





Report on Solving Sokoban

Movers and Packers

October 14, 2011

Astrid Rupp	Nikolaus Demmel
	
Edith Langer	Yogesh Garg
	

Contents

1	Representation	3
1.1	Board	3
1.2	Moves	3
1.3	Actions	3
1.4	Data Structures	3
2	Algorithm	4
2.1	Move generation	4
2.2	Forward and Backward Search	4
2.3	Find Solution	5
2.4	Hash function	5
2.5	Static Deadlocks	5
2.6	Distance heuristics	6
3	Future Scope	6
3.1	Reachable Areas	6
3.2	Freeze Deadlocks	7
3.3	Corral Deadlocks	7
3.4	Tunnels	7
3.5	Finer Optimizations	7
4	Sources	7

1 Representation

1.1 Board

We use an array of tiles (which is a class representing a field of the board) to represent the board. Position 0,0 corresponds to the bottom left corner, highest index is the top right corner. Also, position x,y on the board corresponds to the array index $y * mWidth + x$. However internally we use only indices directly.

1.2 Moves

A 'move' corresponds to a box push. And an 'action' corresponds to a step taken in Up, Down, Left or Right direction. So, in general, a move is an action (with a push) and an action may or may not be a move. A move is completely defined by the box position (of the box moved) and the direction (of the push). To apply a move, we just move the box by one position in the given direction and the player to the old position of the box. To undo a move, we move the box to the old box position, where the player must be right now, and move the player to a tile in the direction opposite to the one described in the move. Thus, the player effectively 'pulls' the box back.

1.3 Actions

As defined in the previous section, an action is a step taken in Up, Down, Left or Right direction. Its easy to detect if an action corresponds to move or not. It is done just by checking the future location of the player. If it has a box, this 'action' is a 'move' and we need to push the box. However, it is not so simple to undo an 'action' as it does not give any information about the box. If a box is facing the player we cannot say if the 'action' was a push 'move' or a simple 'action'. But since all the do and undo-action take place in pair, we save the information about an actions being a push in a boolean and use this to undo the action in future.

1.4 Data Structures

We use our own hashtable class. Hashtables are used to store boards. The size of a table is 512MB.

In our hashtable class we store 2 hashtables: one for the boards themselves and one for the depth of the boards (to be able to do the cut-off of already seen boards)

(For the search look at 2.2 and point 2.4 describes how to compute a unique hash value for each board)

2 Algorithm

2.1 Move generation

First of all, we want to generate our moves from the current state of the board. Therefore we start at the index of the player and start "visiting" tiles that are next to this tile (not going through all indices, but into all directions). If we visit a tile, we set a "VisitFlag" that we've already been there to avoid going round in circles and visiting tiles several times. If we visit a tile with a box next to it, we know that we can reach this box from our current player index. The next step is to check whether the tile behind the box is free - if yes we could push a box there. We store this possible move as index of the box plus direction of the push in a vector list and continue visiting tiles (that are not visited yet, of course). So we can never get the chance to consider impossible moves, i.e. unreachable boxes (from the current player position). Finally we have to clear the flags of the tiles again for the next iterations. Since we also do backward-search, we have to adapt this method for pulling instead of pushing boxes. Our starting point is then the solution - this means that all boxes are in the goals. Then we visit again our tiles and set the flags, but this time we check whether the box can be pulled, which means that if we visit the tile next to a box, we check whether the tile on the opposite site of the visited tile is free.

2.2 Forward and Backward Search

The search method we are using is basically iterative deepening search. We implemented a bidirectional search and that means that we are searching forward and backward. The forward search starts from the initial board. In contrast the backward search starts from the solved board. That means that every box is on a goal. We are using two hashtables. One for every search direction. In principle all possible moves for a board are generated. Then we apply one move and call the search function again and so on. We store all the generated boards in the related hashtable. But if the board is already in the hashtable we can prune that brunch. The search in general stops if the depth limit is reached.

We are doing the backward search before the forward search for some time. The reason is that the branching factor is much lower than for forward search and thus we can reach a higher depth.

Afterwards we are searching forward. Here a second possibility to stop the search appears: doing the search until a board in the backward hashtable can be found that has the same hash (and therefore should be the same board). That means we found the last part of the solution move sequence already through the backward search. The moves generated through the forward search are executed and afterwards the moves from the backward search.

