Sokoban Assignment

James Chen, Dongmin Park and Christopher Ayling

Queensland University of Technology

Introduction

Sokoban is a Japanese computer game created in 1981, it is a puzzle game which requires the player to maneuver crates from their starting positions to their proper storage locations. It is played on a grid of squares where each square is either wall or a floor. Floor squares may also take the form of target squares. On top of floor/target squares may be a crate or the player. To move the crates to the goal squares the player has the ability to push boxes, provided there is a floor/target space adjacent to the box in the direction the player is pushing. Directions that the player/crates can move are up, down, left and right.

Notations

The notations used to represent the warehouses in this report are as follows:

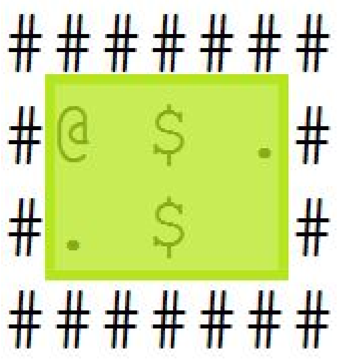
@	The player
#	A wall cell
space	A floor cell
$	A box
.	A target cell
!	The player on a target
*	A Box on a target
X	A taboo floor cell

Functions

`taboo_cells`

Identifies the taboo cells of a warehouse. A cell is deemed 'taboo' if when a box is on such a cell the puzzle becomes unsolvable. Only walls and target cells are considered when determining taboo cells. If a cell is a corner and not a target then it is taboo and if all cells between two corners along a wall are not targets then they are all taboo.

The taboo_cells function returns a string representation of the warehouse showing only the locations of the walls and the taboo cells. The only parameter is a warehouse object.



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

The `taboo_cells` function makes use of a utility function known as `get_valid_cells` which is used to find all floor cells in the warehouse regardless of if they have a box, target or player on them.

Figure 1 (left) highlights cells deemed valid by `get_valid_cells`.

The valid cells must be known to avoid including areas outside of the warehouse in the search space. Once the valid cells are found the corner cells are determined. This is done by iterating over each valid cell and marking it as taboo if it is a corner (adjacent cells on both the x and y axes) and not a target cell.

Next, if two corners are on the same row or column all the cells between them are checked to be taboo. To determine this the function then checks if there is a clear path between the corners (no walls, or targets). If there is a clear path and there is a solid wall along either side, all cells between the two corners are marked as taboo too.

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

When a cell is marked as taboo it is added to a set. This set is then used to return a string representation of the inputted warehouse by placing X characters in all taboo cell locations.

Figure 2 (Left) depicts the input (left) and output (right) of `taboo_cells`.

check_action_seq

The function `check_action_seq` is used to determine if a sequence of actions is legal. For an action to be legal it must meet the following criteria:

-Not push more than one box at a time
-Not push a box into a wall
-Not move the worker into a wall

Note: taboo cells are not considered when determining the legality of an action.

The parameters are a warehouse object and a list of actions to test, the list must be in the following form: `['Left', 'Down',...]`. If the actions are all legal than the function executes each one and returns the resulting warehouse, otherwise it returns the string `'Failure'`.

Solve_sokoban_elem

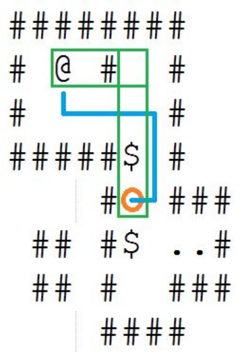
The purpose of function `solve_sokoban_elem` is to solve the puzzle using elementary actions. An elementary action is the actions that the worker takes to push the boxes to their target locations. Elementary actions of the worker are limited to moving Up, Down, Left and Right.

The function's parameter is a valid warehouse object and it returns a list containing elementary actions required to solve the puzzle.

