Université Libre de Bruxelles

Techniques of artificial intelligence (INFO-H-410)

Implementation of a SOKOBAN

BALON-PERIN Alexandre
LOUMAYE Geoffroy

Academic year 2010-2011
April 22, 2011

Contents

1 Introduction

In the course Techniques of articial Intelligence, we were asked to solve a problem of our choice with one of the algorithms seen in class. We have chosen to solve a Sokoban using the algorithm A*. We found out that the single-agent search algorithm IDA* which is a variant of A* was best suited to reduce the amount of memory used. The program was implemented with the language python.

2 Sokoban

Sokoban is a puzzle video game invented by Hiroyuki Imabayashi in 1982. Sôkoban in japanese means warehouseman. The goal of the game is to put boxes on specific places. The player is only allowed to push the boxes and can push only one box at a time. The problem of solving Sokoban puzzles has been proven to be NP-hard. Several works have shown that solving Sokoban is also PSPACE-complete. Sokoban is difficult to solve because the searching tree depth is huge, the numbers of child nodes is also enormous and a good heuristic is hard to find. The best solver are not even able to solve the most complex levels.

3 Algorithm

3.1 A*

The algorithm A* is used to find the shortest path in a graph. To determine the order of the search, it uses the sum of two functions, the first one evaluates the path cost so far and the second one is an heuristic evaluating the quality of the move. The problem with this algorithm is that all the trial paths must be kept in memory. Each movement for the Sokoban is translated by the motion of a box. The program does not take into account the moves that do not move a box because they are greedy in memory and do not have any impact on the map. This mean that the program need to keep in memory the state of the map for every move of every trial path which needs a huge amount of memory.

3.2 IDA*

IDA* is an informed search based on the idea of the uninformed iterative deepening depth-first search. The main difference between A* and IDA* is that IDA* only remembers one path at a time. The principle is a depth-first search with a limiting cost. For each step of the current path the algorithm checks if the sum of the lenght of the path and heuristic is not bigger than the limiting cost. If a solution is not found, the algorithm restart with a new limiting cost equals to the minimal cost of all the path tried so far. Thanks to IDA*, the space complexity was dramatically reduced.

4 Improvement

4.1 deadzones

The first improvement to the algorithm was to mark all the spots that the boxes were not allowed to reach. These spots are the corners and the recess where no goal is placed. These are marked with 'X' on the map.

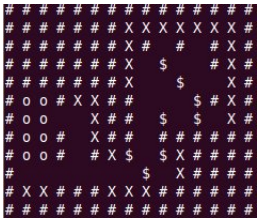


Figure 1: deadzones marked with X

4.2 access area

As previously said the program only considers the movement of the boxes, the movement of the player to reach these are not taken into account in order to increase the efficiency of the algorithm. To do this an access area map was created. This map contains all the places the player can reach without moving a box. Then by considering all the accessible boxes and the possible positions of the player around these, all the possible moves are identified.

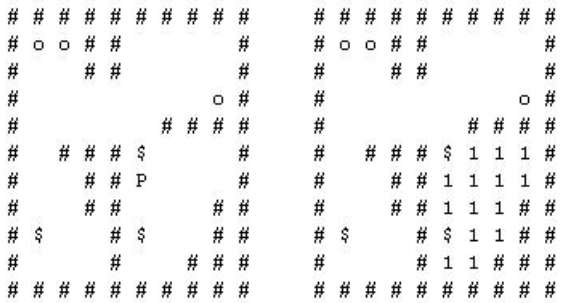


Figure 2: Left: map where 'P' marks the position of the agent. Right: resulting access area map.

4.3 state repetition

A state is defined as certain access area map and a certain position of the boxes. A problem of the program is that it visit a lot of time the same states which means a lot of computation time wasted. In order to avoid this, a state list (accesAreaList in the code) has been created.

