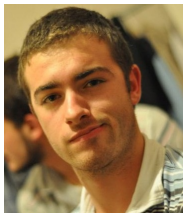
(DD2380) Artificial Intelligence

KUNGL TEKNISKA HÖGSKOLAN

A Sokoban Project

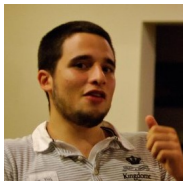
presented by

Da Lappis Cracks



Kévin Anceau *(anceau@kth.se)*

Andrea Baisero *(baisero@kth.se)*



Carlos Alberto Colmenares *(cacol@kth.se)*

Manuel Parras Ruiz de Azúa *(mprda@kth.se)*

Sokoban

Sokoban is a puzzle-style game invented in 1981 in Japan.
By now we should all know which are the rules of the Sokoban. If not, I suggest visiting the webpage '*http://www.sokobano.de/wiki/?title=The_rules_of_the_game'*.

Playing Sokoban requires a set of skills which include planning, judging capabilities, awareness of the outcomes of one's actions, and many more.
It is fairly hard to simulate these skills in a computer program, which could be said being stupid, but on the other hand is much faster than any human. This is the reason why most of the programs that try to solve the Sokoban game turn themselves to search algorithms.

Of course search by itself can't handle the enormous dimensionality that even the easier boards emanate, which is why a good search method has to be matched with intelligent data structures and algorithms that help the search all the way to the solution.
Our implementation of the Sokoban Solver tries to combine a whole series of simple but very efficient observations and methods which endow a very simple code structure with great performances.

The sources we have used to acquire information about the problem are mainly two: the sokoban wiki page and a bachelor thesis realised in 2007 by Michaël Hoste, a Belgian student. It is a well documented thesis where the author sum-up different techniques from several solvers, and even propose some new features.

Bibliography for both these sources are given in the footer of the last page of this document.

Structure of the program

Our sokoban-solver program takes into account the following techniques / methods / observations:

1. Parallel addressing structure
2. Clever node structure
3. Informed search with A*
4. Accurate heuristics
5. Deadlock detection
6. Hash Table
7. Backward Path-Recovery

1.- Addressing Structure

The most intuitive (and human friendly) way to address a cell in a sokoban board is using its absolute position, composed of its ‘x’ and ‘y’ components.
In our program we created a parallel addressing system to address the non-wall cells using only one index ‘i’; we have called it the “relative position”.
To create this relative addressing system, we started flooding the board from the point in which the man stands using a DFS to list and enumerate all the accessible cells.
We then created two “transformation matrices” which allowed us to pass from the absolute to the relative mapping notation and viceversa.

This parallel mapping system, combined with bitmap techniques allowed us to minimize memory loss due to node storage. Computational costs are also amortized by using binary operations on the bitmaps.

2.- Node structure

Usually a node of the search tree, which represents a possible state of the sokoban board, should contain enough variables to recreate the whole information about the board, including box positions, man position, walls, goals etc..
In reality, most of these information are common between all possible nodes (i.e. goal position, wall position, dimensions of the board, mapping between absolute and relative positions), and can be stored once and for all somewhere else.
The only information that each node needs to distinguish itself from all others and to still represent entirely the board are the following:

- box positions
- man position

What is interesting here is the man position. One might say that the man position should represent the cell in which the man stands in that particular node. But it is easy to see that what is important is not the actual position of the man, but the position of the cells where the man can move without moving any box. Using this interpretation helps enormously the search algorithm by creating a much more shallow search tree. This is because every “child” node consists of a box move instead of a man move, and it is easy to understand that the first kind might consist of several, even a dozen of the second kind.
These cells, the ones in which the man can move without moving any box, must be calculated at every node generation using a simple DFS algorithm. Fortunately this operation is not enormously time-wasting, and can lead to immense performance boosting.
The only comeback of this representation is that we don’t really know yet how the man got from pushing a box to pushing the next one, we just know that he could, and that he did. To recover the actual path, a path-recovery algorithm has been created which is used only once, when a solution node is found.

We have concluded that the only necessary variables for every node are the box cells, and the man-accessible cells.
These variables already represent a memory gain compared with the “whole information” method, but we have been able to waste even less memory by using bitmaps.
Having an already defined and working relative position system, we are going to use two arrays of integers, in which every bit represents a boolean value that indicates whether the relative cell contains a box or a man-accessible cell.
Using both of these methods, we have been able to store a whole node’s information in an average of 2 to 4 integers.

It’s very interesting to note that bitmap notation not only assures a minor memory usage, but also helps to calculate other important information that will later be used in the search algorithm.
For example, we can calculate which boxes can be pushed by the man in a given state by making a bit-wise ‘and’ operation between the man’s position’s bitmap and the boxes’ bitmap.
(This is actually true only because we allowed cells that contained boxes to be in the man’s position bitmap).