Example output: `['Left', 'Down', Down','Right', 'Up', 'Down']`

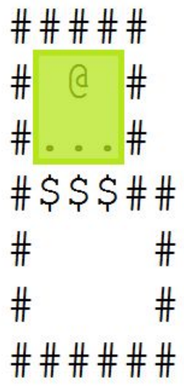
The function works by first evaluating what boxes need to be pushed where and in what order, this is done by calling the `solve_sokoban_macro` function. The function goes through the returned list of instructions and executes each one in order. To execute an action the worker must first be in the correct position, e.g. to push a box down the worker must be above the box. If the worker is not currently in the correct position a search algorithm is employed to move the worker to the required location. When the worker is in the required location the specified action is then executed. All elementary action taken by the worker to solve the problem are recorded and then returned when all macro actions have been executed.

For example, in *figure 7 (Next page)* if the specified action was to move box at location (5, 4) Up the worker would then need to find a path to below box (5, 4), location shown by an orange circle. The path is shown by the blue line. All the actions taken by the worker are appended to the list *elementary_actions*. After completing the first specified macro_action, *elementary_actions* would contain the values `[‘Down’, ‘Right’, ‘Right, ‘Right’, ‘Right’, ‘Down’, ‘Down’, ‘Down’, ‘Left’, ‘Up’]`.



The search algorithm employed for pathfinding in `solve_sokoban_elem` is an a* graph search. The heuristic used is workers manhattan distance from its goal location. The manhattan distance sum of the difference between the x and y values of two points. This heuristic is admissible; due to the worker only being able to move horizontally and vertically, the true cost must be equal or greater than the manhattan distance.

Figure 7 (Left) shows a visual representation (green outline) of the manhattan distance between the worker and the goal.



can_go_there

The can_go_there function determines whether the worker is able to access a specified location in the warehouse. The parameters are a warehouse object and a location and the return type is a boolean, indicating whether the worker can access the specified location.

The function uses a graph search algorithm to find all cells accessible by the worker. The function then checks if the specified cell is in the set and returns a boolean indicating if the specified cell is indeed accessible.

Figure 8 (Left) highlights the cells which would return true.

Solve_sokoban_macro

The function `solve_sokoban_macro` is for determining the macro actions required for solving a sokoban puzzle. A macro action specifies the location of a box and the direction it must be pushed. The parameter is a warehouse object and it returns a list of macro actions required to solve the problem or the string “Impossible” in a list if it cannot be solved.

The function makes use of a problem class called `SokobanMacro`. After creating an instance of `SokobanMacro` it is then solved using a* graph search, the heuristic used is detailed below and as it is admissible the returned solution will always be optimal. Using the solution found by the search algorithm the actions taken in each state and the boxes these actions are executed on are extracted and formatted into a list to return.

Example return value: `[ ((3,4),'Left') , ((5,2),'Up'), ((12,4),'Down') ]`

Heuristics For `solve_sokoban_macro`

The heuristic used in the a* search algorithm is the total manhattan distance of all boxes to their nearest target cell. The manhattan distance is the distance between two points measured along axes at right angles. Due to movement in Sokoban being limited to horizontal and vertical the manhattan distance will always be the minimum amount of macro moves required to push a box to a target cell. This heuristic is admissible as it returns a value lower than the true cost of pushing each box to a target cell. Using the euclidean distance would also provide an admissible heuristic but the manhattan distance is favoured due to it returning a tighter lower bound.

