Artificial Intelligence report : Sokoban solver

Ricardo Alvarez, Antoine Deblonde, Paul Joly, Arnaud Ramey; *KTH – Stockholm, Sweden*
October 2009
{ricardoa | deblonde | joly | ramey}@kth.se

***Abstract*—Artificial Intelligence is a very interesting field in Computer Science. Something fascinating for many. It is the core brain and intelligence of machines and computers. Applying it just to a simple Sokoban game was a very big challenge indeed. We want our Sokoban solver to be as fast and efficient as possible both in memory and in speed. We plan to integrate our Artificial Intelligence knowledge by using all kind of optimization techniques, and integrating a well designed heuristic. We want our searcher to be able to think by itself and solve the puzzle as if it had a brain of its own.**

CONTENTS

I. INTRODUCTION

By applying our AI knowledge, we plan to be able to produce efficient code and optimize it as much as possible for the purpose of solving Sokoban games in efficient time. From several modifications to our brute force solution to an intelligent heuristic, all this was part of a work designed to optimize the efficiency of our program.

Even using C++ was also part of our optimization plan. Creating a heuristic is also elemental in the process, since we need our searcher to reason on some decision that will eventually reduce the complexity of our program.

II. CONCEPTS

The goal of the project is to solve the maps as fast as possible so the program has to examine the least amount of possibilities as possible. So it was necessary to implement solutions that unable to decrease the number of cases examined.

A. Focusing only on the moves of the boxes

The first artifice to reduce the number of cases examined was to focus on the movements of the boxes instead of considering the movements of the man because when you consider the movements of the man you consider useless movements. It is only necessary to focus on the movements of the boxes regarding where the man can possibly move in the board. This reduces a lot the complexity of our code because it greatly decreases the amount of possibilities in our search tree. When achieving the final solution, we will need to transform the movements of the box to the movements of the man. The program's answers should be based on the man's movement, so we must do this. Making the translation is very fast because we just have to do this process only once.

B. Computing the accessible positions

Even though we only consider the movements of the box, it is necessary to compute the accessible positions for the character because he can't go anywhere. So the movements of the box are somehow limited. Thus for each situation, we analyze where the character can go and then we calculate each possible movement of the boxes.

C. Computing the movements of the man

After we have calculated all the movements of the boxes needed to solve a map, we need to compute the movements of the man between each box movement. To do this, we implement a function which take as arguments two situations of the map, with only one box movement between these situations.

The function is supposed to be able to find the localization of the move, and to print the list of movement of the man between its present localization and its new localization, given the next box movement. When we concatenate all the lists of movements between all situations' changes, we have then the whole list of the man's movements for the map, which is the result we want.

Example :

```
s:-> Situation, w=9, h=6
########
#    # .#
#    $$.#
####    #
   #@ ##
   #####

s2:-> Situation, w=9, h=6
########
#    # .#
#    $@.#
#### $ #
   #  ##
   #####

MovementNode::path_string(s, s2):
U R R U U L D
```

As you can see in this very easy example, the box has only moved one spot, but as you know, the answer must given in terms of man's movements. So we must translate this. Our method takes both situations as inputs. It detects the box movement. Since there was only one box movement, our algorithm must determine the man's steps to achieve only this box movement. Our algorithm then returns the correct string: U R R U U L D . If it was the case that there were more box movements we would concatenate any other strings of box movements with the current answer : U R R U U L D .

D. Theoretical presentation of the algorithm of resolution

Assuming that we initially consider only the movement of the boxes, the algorithm of resolution will have the form of a tree–search algorithm. Each node of the tree represents a "situation", i.e. a panel of informations about the map, giving the positions of the walls, the boxes, the slots, and the character, and maybe other informations. Considering a situation, we create its sons, each son being a situation where a different box as been moved in an allowed direction. In order to limit the depth of the tree, we do not create repeated state, that is, situations soon encountered elsewhere in the tree. The search is successful when the current situation/node of the tree is the one with all boxes in slots.

We can basically begin with a depth-first search, but we implemented also an heuristic search of A* type, using if possible the geometry of the map.