A possible problem of this list is that a state could be visited several time but with lower cost, which means that a better path to reach this state has been found. So same states should only be rejected if the cost is bigger than the previous one. But sadly, the computation time gained with the list almost disappears if we take this into account. Consequently, a tradeoff between solution quality and efficiency had to be made. It was chosen to only test the quality a certain percentage of the time. This percentage can be adapted as a parameter of the program.

4.4 heuristic

A very simple heuristic was implemented, It decreases the path cost by a certain parameter multiplied by the number of boxes already placed. This allows the program to make deeper exploration of the most promising paths. Consequently, the program can find solution which

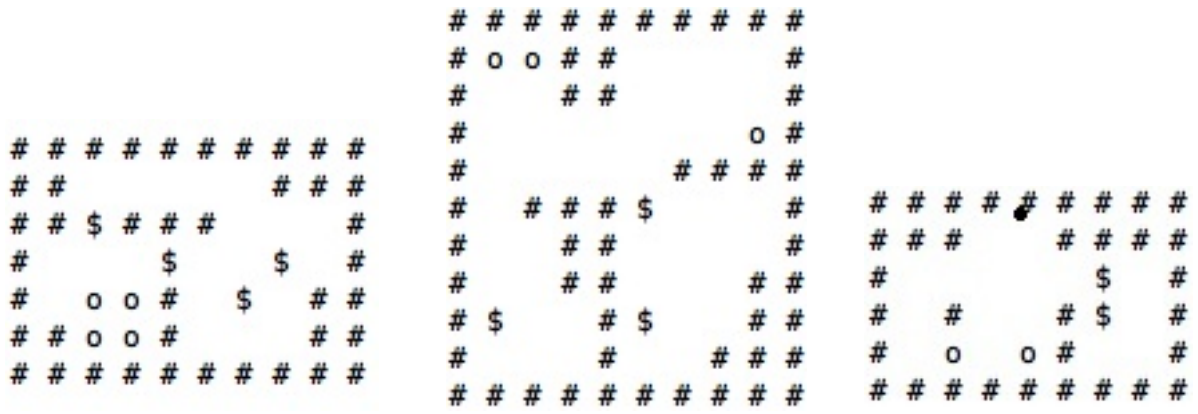
have a longer path than the cost limit. This is very interesting because the number of paths increases exponentially with the cost limit.

4.5 start costlimit

In order to evaluate the start cost limit, the sum of the Manhattan distances between the boxes and the closest goals is calculated. This gives a lower bound of the number of moves and avoids to make several useless iterations.

5 Experimentation

The algorithm was tested on four different puzzles. A menu allows the user to choose between all thes puzzles. The following pictures show these different configurations.



(a) (b) (c)

Figure 2: Here is a sample of the maps that the algorithm could solve

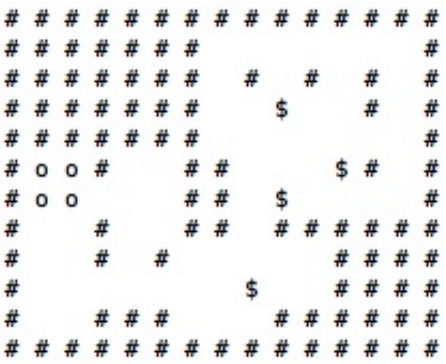


Figure 3: A map too big to be solved

The results were quite good but as discussed before a tradeoff had to be made between the efficiency of the algorithm and the quality of the solution. As a result sometimes the algorithm makes moves that human would find rather stupid. However for the puzzles tested it gets the solution in the end. As expected the algorithm solves the problems where little moves are needed

to achieve the goal. Indeed, for puzzles requiring more than 25 moves, the computation time increases exponentially. Figure 3 shows a map that the algorithm took too long to solve.

6 Conclusion

In conclusion, the program was able to resolve any sokoban needing a number of moves lower or close to 25 (or even more depending on the number of states generated), for bigger problems the computation time is just too long but this does not mean that the algorithm will not solve it.