2.3 Find Solution

We only store how the boxes have to be pushed to find a solution. But we don't care so far how the player comes to the boxes. After all the necessary box moves were found, the steps from the actual player position to the box that should be moved next has to be calculated. We are using a simple Dijkstra algorithm (with weight 1 for every step) which gives us the shortest action sequence for that problem.

2.4 Hash function

We need a hash function that is unique for every board. The only things that can change are the box positions and the player positions. Thus we compute the hash only with position information concerning boxes and player. We generate a random number which is 64 bit long for each field of the board one time for a box and one time for a player. The hash value can then be easily computed with bitwise XOR. We just have to check at which position a box respectively the player is and take the generated random number of that position and XOR it to the hash. The good thing is if we apply XOR with the same value again it means that the box respectively the player left that position.

2.5 Static Deadlocks

Deadlock is a board configuration from where it is not possible to reach a solution. The most basic type of deadlocks are created with only one board. To find these, we first do a backward search with only one box, starting from every goal to the entire board and get a set of tiles that are back-reachable (from where one box can be pushed to a goal). Then we do a forward search again with only one box starting from every initial box position to get a set of forward reachable tiles (a box at any of the initial positions could possibly be pushed to only these tiles). Then we take the intersection of these two sets to get a set of safe tiles. All other tiles are deadlock. Since this tiles are always forbidden, they're called Static Deadlock.

Deadlocks shown with '-' :

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

2.6 Distance heuristics

We do a breadth first search starting from the set of goals and on each tile mark the distance from the closest goal. The least number of moves (defined as box pushes) required to push all the boxes into goals is given by the sum of this distance-from-closest-goal for all the boxes' tiles.

Distances shown by numbers :

```
#####  
#109876543#..#  
#2#0####21..#  
#1$9$7$5#3#.1#  
#2#87654321..#  
#10@#7$#4##1##  
#2#$98765#323#  
#3210$##$#4###  
#432109876567#  
#####
```

Distances and Deadlocks both:

```
#####  
#-----#..#  
#-#0####21..#  
#-$9$7$-#3#.1#  
#-#87654321..#  
#--@#7$#4##1##  
#-#$98765#---#  
#-210$##$#-###  
#-----#  
#####
```

3 Future Scope

3.1 Reachable Areas

First we wanted to store only the reachable areas of the player. If we do so, we do not need to store the player index, only the information about the tile group that he is in at the moment. If a box is moved in a certain way that it blocks the player from reaching other tiles, these tiles would be stored at another group of tiles. Whenever the box is pushed back, these tiles should be joined again to the same group.

The big advantage of this approach is that we would be able to make the move generation much faster and we would not have to compare the boards for its player positions, only for the group of tiles where the player is on. Also the hashing function would be more effective.

3.2 Freeze Deadlocks

The second most important deadlocks that could be easily coded logically and cover a general class of boards is Freeze Deadlocks. These are created when two or more Boxes get alligned in such a manner that none of them could be moved. We had an algorithm for it but due to time constraints, we could not implement it in our program.

3.3 Corral Deadlocks

Corral Deadlocks are generated when there are regions created in the board that cannot be reached by the player. We did not have concrete plans to implement these in our algorithm, but this would have made our algorithm near-perfect.

3.4 Tunnels

Tunnels and in general, no-effect pushes can be discovered and taken care of and this could help us decrease the branching factor manifolds in some boards. These are very useful in certain special boards which take a lot of time to solve presently. We planned to cascade the moves that constituted a tunnel push and generate apply-move, undo-move, shortest-path between moves (moves to actions) recursively.

3.5 Finer Optimizations

If a board can be found during the forward search that is also in the backward search hashtable a solution is found. But we have to do the backward search again to compute and store the moves for the second part of the move actions. But if we store also the depth where a board is found during the backward search in the hashtable, it should be possible to generate the move sequence just out of that and we don't have to do the whole backward deepening search until we find the right board.

4 Sources

Many good thoughts are resulting from the Artificial Intelligence lectures (e.g. the forward and backward-search) and our own imaginations.

Nevertheless, we did some research on the internet. The following links were really useful regarding our task:

<http://sokobano.de/wiki/index.php>

<http://www.iis.sinica.edu.tw/~tshsu> (History Heuristic, Transposition Table, and other Alpha-Beta Search Enhancements from Tsan-sheng Hsu, Ph.D.)