E. Theoretical presentation of the algorithm of the character's movements

The calculation of the list of movements between two situation is done by using a A* algorithm : we explore the movements of the character as a tree of positions, and the exploration is done with the help of an A* algorithm.

To describe it more precisely, we define the class object "Node", each node containing the position of a cell, the "cost" to reach it, its father, i.e. the previous node used to reach it, and its "heuristic".

- The cost $c(n)$ means the number of movements to do to reach the cell, starting from the initial position of the character, and is calculated by adding 1 to the father's cost.
- The heuristic estimator $e(n)$ is the calculated distance, in straight line and in "manhattan distance" (no diagonal move allowed), to the objective of the character.
- The evaluated cost $f(n)$ is the sum of the two previous values.

We create two lists of nodes, an "open" list, containing the nodes which are currently readen, sorted by growing value of evaluated cost, and a "closed" list, containing the nodes soon readen.

In order, the algorithm take the first node of the open list, looks at all its neighbours and checks if they are "reachable" cells (empty). If they are, for each of these cells, its checks if :

- they are soon in the closed list. If it is the case, it checks if the node in the closed list has a lowest cost. If the soon closed node has a lowest cost, nothing is done, otherwise the node is replaced by the checked node, with a new cost and a new father (the current node).
- they are not in the closed list, but they are soon in the open list. If the soon opened node has a lowest evaluated cost $f(n)$, nothing is done, otherwise the node is replaced by the checked node, with a new evaluated cost and a new father.
- are not in the lists. Then they are added to the open list, with the cost of their father incremented by one and their calculated evaluated cost.

After that, the current node is deleted from the open list and pushed in the closed list.

The operation is repeated until the first element of the open list is the destination. Then, we can get the whole path by checking recursively the different fathers, beginning with the node of destination.

III. Resolution

To solve this problem we need to implement a solving algorithm. Roughly,we have firstly implemented a deep-first algorithm ,but it turned out that this method was only working on simple boards ,so we need a more powerful resulution method that we will detail.

A. Representation of a room

Basically, we first needed to develop a parser to be able transform the test data into our own coding terms. For this, we created what we call a situation object. Our code reads each line in the board. It determines the height and width of the board by the number of lines and the line with more characters respectively. It then analyzes each character and sets in our situation object whether it is an **empty space**, a **wall**, a **slot**, a **box** or a **box in slot**. The state of our room is thus contained in an array. The length of our array is of course the width times height. All this creates our situation object.
Then we represent our man differently : we give to it coordinates so we know where it is positioned in the room.

B. Tree representation

To perform our search, we use a tree to calculate the possible movements up to the solution. We start with the initial situation given by the server as the root of the tree. Then for each situation, we calculate all the possible movements and those situations become the son of the previous situation. We thus have a tree and to perform the search, we just have to to expand the nodes we choose to expand.

In usual trees, there is a pointer towards the father. But in our implementation, every node knows the whole path to the initial situation, and does not know its father. This history is in fact the sequence of box moves from the initial situation to the given node. It is stored in a string. We can get the depth and the whole sequence of moves by parsing the string from the beginning to the end.

Example : Father :

```
*** Tree: ***
-> Situation, w=9, h=6
########
#    # .#
#    $$.#
####    #
   #@ ##
   #####
 -> Depth:0
```

Son :

```
*** Tree: ***
-> Situation, w=9, h=6
########
#    #$.#
#    $@.#
####    #
   #   ##
   #####
 -> Depth:1
 -> Diff with father:
    dir=1, new box=(5, 1)  (full= 151)
```

The son's string is 151 : it is the concatenation of :

- 1 : the direction of the last move, that means UP
- $5, 1$: the final coordinates of the moved box

C. Avoiding repeated states

When dealing with the expansion of our tree, we designed a way to avoid analyzing nodes whose situation has already be encountered. For this, we use a set of strings to check if the situation we are currently on has been encountered before. This is to avoid repeated states that would penalize a lot the performance. We basically maintain each different situation as a different string, and when two strings are equal, we know that the node has been already expanded. We use strings because it is easy and quick to compare two strings ,and it doesn't take so much memory.

