest sum of Manhattan distances is used. Number of steps is at which level we are at in the search tree.

![Figure 2: Illustrating Manhattan distance](image)

This means that even if we add a lot of states to our queue, we will explore the ones that lead towards the goal first. This will not improve the worst case scenario performance
of our implementation, but it will increase the probability that we find a solution in a reasonable time.

When the states are asked to provide future states, they try to move each box in the state one step in every direction. They also use the game space search to try to move each box to every goal square. This too will not improve the worst case performance, but increases the probability to find a solution faster.

We also used a HashSet that contains all the states that we have already visited. This prevents loops in the search tree. This is required for the implementation to be deterministic. Since we cannot generate an integer hash that contains the position of every box and the player, we cannot use the hashes to compare states. To make the implementation faster, we start with comparing the normalized player position. If that matches, we traverse both states and compares each found box. If they are all the same, the states are equal.

3.3 Informed search in state space

As mentioned before we are using A* search to find paths both for the player and the boxes. In the search we make use of a priority queue and a list with explored positions or nodes. The priority queue is filled with new states where we try to move the player or box that we are looking at one step in every direction. The cost to get a new node is one more than the previous node. The priority queue is sorted with the heuristic cost + Manhattan distance. Since we try all combination of moves we will get a path from start to finish if there is one.

A* is first expanding in the direction towards the goal which minimizes the distance between the start and target position. If the way is blocked it tries the next shortest path and so on provided a good heuristic.

3.4 Deadspots

Dead spots can be found immediately on the initialization board. Since it is only calculated once and it is just a loop over all map positions it is not time consuming. When we later shall perform a move we check before the move if it will result in a dead spot. If that is the case, the move is never performed. This will have a big effect on the search time since the branching factor will be reduced. The dead spots are found by clearing the board from boxes and check if there exists a path for a box from each tile on the board to any goal. If not we mark that tile as dead (see figure). A path is only valid if the warehouse keeper can be able to push the box in front of her/him.
This search uses the same search as throughout the game for finding new states. It is an A* with the same heuristics as before Manhattan distance + number of steps so far.

### 3.5 Deadlocks

When moving a box certain situations can occur where we cannot keep moving the box, even though it is not on a goal. We can think of these situations as moving a box into a dynamic dead spot. The easiest example is when a move creates a 2 by 2 square containing walls or boxes. Then there is no way to move the box(es) out of the square! There are of course scenarios when deadlocks are created that are bigger than 2 by 2 squares. We used Virkalla?s chart of 2 by 2 and 3 by 3 patterns that are deadlocks to generate our database of deadlocks.

Deadlocks were implemented using HashSets and our generated deadlocks. We used one HashSet for 2 by 2 deadlocks and one HashSet for 3 by 3 deadlocks. The HashSets were initialized with hardcoded patterns that are deadlocks. When we try to move a box we check if the new position of the box in combination with it's surrounding squares matches a deadlock. If it does, the new state is not added to the state queue.

This requires 4 different surroundings that we need to check against 2 by 2 deadlocks and 9 different surroundings for 3 by 3 deadlocks. Since these 13 lookups are done for every state that we try to add to our queue it is important that the lookups are fast.

To make the lookups as fast as possible we only compare hashes. This requires that each combination of pattern generates a unique hash. Each square in the 2 by 2 pattern were given 8 bits, thus creating unique hashes. The 3 by 3 patterns were given 2 bits per square, where each position in the hash can be a wall, box, floor or other. This is unique enough since we only provide deadlock patterns containing wall, box or floor squares.

### 3.6 Tunnels

To further minimize the number of states to search through we implemented macros for tunnels. A tunnel is considered to be a straight path, vertically or horizontally, and the length between the walls is one square. The tunnel macro automatically pushes a box through the tunnel. The macro is ran if a box is on a tunnel entrance and there exists no path for the player to the squares inside the tunnel. A box is considered to be on a tunnel entrance if the only direction we can push the box in is either
• into the tunnel – player standing outside of the tunnel
• directly out of the tunnel – player standing inside of the tunnel.

The search for tunnels is made only once when we initialize the game. The map is emptied on boxes and the search iterates over all squares on the map to check if the conditions for a tunnel entrance are met. All matches are saved in a separate two dimensional array of macro objects so when the program runs it just checks for every box push if the box ends up on a macro.

The macro pushes the box until either one of the following conditions is met:

• The next square is a wall or a dead square.
• The squares perpendicular to the path is not both walls.
• The current square is a goal and the next square is not a goal.

4 Results

Our Sokoban solver ended up solving slightly less than half of the given boards, 45/100. All of the concepts that were added were positive for the performance measured in number of boards that were solved. However there are a few concepts that the solver only benefits from in certain cases. In the other cases it will takes more time for the solver to perform checks, thus slowing it down. We observed that tunnels is one of these concepts, giving huge performance gains on some boards but slowing the solver down on most of the others.

5 Discussion

5.1 Performance

We feel that our implementation has performed as expected. We did not solve all puzzles within one minute because our search tree in some cases still is too big.

Solving Sokoban puzzles with an artificial intelligence is an active research field and there are many more concepts that we have not considered or implemented, mainly because of lack of time. We would implement the following concepts if we had more time.

5.2 Corral deadlock detection

Another type of deadlock is corral deadlock. A corral deadlock happens when the player is "trapped" in an area (corral) where he can still push boxes but never to a goal. This type of deadlock is recommended to prevent if implement a better Sokoban solver.
Figure 4: Example of Sokoban board in a corral deadlock

5.3 Goal packing and packing order

When moving boxes into a goal area it might be good to consider how to arrange them, especially if the goals are close to each other on the board. Wrong movements can lead to deadlocks.

6 Reflections

We started with a research to find information about different approaches and smart strategies to solve Sokoban. When the search was done we decided to have Timo Virkalla’s master thesis as an inspiration source since there were many good perspectives on different things there. We also played a couple of games ourselves to figure out smart strategies. In the end we found that all of the smart strategies we could come up with were discussed in Virkalla’s paper.

So we then decided to first start with implementing a naive Sokoban player that defined states and could move boxes. The next step was to implement the A* searches. After that we planned to implement as many of the methods for minimizing the search tree as we could until we either ran out of time or had a Sokoban solver that would suffice for the course grading. The former ended up being the limitation.

In the beginning we had discussion on using IDA* (Iterative deepening A*) and/or bidirectional search. But due to lack of time made us focus on an algorithm we master, namely the A*.

We evaluated the implementations by testing them against the test servers that were provided for us. When we could solve a couple of maps we wrote a bash script that simply tested the solver with all of the 100 maps. This gave us a pretty good idea of how the new methods affected the performance.

We ended up with implementing just one method with a couple of add-ons. So comparing the results is quite hard. All of the add-ons we implemented gave positive results on our performance. One important thing that we learned was that adding states when we push a box directly to a goal square made a big difference in most maps. Generally it is by far worth it to do more processing at each state if it discards invalid children. If we were to implement a Sokoban solver again, we would do exactly the same! Hopefully we
would have time to implement some more ??addons??.

References

