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, XXXXXX-XXXX
Sindri Magnússon, sindrim@kth.se, 871209-7156
Tobias Johannes, Uebbing@kth.se, XXXXXX-XXXX

October 9, 2012

Contents

1 Introduction **2**

2 Design **2**
2.1 Project organisation 2
2.2 States representation 2
2.3 Initial algorithm 2
2.4 Deadlock 2
2.4.1 Frozen deadlock 3
2.4.2 Dead positions 3
2.5 A* 3
2.6 Pathfinding algorithm 3
2.7 Search 3
2.8 Pruning 3

3 Results **3**

4 Conclusions **3**

Abstract

asdf

1 Introduction

Sokoban is an popular puzzel dating back to the eighties. The original setting of the puzzel 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.

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 brancing factor but also the enormous deepth 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. We want to have a new state when we push a box and not when we only move the player, without touching a box. So instead of considering the player position, we consider the area he can reach. This area can be computed easily by a recursive algortihm. Two states can be compared by looking at the box position and the leftest, upmost case of the reachable area.

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. 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. 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. This algorithm performs a Breadth-First-Search in the search space.

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 an 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.

2.5 A*

2.6 Pathfinding algorithm

2.7 Search

sdf [3]

2.8 Pruning

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.