1) Computing the string description of the situation: This string is calculated in the following way: You get a number for each position of the box and then you put the numbers together to form a string. The board numbers start from the northern left corner and end in the southern right corner and each cell has a value, starting with 01. But we still have to add the man's position to this string because the position of the boxes might be the same ,but the man might not be in the same place. However we only consider the accessible position of the man. To represent the man's accessible positions we add a separator "-" and a number like previously. This number represents the top-left accessible position by the man.

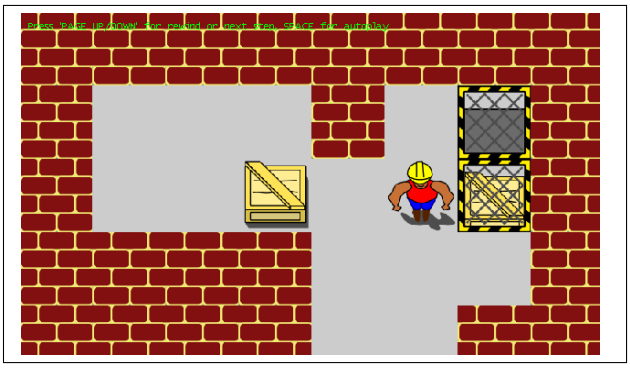

Fig. 1. Board whose corresponding string is "2023-14"

D. Winning the game

It is necessary to implement in our algorithm a function that checks for each node if the situation is won. To do that, at the beginning, we compute the string corresponding to a situation where the game is won, that is to say when all the boxes are in a slot.
Then, for each situation, we compare the first part of the string representing the situation (without the position of the man) to the string representing the game won.If they match we have found the solution.

E. Memory handling

Oour first implementation was in Java. It was unsuccessful because we had often some stack overflow or memory overflow. That is one of the main reasons we switched to C++.

In C++, we deal ourselves with memory handling : every object of the program is created thanks to the `new` instruction, and deleted with `delete`.

Example : this is how we get a situation from the server and the delete it :

```
int conn_id;
Situation* s = IO::get_board_from_server(&conn_id, 10);
Solver* solver = new Solver(s);
solver->solve_and_display(f);
delete solver;
delete s;
```

Every object is deleted mannually. This enable us to delete objects when we know we don't need them anymore. For instance, during the search in the tree for A^*, once you have expanded a node, you don't need it anymore. The sons of this node know indeed all the history, as explained in III-B.

We spent quite a lot of time to get a really well-tuned gestion of the memory. With the utility `valgrind`, you can trace the utilisation of the memory for your C++ program. Here is the output of `valgrind` for the resolution of the board 1.

```
==8581==
==8581== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 28 from 3)
==8581== malloc/free: in use at exit: 0 bytes in 0 blocks.
==8581== malloc/free: 2,222 allocs, 2,222 frees, 409,121 bytes allocated.
==8581== For counts of detected errors, rerun with: -v
==8581== All heap blocks were freed -- no leaks are possible.
```

Fig. 2. Output of valgrind on a given board. As we can see, every memory allocation is then freed.

1) Storing and accessing strings in memory: The strings used to represent each particular and different situation are stored in a set of strings (the object `set` in th STD library).

In this set, the strings are ordered alphabetically. We have chosen a set instead of a vector because storing and accessing a string in this data structure is a lot faster than the other. The complexity of a set is logarithmic both in storing and in accessing. Meanwhile the competitors complexity is linear.
So with this in hand, we can easily discard a repeated node and continue our search. This is elemental in increasing our program's efficiency, since our search avoids a great amount of repeated situation nodes.

F. Pruning the tree

One of the first and most important things to do is to check for blocked positions in our board. This helps a lot to reduce the amount of movements for each box and thus the complexity decreases a lot since it is possible to detect a dead end early.

