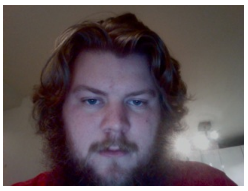
Royal Institute of Technology

(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, 900617-T251

October 10, 2012

Contents

Abstract

Sokoban is a quite a famous game invented during the eighties. Even if the rules are quite simple, it is often studied in artificial intelligence since it has been proved to be a NP-hard problem. The search space can be really huge, so everything is about finding smart ways to make computation fast and pruning the search tree as efficiently as possible. In this report, we will present the different techniques we implemented in our Sokoban solver. We will explain both why we decided to use them in our project and how well they improved our solver (in terms of time and number of nodes explored).

1 Introduction

1.1 Problem description

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].

1.2 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.

Team work on a software project can be challenging because it involves concurrent work on the same source code. That's why we decided to use a revision control software, git. Git enables us to work concurrently on the same code source more easily. It also keeps track of the history of the source code. Thus we can compare the results of the current version with an older one and we can revert the changes if the performance decreased. Our git repository is on GitHub `https://github.com/carolineKer/sokoban.git`.

2 First version

2.1 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.

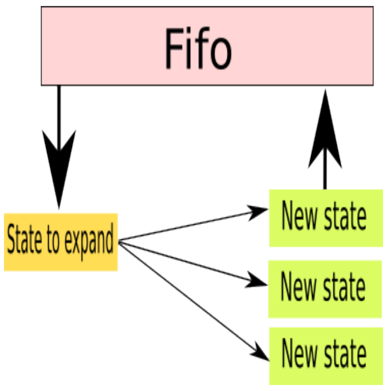
2.2 Search algorithm

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.

2.3 Pathfinding algorithm

The function findPath() in the class Ground will return the needed string containing the directions from the given start to the given goal. The function first checks whether the given start is not already the desired goal. In that case the function would just return an empty string. Otherwise findPath() calls it's helper function explorePath() that actual contains the pathfinding algorithm.

The algorithm used within the explorePath() function is a directed DFS, that means it will dynamically try to go in the direction of the goal instead of always start to explore one specific direction. There will be more detailed explanation of this process later on. The explorePath() function first checks whether it has reached the aim. If that is true it returns. If not it tries to get a accessible neighbour point to go to using the getNextCell() function. If there is no suitable neighbour available the function returns the current point. So explorePath() ensures that the next cell to explore is not the current cell and then pushes the next cell on the end of the path point vector. Afterwards it calls itself recursivley with the next point as start and the the same goal as before. In the case there is no suitable neighbour to the current cell on the board explorePath() pushes the current cell to the daedends point vector which is used by the isPassable() function to make sure that the algorithm will not get to this board cell again. Then it calls itself recursivley again with the ancestor in the path point vector. It can happen that the algorithm returns back to a path point and it came from the last possible move to make from that point. Than explorePath first removes this cell from the path point vector by testing whether the current cell is equal to the last path point saved in the path point vector before it does the recursive call.

Next the functionality of the getNextCell() function will be described in more details, since it dymancially decides which direction is best to go to reach the goal as fast as possible. First getNextCell() calculates the distance of the x and y position of the goal to the ones of the aim. Then is tests which of the four directions is accessible and stores the result in an array. Then it decides which direction to go depending on the algebraic sign of the distance and whether this direction is passable. It has a preference to first go into the x direction and then in the y direction. If none of the prefered direction is suitable it attempts to choose the first passable direction within the passable array. If all that fails the return point stays the current point and the functions returns it.

The isPassable() function that is used by the getNextCell function considers multiple situations that makes the point not passable. It takes into account whether the point was already visted before, is out of the board area, is blocked by a wall or a box or is a deadend that we have explored before.

When the explorePath() function returns the findPath() function goes through the generated path point vector and generates the path string and returns it.

A point of improvement would be to introduce a depth limit that is orientated on the manhattan distance to the goal and use iterative deepening. Additionally to reverse the order of the prefered dirctions and run the algorithm again before increasing the depth limit would improve the perfomance. But since this path finding algorithm effects the perfomance of the agent only very restricted the project team concentrated first on improving the solution finding algorithm.

2.4 Results

With this first version we could only solve 1 board out of 100.

3 Improvements

3.1 Hash table for repeated states

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.

3.2 Deadlocks

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.

```
#############  The rightest box is stucked in a corner.
#X .       $#  This state is a deadlock.
#           #
###$#########
#     $  ...#
#############
```

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.

3.2.1 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 causes 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 causes 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 causes 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. The idea of the recursive algorithm is taken from [2].

3.2.2 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. These positions does not depend on the position of other boxes. Hence they can be computed once, at the begining of the game. Then we just need to make sure that we do not 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.

Our algorithm is able to detect 3 kinds of dead positions. The first configuration is the easiest to detect. A box is blocked if two orthogonal directions are blocked:

```
     #   #  #$   #  #       The box is stuck in both the vertical and horizontal
    #$  $#   #  #$ #$# ...  direction and is not on a goal.
```

A box is also in a dead position if it is blocked on a line against a wall where there is no goal. To detect this kind of configuration, our algorithm starts from a dead position and progress along the wall until it finds another dead position (the line is dead), the end of the wall (the line is not dead) or a goal (the line is not dead).

```
    ##
     #      ####### The box is stuck in a verical or horizontal line
    $#      #  $  # which contains no goals.
     #
    ##

 ############ This line is dead (noted D).
#D-->       D#
:
 ########## This line is not dead because there is no wall
#D-->      O             in the upper direction of the case O
```

The last configuration are tunnels without goals which end on dead positions wich are blocked by three walls:

```
 #########
#D     $                 If we push a box in this tunnel, we can not push it out.
 #########


#@#########
#D     $                 Here the dead position is only blocked by two walls. The player can e
##########               the tunnel and push the box out. It is not a dead tunnel.
```

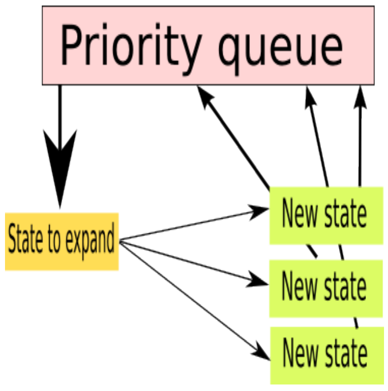
However we had to consider the special case where the player and the box are already inside the tunnel. This can happen at the begining of the game.

```
 #########
#D @    $O                Even if the case O is a dead position, we must push the box on it.
 #########
```

3.3 Search algorithm

3.3.1 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.



3.4 Search

sdf [3]

4 Results

5 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.