Royal Institute of Technology

(a) Caroline

(b) Paul

(c) Sindri

(d) Tobias

Artificial intelligence, dd2380

---

# Final project: sokoban

---

Caroline Laurène Kéramsi, keramsi@kth.se, XXXXXX-XXXX
Paul Lagrée, lagree@kth.se, 900629-T133
Sindri Magnússon, sindrim@kth.se, 871209-7156
Tobias Johannes, Uebbing@kth.se, XXXXXX-XXXX

October 9, 2012

# Contents

**Abstract**

asdf

# 1 Introduction

Sokoban is a popular puzzle dating back to the eighties. The original setting of the puzzle is a warehouse and the problem is to push boxes around to predefined storage locations. But the underlying problem is of course much more abstract. The rules are simple [1]:

I Only one box can be pushed at a time.

II A box cannot be pulled.

III The player cannot walk through boxes or walls.

IV The puzzle is solved when all boxes are located at storage locations.

Even though the rules are simple, the problem is quite difficult and has been proven to be $NP$-hard. This is not only due to the branching factor but also the enormous depth of the search tree [1].

# 2 Design

## 2.1 Project organisation

We started with a simple implementation which was able to solve a small number of boards. Then we applied different methods to improve it. As it takes time to get feedbacks from the server, we wrote our own bash script to evaluate the number of boards we are able to solve. This script runs our solver on 100 boards (server port 5032). We didn't want to wait too long for the results of the script so we limited the search time to 30 seconds per board.

## 2.2 States representation

A state can be represented by the position of the player and the position of each boxes on the map. Yet this representation is naive because it considers that states which differ only by a player move are different. For exemple the two following situations are considered as two different states even if we can go from one to the other only by moving the player without touching a box.

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

We want to have a new state when we push a box and not when we move the player. So instead of considering the player position, we consider the area he can reach. This area can be computed easily by a recursive algortihm. The two situations can now be represented as the same state:

```
#############
#XOOOOOO$OOO#
#OOOOOOOOOOO#
###$#########
#     $  ...#
#############

O: reachable area
X: leftest, upmost case of the reachable aread (called normalized position)
```

Two states can be compared by looking at the box position and the leftest, upmost case of the reachable area. To avoid to save too many data, we destroy the reachable area when computed, and we keep only this normalized player position.

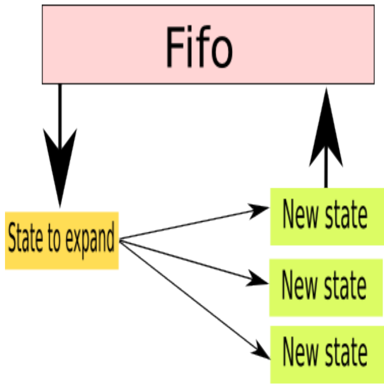
### 2.3 Initial algorithm

To expand a state, we look at each direction of each box. If a direction is inside the area that the player can reach, we look if there is a hindrance (box or wall) in the opposite direction. If this is not the case, the box can be pushed and we can create a new state.

```
#############  Here we can push down the leftest box because
#X .    $   #  the upper direction is inside the reachable
#  U        #  area and there is neither a wall nor a box
###$#########  in the opposite direction
#  D  $  ...#
#############  X: normalize player position

We can create a new state:
#############
#X .    $   #
#     @     #
### #########
#  $  $  ...#
############# X: normalize player position
```

The new state is compared with the states which have already been created. If the state existed already, we delete the new state. Otherwise we add it to the list of created states and to the fifo of the states wich must be expanded. Then we can take the next state in the fifo and repeat the procedure. This is the Breadth-First-Search algorithm.




We stop when we reach a final state (ie. a state where all the boxes are on a goal). Once we found a final state, we apply a pathfinding algortihm to build the solution string from the succession of states. With this first version we could only solve 1 board out of 100.

### 2.4 Deadlock

1 board solved (the same that with the initial algorithm) but the board is solved after around 3500 nodes (instead of 7700 before)

It is not possible to go from all states to the final state. For example if a box is blocked by a wall in both vertical and horizontal direction and is not located on a goal. Since the box is stuck it can never reach a goal and hence it is not possible to go from this state to the final state. We call such states deadlocks.

Search in the subtree below a deadlock is just a waste since it is not possible to reach the final state from there. It is therefore very desirable to be able to detect deadlock as soon as possible.

#### 2.4.1 Frozen deadlock

We say that a box is frozen if it can neither be moved in horizontal nor vertical direction. A frozen box can sometimes be made un-frozen. For example if the box is blocked by a box that is not frozen. Frozen deadlock occurs when a box is not located on a goal, frozen and can not be made un-frozen. A box can not be made un-frozen if it can not be moved in horizontal direction because of either:

I There is a wall on either the left or right side of the box.

II There is a frozen box on either the left or right side of the box.

and it can not be moved in vertical direction because either:

I There is a wall either above or below the box.

II There is a frozen box either above or below the box.

With these rules we can recursively check if a box creates a frozen deadlock. The only problem is that we can get stuck in loop if some of the boxes in the recursion checks if the original box creates a frozen deadlock. This can be avoided by treating all already checked box as a wall [2]. The initial state should not contain a frozen deadlock. For each new state after the initial state we check if the newly moved box is on goal and if not we check if it creates a frozen deadlock. Even if the box is on goal the recursive algorithm could detect a frozen deadlock if the move freezes some other box that is not on a goal. There was not enough time to implement this though.

### 2.4.2 Dead positions

There are some locations on the map such that if we put a box there it will always lead to a deadlock, we call those locations dead positions. Some of these dead positions can easily be be found before we start the solver. Then we just need to make sure that we don't move boxes to these dead positions. By doing that, we reduce a lot the search space since we remove many possibilities where we can push boxes.

## 2.5 A*

After having coded a simple BFS algorithm, we decided to improve it in an A* algorithm. Instead of using a simple fifo for the states which still have to be explored, we sort them depending on a heuristic function. This function compute the Euclidean distance (also know as 2-norm distance) of each box to a goal. Of course we do not use twice the same goal and we use the closest goal when it is possible. This implementation has improved our Sokoban solver by reducing the number of nodes needed to be explored.

## 2.6 Pathfinding algorithm

## 2.7 Search

sdf [3]

## 2.8 Pruning

To evoid to explore too many states, we decided to save all explored states. When a state is created, we compare it to the already explored states. If it has already been explored, then we do not add it to the fifo of states which will be explored. By doing so, we expect to reduce the search space. In the beginning, we used to save explored states in a simple list, but when we added a new state, we were obliged to scroll the whole list. This linear time was a huge slow-down, so we decided to use another representation for explored states.

Instead of using a simple list, we finally decided to use a hash table to represent them. It permits to explore much more nodes in the tree since it runs much faster. We do not scroll a whole list, but we compute a hash given the positions of boxes and the normalized player position. We concatenate these parameters in a single string and compute the hash using an existing function for strings in the standard library.

# 3 Results

# 4 Conclusions

# References

[1] Unknown author. sokoban on wikipedia. `http://en.wikipedia.org/wiki/Sokoban`, October 2012.

[2] Unknown author. sokoban wiki, frozen deadlock. http://sokobano.de/wiki/index.php?title=How_to_detect_deadlocks, October 2012.

[3] Timo Virkkala. Solving sokoban. Master's thesis, UNIVERSITY OF HELSINKI.