# Sokoban project report

KTH DD2380 Artificial Intelligence ai10

Group: Voddler

Mikael Falgard | falgard@kth.se
Mats Jägmalm | mats.jagmalm@gmail.com
Per Jägmalm | pjagmalm@gmail.com
Robert Zetterqvist | rze@kth.se

## Summary

The project goal is to create an agent that solves as many Sokoban game levels as possible. We have created an agent in c++ that uses a depth first search algorithm with backtracking. We keep track of states in order to detect and avoid repeated states. We've also implemented a few heuristics to speed up the solver. Simple dead area checking and time breaks for backtracking. Our solver solves about 10 boards.

## Search algorithm

**Depth first search**
We use a depth first search algorithm. Before each expansion of a node (move of the player) we check that a move is valid according to the board level and also to our heuristics. When all possible moves for a state have been tried and failed we backtrack in the search tree.

**Repeated states detection**
Every time we make a move we save the state of the board. In order not to evaluate the same states in the search tree several times, we check for repeated states, every time we've made a move. If the state has already been visited, we backtrack in the search tree.

## Heuristics

**Stick to box**
In order not to explore areas where there are no boxes, we forbid moves that move the player away from a box, once he's found one. The player is allowed to go around and push a box, but must stay adjacent to the box. We allow the player to move away from a box if it's in a goal square. While this is not a perfect and generic heuristic it is effective on simple game levels.

**Reachable states checking**
This means that we check for repeated states not only for the exact position of the player, but also for positions that it can reach. If any of these states have been reached before we can stop, because then our current position will also be reachable from previous states.

**Dead areas**
Obviously  we would like to minimize the number of nodes we have to check when we're solving the puzzle. One way to do that is to mark nodes that we're not allowed to either move the player to or push a box to. This way we can save a lot of time by decreasing the number of states we have to check.
The approach we use to implement this heuristic is to keep a second board with all the static objects on the board, that is the wall and goals squares. We then go through every square on the real board before we start to solve it, if there's not a goal on that square and if it's impossible to reach a goal from that position we mark it as a "dead area" in the second board. Then when we push around a box we don't have to check if there's no point in pushing it to a square if for example the intended push would make the box end up in a corner. Note that the player is still allowed to move around in dead areas, it's just the box that have restrictions.

**Dead areas - Corners**
The easiest dead areas to find are all the corners on the board, if we put a box in a corner we'll never get it out of there and waste a lot of time, that is if there's no goal in the corner. So if the corner != from goal don't push the box there.

**Dead areas - Walls**
It's the same thing with pushing boxes next to walls that don't have any goals that are reachable along that wall. To do this we check if there's a wall square in any of the directions next to the square we're inspecting. If we for example find a wall square above the current square we look to the left and the right of the current square until we find a wall square, if the board is a 10*10 board with out any wall squares except the frame, then that's what we'll find. Then we just have to check between those wall squares if there's a goal, if not just mark them as dead area. If we instead find a wall square to one of the sides of the current square, we do the same thing except we check vertically instead of horizontally like so:

```
if(theboard[j][i] != '#' && theboard[j][i-1] == '#') {}
```

If the current square we're looking at, theboard[j][i] are at [2][1] (which would be the square to the left of the player symbol in ex 1.1) then the square next to that [i-1] is a wall square as seen in example 1.1. When we're done with our check, we would know that there's no goal neither above nor under the current square and therefore we can mark it as dead area.

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

*Ex 1.1 The red area will be marked as “dead area”, and we’re not allowed to push a box here.*

**Dead areas - Dynamic areas**
The above mentioned dead areas are all easy to detect and mark before we start to solve the board, but it’s also possible to create dead areas, such as corners. Therefor we need to check before we move a box if it will end up in a corner that’s been created by other boxes and or wall squares. If it’s a big board with many boxes this will help us avoiding many “dead” states.

**Time limited backtracking**

The idea of this approach is to give up after x seconds when no goal have been achieved and backtrack to some earlier random point and continue on another branch. This way we managed to solve some more boards. This is because our DFS is prone to waste time i deadlocks, so if we haven’t found a solution in a few seconds just drop that branch and try a new one.