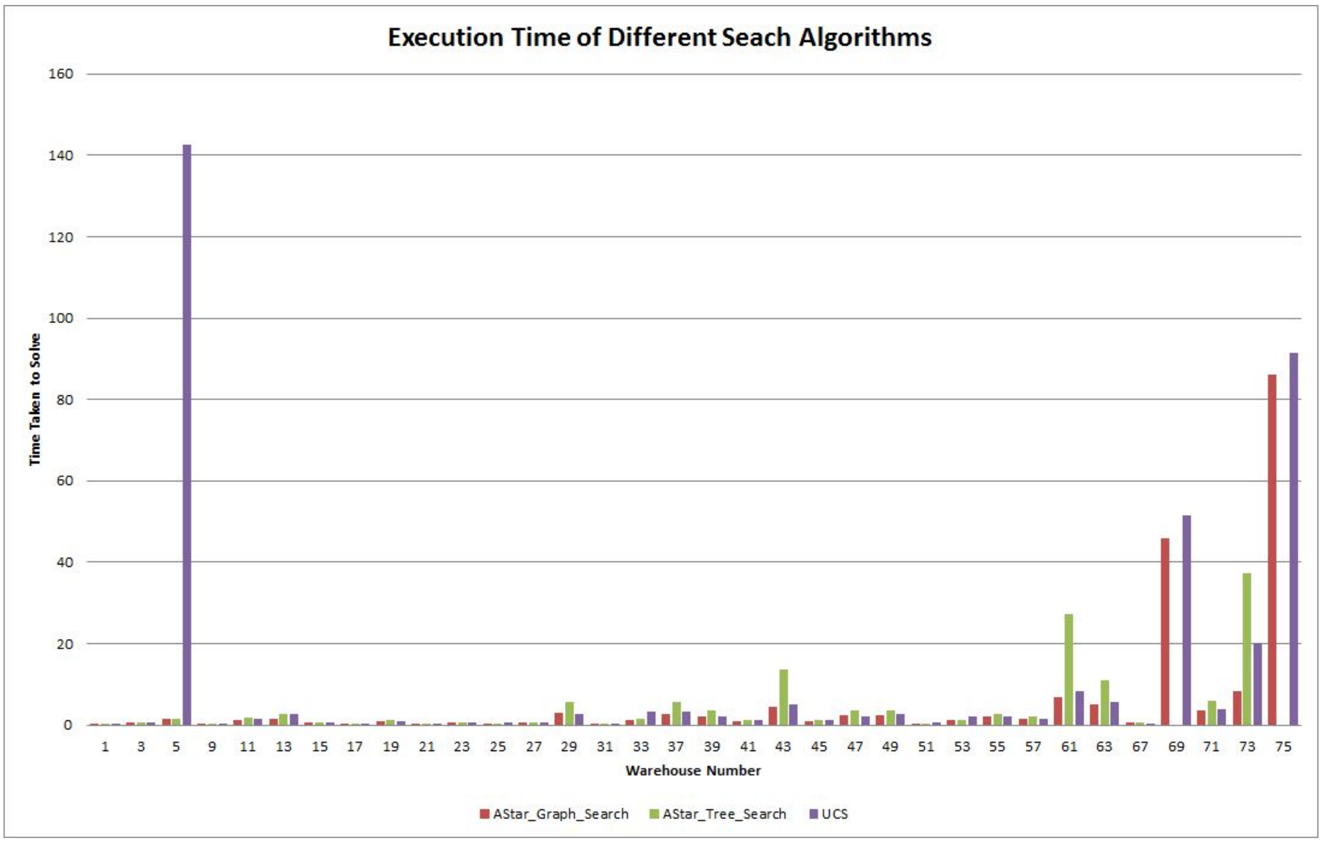
Figure 6 shows the optimal path for the box to take to reach the goal, this path requires 6 pushes to complete, this is equal to the manhattan distance between the box and the target cell.

Figure 5 highlights how this heuristic will always returns a lower bound, the heuristic calculated for this warehouse despite the true cost being much higher.

Figure 4 highlights again how the heuristic returns the lower bound. The closest target cell of each box is the same causing the heuristic to be impossibly low, this is satisfactory as it is admissible although a tighter lower bound could be determined if each box was assigned a different target cell.

It is important to note that when calculating heuristics the locations of walls and other obstacles are not considered.

Results



The above graph shows the execution time when `solve_sokoban_macro` is executed using different search algorithms. The algorithms tested with are a* graph search, a* tree search and Uniform Cost Search. Uniform cost is almost always the slowest algorithm and depending on the warehouse graph or tree search solve it in the lowest time. On average tree search is faster but more inconsistent, timing out for warehouse 69 and 75. Due to the inconsistencies with tree search we decided to use a* graph search.