Sokoban Puzzle Solver

By Siddhant Mahapatra – n9840371 and Alec Gurman - n9160531

Introduction

The Sokoban puzzle is based on a warehouse scenario consisting of boxes, walls and goal locations. The computer game allowed players to push boxes around the maze to correctly place them in the right locations. Despite the fact, the Sokoban puzzle is modelled as a computer game, it can be related to an automated planning problem for robots through the interpolation of artificial intelligence techniques.

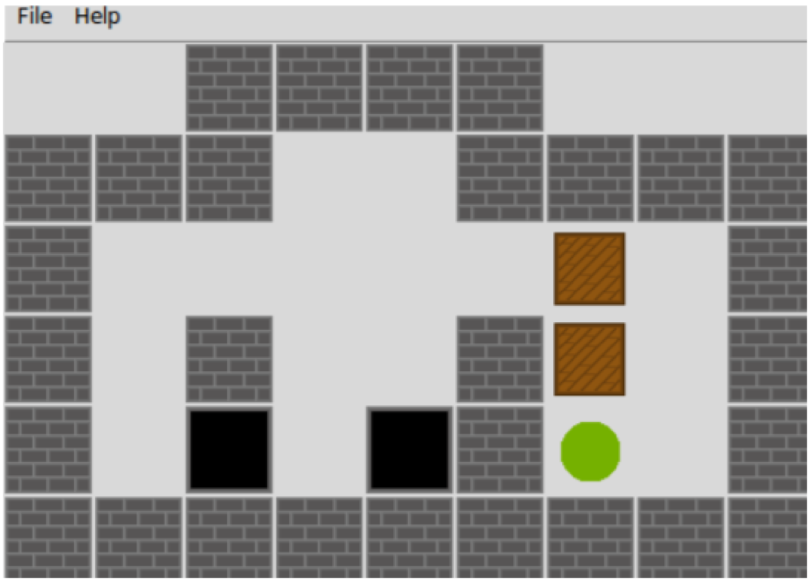

Figure 1: Example sokoban field layout

Objective

The task is to construct an intelligent agent capable of solving the Sokoban puzzle for multiple different field layouts. This should be done through the use of search algorithms with the implementation of heuristics. The Puzzle should be solved using both a single move solution and a macro move solution which will involve a sequence of elementary single worker moves.

Solution

In order to solve this particular problem, the A* search algorithm was chosen as it is a heuristic search algorithms that helps in cases where the problem at hand is NP-complete. A* is commonly used in AI engines like chess or sudoku and is well known for its speed. The basic layout for the solution of the Sokoban puzzle is to use the know position of field objects such as the walls, goals, boxes or the worker to form a series of actions that can be executed in the current state of the puzzle. The actions act as the nodes in our search problem which take the form of "Up, Down, Left, Right". Using our nodes, our search algorithm determines how the frontier should be expanded which moves our worker around the field on a path to the goal.

Object coordinate system

Our field layout is represented using a standard coordinate system with a few symbols representing different objects. Starting at the top left corner (0,0), our field is populated with objects; one such example of this is as follows:

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

Figure 2: Field Layout with symbolic representation of objects

Using this we can map out our objects and set initial value. Most of our data is stored in lists of tuple's containing the x and y coordinate of said object.

Formulating actions

Our next step is to work out a few bounds for recognizing what current available actions exist. This is done by using our worker to determine what directions we can move based of our position to any walls, our proximity to boxes and if our move will put us in a taboo cell. Taboo cells are cells defined using the following two rules:

1. If a cell is a corner and not a target the it is a taboo cell
2. All the cells between two corners along a wall are taboo if none of these are a target cell

Our actions are determined by looking at what our position will be like one step ahead, once we have determined the directions we can move in we store our actions in a list. Now that we have our actions, we fetch some nodes that can be used with our heuristic search algorithm

Heuristic Search

As mentioned previously the search algorithm used is the A* heuristic search. The A* search is simply based off a breadth-first tree search with the score being a sum of the path cost and the heuristic function. The heuristic is used to give the search a means of direction through information such as distance to the goal which allows the search to expand neighbors and with the lower scores first to find possible shorter paths. For the Sokoban application, the heuristic being used is a simple "Manhattan distance" between boxes and targets. We calculate the minimum distance between a box and the closest target however it is important to note that this does not take into account multiple boxes matched to the same target.

The Manhattan distance heuristic is simply calculated as the sum of the difference between the initial and final x and y positions of the object where the final positions are generally the target location.

$$distance\ to\ box = abs(box_x - target_x) + abs\left(box_y - target_y\right)$$

Using the above heuristic we have given the search algorithm a means of determining which objects are closest to goals and so the shortest path for the problem can be found. The A* search algorithm explores the possible solutions in a lowest-cost breadth-first manner. This means that nodes are explored from left to right with our heuristic determining which nodes in the frontier are expanded first. Unlike depth-first search we do not expand all the way to the end of the path of a single node but instead expand the nodes as a FIFO queue or a Priority queue.

Once the frontier has been expanded we pass through a result function which updates our worker positions along with any boxes it has pushed, we recalculate the current available actions to form the frontier and continue with the search. Once a solution has been found the path is returned which is simply a series of nodes (or actions in our case) the can be interpreted as directions on how to solve the particular puzzle. It is important to note that this solution follows the single move method meaning each move of the worker in our case is only by 1 cell. Another useful trick to explore large search spaces is to use macro moves.

Solving with Macro Moves

A macro action is the decision of a manager to push on specific box to an adjacent cell, within the macro action we have a sub problem which specifies if a worker move next to a specified box. The solution for a macro based Sokoban solver that has been implemented is as follows. The first thing to complete is a check to see if the worker is adjacent to a box.

1. Is the worker adjacent to a box
2. Is pushing this box possible
3. Does pushing this box cause a dynamic deadlock

Our deadlock can be described as a state where our boxes are put into an impossible format. An example of this is having two boxes up against a wall with a gap in between. Pushing a third box in this gap would cause all three boxes to lock up.

Our adjacent checker makes use of these rules to return a list of actions that would allow the box to follow suit. Following this we being our sub problem which uses the A* search to get a path from the worker to the macro endpoint. The macro endpoint is defined as are the points near a box that would allow us to move it following our rules. The result of the sub problem is used to perform our main heuristic search using the same method as our Simple single move solution. In this way our macro action becomes a sequence of elementary worker moves.

Performance and Limitations

In terms of performance by using A* search the computational time is a lot quicker when comparing to other search algorithms due to its heuristic nature. 2-3 boxes in one layout tend to compute fairly quickly (within 5 seconds) but adding a 4th or 5th box significantly increases computation time due to the increase in permutations.

Because A* follows a bread-first search format, it follows much the same limitations and therefore is poor when all the solutions have long paths. As explained above, when adding more boxes to the equation, we get longer computation times. The A* search does a good job at getting the lowest-cost solution as the first solution however as the Sokoban puzzle gets larger and more boxes are added, the permutations increase and the steps in each path become more complex which start to expose the weakness in the A* search algorithm.

Overall the A* algorithm is quick as low levels and is able to find the lowest cost solution first with a given heuristic however with increasing path complexity the A* search starts to show weakness and it would be recommended to use a better search strategy for bigger puzzle's such as an iterative depth first search approach capable of handling shallow depths as well as the deeper depths.