CAB320 - Artificial Intelligence
Sokoban Solver

Nathan Perkins

May 8, 2016

1 Computational Environment

1. Operating System: Ubuntu Linux 15.10 was used in the creation of the Sokoban solver and all testing was undertaken in that environment
2. Python: Python 2.7.10 was used as the basis of the solution to maintain compatibility with the search algorithms already implemented.
3. Python Dependencies:
   (a) Cab320_search.py: Search algorithm library implemented by Frederic Maire
   (b) Cab320_sokoban.py: Sokoban helper class library implemented by Frederic Maire
   (c) Time: Core python library, used to time function calls. No operating system platform dependency.
   (d) os: Core python library, used to iterate over files in folders. No operating system platform dependency.

2 Elementary Solver

2.1 Structure

The utilised search algorithms in cab320_search.py required a state variable be implemented. To keep memory and therefore time to a minimum in searching for the solution space, only the dynamic elements of the puzzle were stored, including the worker and box elements. Static elements, such as the walls, taboo cells and targets were simply stored locally in the SokobanPuzzle class.
The state takes the following form

state[0] = worker : where worker is of the form (x,y). x and y are index co-ordinates
state[1] = boxes : where boxes is a tuple of tuples, taking the form $((x_1, y_1),(x_2, y_2)...(x_n, y_n))$

2.2 Updating State

In the elementary solver, an action takes the form of possible worker movement. The Cardinal directions dictate the possible actions, however not all directions in a given state are legal. Actions are checked at each state, removing any that would constitute an illegal move. Illegal actions include all of the following and are implemented in the member class function actions(state).

- Space occupied by wall.
- Space occupied by box if that box can't be pushed in that direction.
- Boxes can't be pushed in a direction if there exists no unoccupied space in that direction.

If a given action has been deemed legal and selected for the next state, the class member function results(*state,action*) returns the result of that *action*. The worker location stored in *state[0]* is updated with the new location based on current location and *action*. If the new worker position shares a location with a current box position taken from *state[1]* then that box is also updated with a move in the direction of *action*.

2.3 Heuristics

The heuristics function, denoted *h(node)* takes a potential *nodes state* variable, then takes the manhattan distance of all boxes to the closest goal state. It then adds the manhattan distance of the worker to the closest box before returning total cost. Due to the nature of the sokoban puzzle, a solution will almost never take the form of simply moving the nearest box to the nearest goal position. In all cases it is at most equal to the actual cost of solution. Therefore, the manhattan distance heuristic is an admissible heuristic and suitable to find the lowest cost solution to the puzzle.