# Implementation

**C++**
Our implementation is done in C++ and uses some features found in the new standard. Such as the built in functionality to calculate hashkeys of basic data types. We used g++ 4.4 and the new standard is invoked by adding the flag -std=c++0x. We used the SGI extension header hash_multimap.h for our hash table implementation. This is typically available in “ext/hash_multimap.h” with g++.
After implementing our hash table using this container we noted that the new standard also comes with the unordered_map container that should provide the same functionality.

In the header file board.h there is makros that make it possible to turn on and off different parts of the code. This is because different algorithms has proven to be more or less useful and some combinations of them are not so appropriate. The most reasonable settings we have found will be default.

**Data structure**
Our boards are stored as a vector of vectors of chars. Each char vector represents a row and together they hold a board. This data structure caused some problems when we implemented the hash table for visited states as the hash function only takes basic data types. The workaround was to convert the board to a string before calculating our hash. This is obviously ineffective but we had no other simple solution.

**DFS and Backtracking**
Our DFS tries the moves 'D','R','U', 'L’ in that order, on every depth level. If a move is permitted

and we can find no good reason not to go there, we will do that and go to the next depth level. When all 4 directions have been tried on a depth level we will backtrack 1 depth level. Our algorithm does not use recursion, and that should save some stack overhead, and add some speed.

**Repeated states detection**
Our first implementation of repeated states detection was a vector holding visited boards. We then checked this vector for every new move. This implementation has linear complexity and it causes a severe drop in node checking performance when the vector of visited states grows. The solution was to implement a hash table for visited states. Our first attempt at a hash table used the SGI extension hash_map.h. We noted after some testing that the number of visited states got so large that we got hashkey collisions which caused overwriting of our visited states as the hash_map container only supports one value to associated with each key. This had to be solved and we implemented a new hast table using the SGI extension hash_multimaps which supports multiple values per key. With this implementation our repeated states detection runs in constant time regardless of how the visited states collection grows.

**Initial one time look up of dead areas**
After parsing the board we scan it for dead areas in the form of walls and corners. We use a separate board structure to mark these squares with ‘x’. We can then use this to do more effective validation of moves once we start our search algorithm. This only applies to static walls and corners, and not e.g. a corner made up of boxes, since this requires on demand validation.

# Failed Approaches

**Unordered map hashtable**
In an attempt to replace our deprecated hashtable “ext/hash_map” with something more modern we tried the unordered map in the new C++ TR1. Unfortunately we did not have time to get it working properly. The code can be toggled on and off in the beginning of header.h. Default is off.

**Goal move**
When we thought about heuristics for the solver, one of the most obvious things to do if we want to solve the board is to tell the solver to push the boxes to a goal square instead of letting it just push it around randomly hoping that it eventually will find a goal. It sounds pretty straight forward but it gets really complex.
The idea was to save all the goal squares coordinates and when we start to push a box we would calculate the route to a goal (closest or randomly chosen). This works for a board without obstacles, given the coordinates we would know exactly how to push the box and we could even try different combinations to avoid obstacles. But for just slightly more complex boards it went out of hand and the number of special boards and cases where it wouldn’t work made us focus on other improvements instead. Just the question; *“which goal should we choose to move towards?”* have so many different answers for different boards.

## Results

We normally solve boards 1, 7, 16, 18, 58, 87, 131-135, and maybe some more..

## Build instructions

The source code is packaged in a zip file. To compile the program unzip the package and run “make”. To run the program against sokoban server and level 1, execute “client 1” or run the script compileAndRun.sh. To test many boards, for example the first 100, you can run “client_all 100”.
We used g++ 4.4 to be able to use the new c++ standard. g++4.3 could work but we have not verified that. On a Mac OSX system we also had to add the -m64 flag because the boost libraries were 64bit.

## Improvements

We clearly need to do some improvements if we want our solver to solve some more boards. One thing that probably would improve our results is a “tunnel check” that handles tunnels better than we do right now.  And also other different special areas, that we could handle better than we do now.

Our agent spends much time in deadlocks because of poor deadlock detection. This is why the time limit triggered backtrack helps us on some boards.

Another improvement would be to handle “PI-corrals” which is when we have a corral (an inaccessible area for the solver) and all the boxes on its border only can be pushed inside the corral and all the moves are possible right now, not blocked by another box.
We would sooner or later have to push a box on its border to solve it so we might as well do it immediatley. This would reduce the number of moves, and avoid “dead” positions.

```
######
#   $
#    $
#   $$
####
   #
```

*Ex 1.2 PI-Corral*