3.- Informed search

The algorithm chosen for searching a solution was an informed search algorithm, specifically the A* algorithm. This algorithm profit the heuristic calculations (section 7) to decide which node to expand next.

The general algorithm is the following one:

```
a_star begin
Variables used:
      q <- Priority Queue of soko_nodes
      sn <- A soko_node, initialized with the initial board
      v <- A vector of soko-nodes
      ht <- Hash table of soko nodes
Algorithm:
      introduce( q, sn )
      while no_empty( q ) do
            sn <- pop( q )
            v <- get_sons( sn )
            for each son in v do
                  if is_solution( son ) then
                        return son
                  else if no_deadlock( son ) and
not_inside_hash( son ) then
                        insert( son, ht )
                        push( son, q )
                  end if
            end for
      end while
      return "no solution found"
a_star end
```

From the algorithm it is important to emphasize the way that the sons of each node are generated: Each box in the area of the player is considerated and then a different node (son) is generated for each push that can be done to each box. This methodology makes the depth of a node in the search tree be equal to the number of **box pushes** that has been made from the initial node. Hence, the A* will find a solution that will be **optimal in the number of box pushes**.

The next table shows the throughput of the algorithm, measured by the number of generated nodes, queued nodes (nodes that were not deadlock or already in the hash), and expanded nodes (popped from the priority queue). This numbers where varying according to different number of boxes on the board, not the number of cells.

Number of boxes	Generated nodes per second	Queued nodes per second	Expanded nodes per second
6	354648.2	83039.3	63632.1
8	300244.0	66063.0	53599.0
10	170696.6	67014.6	12972.0

4.- Accurate heuristics

For estimating the heuristic of a node, two methods were used:

4.1.- Minimum distance from single boxes to goals

This heuristic measures the minimum number of pushes that are needed to take a box from any cell “i” to any other cell “j”.

The algorithm used for calculating this heuristic was a BFS made over a complex graph that takes into account the consecutive positions of the player, and therefore the solution will be the real number of pushes needed to transport a box from two cells if there where no other boxes on the board. A more detailed explanation of the graph used can be found in [1]. For building the “user-position-relative graph” it was necessary to implement an algorithm for detecting articulation points and biconnex components on the board.

Due to the fact that the calculation of this heuristic does not vary whit the position of the boxes, it is totally calculated in a matrix M which dimensions are NxN, where N is the number of accessible cells on the board (reachable by the player). This matrix is calculated before the A* search is performed and the procedures implemented for its calculation are asymptotically bounded in time by O(N^2).

4.2.- Total estimation for all the boxes

The total heuristic h(n) for a node “n” is calculated as:

$$h(n) = g(n) + f(n)$$

Where g(n) is the depth of the node in the search tree and f(n) is the **total estimation of all boxes**.

Once having the matrix M described in section “4.1”, for each node another matrix “O” of dimensions “BxG” is created, where “B” is the number of boxes on a board an “G” is the number of goals. The matrix “O” is filled whit the values found in matrix M, in such way that O[i][j] contains the minimum number of pushes needed to take the box number “i” to the goal number “j”.
Once having the matrix “O” calculated, then f(n) is calculated as a solution for the **minimum assignment problem**. The “Hungarian method” algorithm was implemented to solve such problem, and its time complexity is asymptotically bounded by O(B x G^2).

4.3.- Heuristics conclusions

Although the calculation for the total estimation for all boxes might be slow, its resulting value of h(n) tends to be very near to the real cost value h*(n) (only when combined to the heuristic that we applied for single boxes estimations), and therefore, the A* algorithms expands only promising nodes and arrives to a solution faster (even if the generation of each node is more time-comsuming).

5.- Deadlocks

Deadlocks are situations in which a solution is forever compromised. If the player arrives in a deadlock state, it's game over and he has to restart the level from a previous state (in the worst case, from the beginning).
Studying deadlock situations is of primary importance for the program: a good analysis can help cut great portions of the search tree, resulting in a much faster program.
Deadlocks come in many different types; some are easier to find and some are more difficult, and some are more common than others. The major difficulty is to detect deadlocks as soon as possible, so that we don't go too far with a compromised board. Luckily, the most common types of deadlocks are also the easiest to detect. These are the Static Deadlocks and the Freeze Deadlocks.

5.1.- Static Deadlocks

There are certain cells in the board in which a box should never be pushed. These are cells from which the box will never be able to reach a goal cell anymore. If a box arrives in one of these cells a static deadlock is triggered, and the box is doomed, while still pushable, to be restrained in a certain "deadlock zone". These cells are independent from the box disposition in a certain state, and so it's possible to calculate them just by using the walls' and goals' disposition.
In our implementation of the sokoban-solver, these cells are calculated once and for all at the very beginning, and the deadlock is studied by seeing if a box is in any of these cells.

