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.
   (e) thread + threading: Core python libraries, used to ensure function calls don't go over cut-off time. No operating system platform dependency.

2 Elementary Solver

2.1 State

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 $((x_1, y_1),(x_2, y_2)...(x_n, y_n))$

2.1.1 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*.

2.2 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.

2.3 Taboo Cells

There was 2 types of cells identified that were taboo for static taboo cells. Static taboo cells occur due to the walls and other unchanging parts of a puzzle.

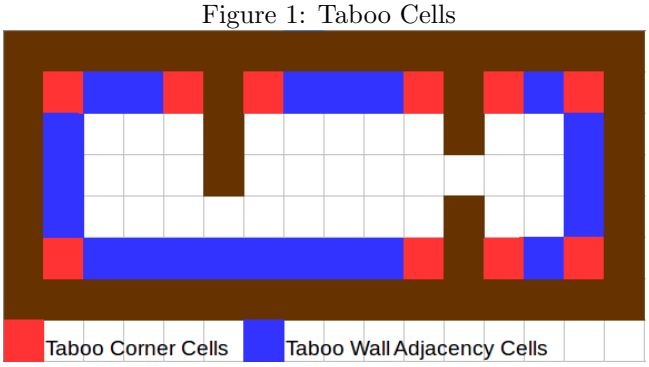
2.3.1 Corner Cells

The first type of taboo cell occurs when a corner is encountered on the walls. A worker needs to be on the other side of a box to move it in the chosen direction, therefore once a box moves onto a corner it can't leave. An example of corner cells can be seen in figure 1. To compute the taboo cells, each wall location was checked against the remainder of the walls. If the current wall contains any neighbouring wall in a diagonal location, then the two adjacent cells that share both diagonal walls are marked as taboo. After each wall is checked, the list of corner cells is checked to make sure no wall cells or target cells were marker as taboo.

2.3.2 Wall Adjacent Cells

Wall adjacent cells is another type of taboo cell, however this is harder to find. If a box moves adjacent to a wall, where there exists no openings for the worker to get behind the box, it is also a taboo cell. An example of this can be seen in figure 1. After checking for taboo corner cells, any cells between two already marked taboo corners is marked taboo if there exists no gap in the wall it borders.

Figure 1: Taboo Cells



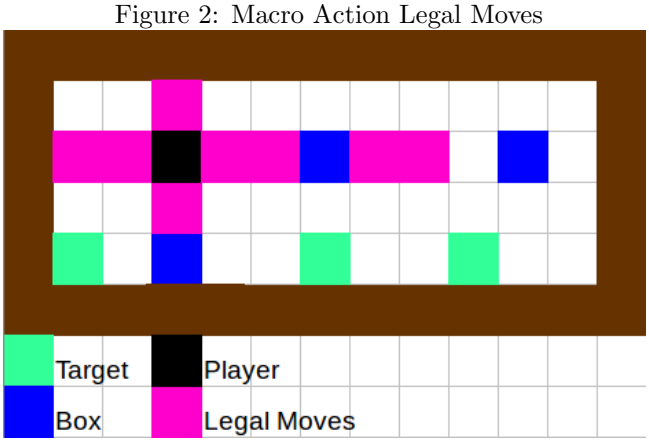
3 Macro Solver

Most features of the elementary solver were carried over to the Macro solver. These include the *state* variables, taboo cells and other static elements of the puzzle. Only 3 things changed between the implementations, the action, its resultant state update, and the heuristic.

3.1 Actions

Actions were redefined for the macro actions. Rather than, as in elementary, just moving the worker a single space, all possible spaces in a line are considered. If the worker has an unimpeded "line of sight" to a grid space in one of the cardinal directions it is considered an available action. Furthermore, if "line of sight" is blocked by only 1 box, it can push that box until the box contacts the wall, a taboo cell, or another box. An example of which can be seen below in figure 2.

Figure 2: Macro Action Legal Moves



3.2 Heuristics

Due to the creation of macro actions, the old heuristics become inadmissible and new heuristics had to be devised. For each box that isn't on target, it computes the manhattan distance of macro moves required to put a box on target. There are 3 potential states it computes for each box.

1. The box is on the target, no added cost
2. The box is directly in line, either x or y with a target, added cost of 1 move
3. The box is not in line with any targets, added cost of 2 moves

The heuristic adds all these values, then adds a further value for the worker using the same criteria as above, except between worker and box.

4 Performance

4.1 Elementary

The elementary actions solution is implemented, and can solve given puzzles. Puzzles with simple solutions can be computed quickly, however puzzles with larger complexity take much longer to solve. These harder puzzles can take minutes to fully solve. Testing was performed on a range of the warehouses sequentially, using the os library to iterate and complete each puzzle.

4.2 Macro

The macro solution currently isn't fully implemented, returning an incorrect list of actions due to an error in the *results(state,action)* function.

5 Limitations

5.1 Elementary

Taboo cells only computes the corner cells. An error exists in calculating the adjacency taboo cells. For better functionality, adding in these cells will increase speed and reduce node calculation.

5.2 Macro

Macro solutions currently does not correctly evaluate the resultant macro action, it can't therefore solve a problem correctly. The function