Thomas Waters
CAB 320 Artificial Intelligence
Project 1: Intelligent Search, Automated Sokoban Puzzle Solver
Due Date: May 8th, 2016

1. Overview

This report goes over an implementation of an intelligent puzzle solver for the Sokoban puzzle written in python 2.7. The approach uses graph theory techniques such as tree searches, graph searches, and heuristics to determine a solution to the problem. The algorithm described below is based on two external files.

The first is the *cab320_sobokan.py* file which contains the *Warehouse* class which is used to load a text file containing the Sokoban puzzle and extract the box, target, wall, and player positions and place them into tuple representations of their locations. It is also capable of taking a different representation of the puzzle and outputting a string representation of the puzzle.

The second file is the *cab320_search.py* file. This file contains many functions such as tree search and graph search algorithms. It defines an abstract *Problem* class and *Node* class which are used by the search algorithms. The *Problem* classes will be used as a basis for the implementation of the elementary action problem solver and the macro action problem solver.

2. Taboo Cells

Taboo cells are puzzle positions that cause deadlock states when a box is moved onto it. A deadlock state is a puzzle state which makes the puzzle impossible to solve.

The *taboo cells* function takes as the puzzle file name as a parameter and returns a string representation of the puzzle with all the taboo cells for the box locations marked as walls. This function is used to get the positions of the boxes that should be avoided because if the box was put into that location, then the puzzle would become unsolvable.

Corner cells in the puzzle are considered positions in which there are two adjacent or more adjacent walls where at least two of the walls walls are in neighbouring positions, thus creating a corner. If a box is pushed into one of these corner cells, then the box cannot be moved in any direction and is therefore a dead position.

The other taboo cells that are generated are the cells between two corners. If two cells are in a line, that is they share horizontal or vertical coordinates, then the space in between them are considered for being dead cells. If for every cell in-between the corner cells has at least one wall neighbouring it and the cell is not a target cell, then all those cells are considered dead. This is again because if a box is pushed onto that wall, then no matter what direction the worker pushes the box, the box will not be able to leave the wall. However the distinction is made that if a cell in between the two corner cells is a target cell, then the cells are not dead cells, because although the box may not be able to leave the wall, the box could still reach a goal state.

3. Elementary Solution

The elementary solution is a naive solution to the problem. It works by inheriting the *Problem* class from *cab320_search* so that it can be solved with one of the search algorithms in the same file. The solver works by regarding each action taken as the worker moving in one of the four cardinal directions. The goal is that the search algorithm will find the solution to the problem of

pushing the boxes to the goal by searching all his options of moving in those directions. The following will cover the specifics of the problem class for the elementary solution.

3.1 Initialization and the Puzzle States

The elementary problem class takes the puzzle file name as its only argument. The puzzle is read from the file and stores the wall positions and the taboo cell positions as defined by the *tabooCells* function.

A puzzle state is the elements in the puzzle that change in the given problem. In this implementation the state is just the worker position and the box locations. The goal state can be defined even simpler than the normal state. Since the goal state is dependent on just the box locations which is all the goal state stores. The default initial state is the worker and box positions defined by the puzzle file. The default goal state is the target locations for the boxes.

The initial state for the puzzle is the worker position and box locations as defined by the puzzle file. The goal state is just the target locations of the boxes. Both of these are also defined upon initialization of the puzzle.

3.2 Actions

The actions function determines which states the search function is able to traverse to given the current state. For the elementary solution the function looks at a current state and returns a list of the actions that the worker can take for a single move as “Left”, “Right”, “Up”, or “Down”.

The actions function worker by iterating through the four directions that the worker could possibly move and checking what is in the neighbouring cell. If the cell is empty then that direction is added to the list of actions. If the cell has a box, then it checks if the worker can move the box. This is done checking the cell past the box. That would be the cell 2 steps away from the worker. If that cell does not contain a box or is not a part of the taboo cells, then the worker can move the box and the action is added to the list of actions. If either of these two criteria is not met, then the action is not added to the list of actions.

3.3 Result

This function is called after the search is used. It generates the next state from a current state and an action that is assumed to be generated by the actions function. The next state is the worker shifted one tile in the direction of the action. If there is a box in that location, then the box is moved one tile in that direction as well.

3.4 Goal Test

The goal state is reached when each of the boxes in the current state is in one of the locations specified by the goal state which is either the target locations or the input goal state.

3.5 Path Cost

The cost of an action is the amount of steps or actions taken by the worker. Since each state moves the worker one space the path cost is simple the previous path cost plus one.