We created a method `compute_blocked_positions` that creates an array of Booleans from our initial situation. So we go through the room, and we set a true or false value depending on the shape of the room.
First, we know that an empty slot is where we have to put the box and the initial positions of the box must not be blocking positions ,otherwise the game would be blocked from the beginning. Then, we also know that a wall is a blocked space, but if the space is empty then we need to check if the position is blocking or not.
Our method can detect two kinds of blocked positions :

- a corner, if the empty cell is surrounded by at least two walls with forming a corner;
- a "U", that is to say when a box is trapped against a wall and that it can't escape because the wall is continuous and leads to two corners. It is also important to mention that we consider it blocked if there is no slot between the two opposite walls.

So basically, in a separate method, we just check for all the situations that are blocked in a corner and then we compute the postion blocked in a "U". When traversing

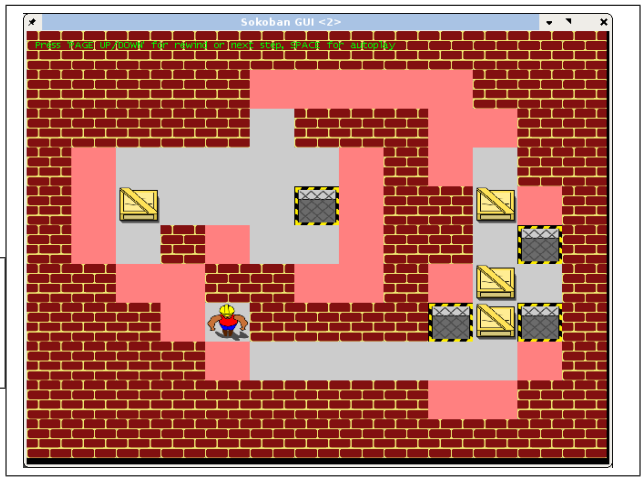

Fig. 3. Illustration of the forbidden cells in red

and expanding a tree, if in our situation node we come with at least one blocked situation, we prune the node because a blocked position means a lost game, so there is no use in continuing to expand this node. This is not very costly because the matrix with the blocking positions is only computed once, at the beginning of the algorithm.

Another calculation is made to detect other forms of blocking situation but this time, it is calculated for each situation examined. It consists in analizing the last movement to avoid to push the box in a "square". A "square" appears when four adjacent cells containing a wall or a box is created; In that case, it is not possible to move the boxes anymore so we can prune the situation and we don't need to expand it. As we said previously, this calculation is done at every step so it is a little bit time consuming but it is worth losing a little bit time for this calculation because it will detect a lot of dead ends.

We can also detect "doors" where the boxes can be blocked if we push two boxes in a door.

Let us give an example. Here is a situation, with walls (#), and the character (@). We define a door as a cell through which the movement between two rooms (several empty cells grouped) is compulsory. The doors are specified with a ($). The empty slots at the bottom of the room where the character is are not doors, because there are two possibilities to enter this room.

A property of a door is, that when you go through one pushing a box when there is a box just two slots beyond the door, the situation is blocked.

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

Example of such a blocking movement :

```
|  last direction
V
man (@) on a door cell ($)
BOX
BOX
```

G. Depth-first search

Having done this, we can finally initialize our queue of nodes to expand our tree from the initial situation. We started by using a depth-first algorithm. We expand first one of the the son of the current node then we exand another node of this node and so on. And the process continues like this as in depth-first search. We explore each situation node. We add possible movements to a queue. We nontheless check for repeated states or blocked positions to avoid expanding these nodes.
If we don't find a solution in our depths, our searcher goes back to the parents, and the process continues until it finds the correct solution. This is still without applying any heuristic.
The problem with this method is that there is no intelligence inside, it chooses a node to expand more or less randomly.

H. Heuristic and A^*

In order to improve the performance towards the depth-first algorithm it was necessary to implement a heuristic. The problem with a heuristic is that it needs to be calculated for each situation and it requires a lot of time to calculate it. So, in order to reduce the calculation required for the heuristic, we decided to calculate the heuristic only for the last box that has been moved instead of the whole situation. This is a little bit less effective than calculating for the whole situation but it gives a relevant value for the movement and helps to perform the proper movement.
The heuristic is fundamental in the efficiency of our program. It helps us optimize our program by selecting which nodes to expand first. This process selects the path to search based on the heuristic value we give to each node.
This number is based on four important factors which lead to the best option. The four factors we examine are the following:

1) the number of free slots in a board,
2) the distance of the cells to the closest empty slot,
3) the fact of pushing boxes in a cell against a wall or in a corner with empty slot,
4) the fact of pushing a box against a wall or box to avoid blocking when there is no empty slot.