An easy and quick way to detect static deadlock is to start from the corners. If a box is pushed in a corner, it is obviously impossible to push it away. So, unless a goal is located on this cell, a corner will always be a deadlock.
Then, we can expand our static deadlock list by looking at cells beside walls. Sometimes, when a box is against a wall, there is no way to push it away from here (see picture). To detect these cells, we just "walk" beside the wall from a corner and see if any box placed here could be pushed away, if not, all the visited cells are declared deadlocks.

5.2.- Freeze Deadlocks

Freeze deadlocks arise when one or more boxes lay in a non-goal cell and can no more be moved from that cell in any possible future state. In fact, some Freeze deadlocks are also Static deadlocks (ex. a box pushed in a non-goal corner).
These deadlocks heavily depend on the boxes' disposition, thus a preliminary study of these deadlocks is impossible: every state has to be analyzed!

Again, good fortune comes in our way for the following reasons:

1. Even though a freeze deadlock might involve many boxes in the board, a study on a very specific (and smaller) portion of the boxes will still lead to the same conclusion.
2. If a state doesn't represent a freeze deadlocks, then an immediate successor will only represent one if the move made to get from one state to another has created the deadlock itself. Thus, the last-pushed box has to be included in the deadlock, and we can take advantage of this by restraining the search to the immediate surrounding of that box.

As already said, only few configurations (with all relative rotations and reflections) exist, and so a case-by-case study is possible and efficient, especially since all relative rotations and reflections are handled intelligently by converting them to a standard direction.

5.3.- Deadlock Conclusions

These are not the only type of existing deadlocks, but most of the remaining are very hard to predict, and will eventually (within some moves) become either a static deadlock or a freeze deadlock.
The analysis that our team has made of deadlocks could be completed by including other types, but results so far have already been very promising, and the time needed for many boards has diminished by 60%, allowing us also to solve many boards which were unsolvable before.

6.- Hash Table

We have constructed a very basic Hash Table to store already expanded nodes and to avoid re-expanding them. The hash table consists of an array of linked lists which contain the already visited nodes.

6.1.- Collisions

Collisions are handled using the Direct Chaining technique, in which all entries which have the same Hash value in common are stored in the same linked list.
This technique is very simple and can also be very proficient given that the load factor of the Hash Table remains limited. For this to happen, an adjusted Hash Table size has to be chosen, and a good Hash function has to be implemented, to keep the collisions at a minimum.
For the dimension of the Hash Table, we decided to use a fairly big prime number, and chose 1000003.

6.2.- Hash Function

The Hash function we have designed makes use of all the bitmaps contained in a node.
Basically we iteratively make use of bit-wise xor and shift operations over the integers of the bitmaps. The resulting Hash function turned out to be extremely efficient, as we can see from the following statistics over a couple of the toughest boards: board number 48 and 52.

Size	1000003
Number of Nodes	6910743
Used Cells	99.86%
Avg Pointers per Cell	6.92
Max Pointers per Cell	26
Cells with 1 pointer	0.86%

Size	1000003
Number of Nodes	1712815
Used Cells	80.08%
Avg Pointers per Cell	2.14
Max Pointers per Cell	13
Cells with 1 pointer	37.84%

As we can see, most of the cells are being used, and the maximum amount of pointers per cell remains in a limited distance from it’s ideal value for both problems.

7.- Backward Path-Recovery

As we have already analyzed, our nodes don't contain directly the position of the man. Instead, a set of all accessible cells to him is stored using bitmaps.
This creates a little problem: once we find the solution node, how do we extract the solution?

The Backward Path-Recovery function allows to analyze all the nodes from the root (starting state) to the solution node/state. It works by recursively chaining the moves that are necessary to get from a father's state to a direct child. It is obvious that these moves have to avoid pushing any box since the box pushing part has already been made by the search algorithm. So, to get from a father's state to a direct child's one we created an algorithm which behaves like a BFS and recovers the shortest path the man has to follow to get from a point to another avoiding walls and boxes.

Now there is only one last problem, which are the cells the man has to move from and to?! Our nodes don't contain such information since our nodes contain only the area in which the man can move. The solution is quite simple: in every node, we added a variable that indicated which was the box that had to be pushed for last to arrive at that state.

Having all this information allows us to calculate sub-paths that go from one state to another, and then appending them altogether to create the whole final solution to the sokoban board.

Bibliography

- [1] "Jeu de Sokoban - recherche de solutions optimales" by Michaël Hoste
- [2] http://www.sokobano.de/wiki/ - Sokoban Wiki Site