3.5 Heuristic

The heuristic is defined by the minimum distance of the boxes to a unique target plus the distance of the worker to the nearest box. The distance used is the Manhattan or grid distance, which is the non-tangential distance between two coordinates.

For the minimum box distance to the unique targets, a different approach to the one implemented could be used. For example each permutation of the box states could be calculated along with the corresponding target locations. The minimum distance could be found by finding the distance between the two sets and choosing the minimum of those values. This approach could be computationally expensive and was not the one used in this implementation. The approach was to iterate through the boxes and pick the target location that was closest and add the distance between the two to the total value. The target location and the box would then be removed from consideration for the next distance. Then finally the value of the minimum distance between the worker and the other boxes would be added to the heuristic.

3.6 Solve Elementary Actions Function

This is a function that takes the puzzle file name and an optional timeout parameter and outputs the solution to the puzzle using the elementary solution class. If the problem searches but hits the time limit for computation it returns [‘Timeout’] and if the problem cannot be solved then it returns [‘Impossible’]. Otherwise it returns the solution as a list of strings which contain the actions that the worker takes to solve the puzzle. If the puzzle is already in the goal state, then it simply returns an empty set.

This function works by first initializing the elementary solution puzzle class from the puzzle file. It then runs the iterative deepening A* search on the problem. If the problem was cut off or didn’t return a solution, the function returns the output described above. Otherwise it parses the resulting solution from the search function into the list of actions it took to get there.

4. Macro Action Solution

The macro action solution is a more informed version of the elementary solution. The macro action puzzle considers each state as a movement of a box instead of a movement of the worker. To do this, the macro class employs the help of the micro action class, which takes care of solving the problem of wither the worker can move a particular box or not.

4.1 Micro Action Solver

The micro action solver works extremely similar to the elementary actions solver with a few tweaks. For the micro action solver, the only thing that is being solved for is if the worker can be moved to a position to push the box, so both the initial state and the goal state are just the workers position. Also since we only want the worker to move the box we are asking about, when the problem is being solved, all the boxes are considered walls. This makes sure that the state in the macro action solver does not change.

The heuristic used by the micro action is the Manhattan distance from the worker to his goal location. This heuristic is both admissible and consistent. This is because the Manhattan distance would be the path cost of the worker if there were no obstacles in his way, so the actual path cost can only be greater than or equal to the heuristic.

4.2 Initialization and Puzzle States

As mentioned previously, the class uses the same initialization procedure and puzzle states are used as mentioned in section 3.1. The class also initializes a micro action puzzle solver object, to be used in section 5.3.

4.3 Actions

The macro actions considers movements of the boxes to progress the puzzle states. As such, to determine which actions are possible the algorithm must iterate through all the current box locations and then each possible direction of movement and check if those actions are possible. First we have to determine if the box has free space to move, which is done by the same procedure in section3.2 for checking box movement. Then we have to check if the worker is able to get to a location to move the box. To determine if a movement is possible the algorithm uses the micro action object. The object is initialized with the worker position, his goal location, which is the location next to the box where he would push the box from, and the current box locations. The a* search is then run, and if it returns a result then that action is added to the list of actions. This is done for all the possible combinations of boxes and movements. The action that is returned is the box that is moved, the direction that is was moved in, and the goal node of the worker path as returned from the a* search.

4.4 Path Cost

The path cost is the length of the distance travelled by the worker from an action plus the previous path cost.

4.5 Result

The result can be quickly calculated assuming that the actions taking from section 4.3 are valid. The worker position become the old box location and the box is then moved one tile the direction of the action.

4.6 Solve Macro Actions Function

The solve macro action function works in the same manner as the elementary problem solver as described in section 3.6 but is calls the macro action class instead.

5. Performance

The elementary action solver is able to solver puzzle 01, and 03, in the example puzzles relatively quickly but is not able to solve puzzle 05 is a reasonable amount of time. The macro action solver gives a huge performance benefit over the elementary solution. However because of the repeated use of the a* search in the macro action solver, the elementary solver comes to a solution faster than macro action solver for problem 03. The macro action solver has been able to solve many of the puzzles up to puzzle 35. Although not all of the puzzles have been tested, the macro action solver has provided fairly quick results to many of the problems.

6. Limitations

There are many improvements that could be made to increase the speed of the Sokoban solver. For example the puzzle does not currently consider deadlock states. It attempts to avoid the deadlock states through the use of taboo cells, but it is still possible for the solver to attempt to solve a problem while it is in a deadlock state and has no solution. This can dramatically increase the search space for complex puzzles.