All these factors are weighted to produce a number. This number represents the order in which we expand the nodes. The nodes are put into the queue depending on this number. The lowest the number means that the situation is favorable so we will expand this situation as soon as possible. So lowest numbers are placed in the beginning of the queue, and highest numbers are placed at the end of the queue.

1) Number of free slots: At each step, we compute the number of free slots in the board. From common sense, we know that if we have a lower number of free slots, (slots with no boxes yet) then we are closer to the solution because if we have already put several boxes into a slot, the resolution is probably more advanced. This means we should expand those situations instead of the ones where not any box is in a slot.

2) Distance to a slot: For each cell, we calculate the distance to the nearest slot taking into account the walls. We create a distance table at the beginning that enables us to have knowledge of the distance without recalculating it. This again allows us to give preferences to nodes because if we know that a box is close to some free slots, then we will move this one in priority to put it into a slot. This heuristic is really efficient on some maps where the slots are packed but if it's not the case or if we have to move back before moving forward, this heuristic can decrease the performance.

3) Pushing a box in a slot against a wall: When we push a box into a slot, it is already a favorable case due to the number of free slots but also due to the distance to a slot. But once in a slot, it might be useful to push it against a wall ,or even better, in a corner. Why? Because doing so, we can block and then we don't have to move this box anymore, thus decreasing the complexity.

4) Pushing a box in against a wall or a box: On a general basis, it is usually not good to push a box against a wall or a box, or even aside them because you have less freedom for your movements so when we do such a movement, we apply a small penalty on the movement. This heuristic is actually rather efficient.

At the end, after computing a number for all possible situation, we compute our heuristic value. The calculation of the heusitic is acutally difficult because it is difficult to balance every calculation. A calculation can really improve the performance on a map but on the other hand, it can decrease the performance of another so it is necessary to apply the proper weight on each calculation that will improve the overall performance.

IV. Results

A. How to launch the code

The program is written in C++. It does not require any other lib. It compile without problems on Linux, not tested on Windows. If there is a compilation problem, mail us !

Steps :

1) **Compilation :** execute the instruction `make`
2) **GUI (optionnal) :** There is also a graphical user interface (GUI). First compile it : `make GUI`.
3) **Test :** execute the program `./text.exe` and follow the instructions.
4) **Test GUI :** execute it with `./test_img.exe` followed by the number of the map you want to execute as an argument.

B. Comparison between DFS and A^*

It is necessary to compare the effectiveness of the A^* over the DFS because it should improve the number of nodes expanded but it may require more time to calculate the heuristic.

1) Map 4:

	DFS	A^*
Examined Nodes	739153	541787
Queued Nodes	194893	174331
Expanded Nodes	193100	146396
Time to solve	6sec	4sec

As you can see the heuristic is efficient

2) Map 5:

	DFS	A^*
Examined Nodes	inf	1037023
Queued Nodes	inf	555878
Expanded Nodes	inf	64232
Time to solve	inf	20sec

Again the heuristic is efficient here, because we can't solve this map without it.

As a general rule, the heuristic improves a lot the performance.

V. Conclusion

The job was a complicated process, but at the end we produced an efficient and powerful searching tool for our Sokoban solver. Our project in summary was the use of depth-first search in a tree of nodes, in which each node represented a whole situation or board with positions, type of space, etc. To our brute force algorithm, we added some methods that optimized its efficiency. We based our whole program in the movements of the boxes rather than movements of the man. We avoid repeated states and blocked situations. We also created other methods that optimized the work as a whole. Finally we designed an intelligent heuristic that would be the brain or thinking part of our program. Our heuristic avoided us expanding many nodes ,and was based on a heuristic value which decided the order of the nodes in which they are to be expanded. After testing our program with the different boards, we had successful results. We were able to resolve boards 1 through 60 in less than 1 minute. Our solution proved to be optimal.