CAB320 Artificial Intelligence

Assignment 1: SokobanSolver

Professor: Dr. Frederic Maire

By: Vanessa Gutierrez (n9890394), Glenn Christensen (n9884050), Marius Imingen (n9884076)

May 2, 2017

1 Heuristic

Our search algorithm of choice is A* search, because with an admissible heuristic, it will always find an optimal solution (if there is one to be found). Optimal solutions are especially desirable in a problem, like the Sokoban Puzzle, which have large search states. Another reason we chose A* is that it remembers the states which have already been explored, thus avoiding loops and infinite searches which might otherwise be included in the search tree.

The heuristic, defined as path cost plus value, used in the SokobanSolver described in this report is calculated in part by the `path_cost` and the `value` methods, named appropriately. A state's path cost is counted by the number of elementary moves the worker took to reach that state from the initial state, and the state's value is calculated by measuring the minimum distances of boxes to targets. The heuristic is admissible as it never overestimates the actual cost (in elementary worker steps) to reach a goal from a given state.

1.1 Value

The `value` method calculates a given state's value by finding the distances between the boxes and targets that are closest to each other (using the Euclidean distance), where each box and target is only included in a distance measurement once, and all boxes and targets are included in the measurement. The method accomplishes this by making a list of all box-target combinations and their distance, finding the smallest distance, adding it to the `value`, then removing all list elements that contained that target or that box. These steps are repeated until the list is empty, and the `value` is the sum of all the *minimum* target-box distances, contributing to an optimistic estimate and thus an admissible heuristic. Note that a box can only be moved to a target along a "Manhattan" distance, so using the Euclidean method to calculate the distances between a box and a target is either an exact or optimistic estimate. This, along with the ensuring the closest distances are calculated and the exclusion of the workers movements, guarantees the estimated value of moves from the state to the goal is optimistic and thus the heuristic is admissible.

Initially, we were using the combined value of a box to its closest target (discussed above) and adding the distance of the worker to its closets box thinking this would encourage the worker to move in the direction of a box, especially encouraging the worker to continue pushing the same box it last pushed as is often necessary when solving the puzzle optimally, but after thoroughly testing the search using both versions of the `value` method (Figures 3 and 4) we found that this variation of the search calculation is both slower and its solutions are not optimal.

1.2 Path Cost

The `path_cost` method calculates the amount of elementary moves the worker took to reach a given state. Initially, we counted only the amount of boxes' elementary moves to reach a given state as our path cost, but testing (Figures 1 and 2) confirmed that using worker movements as path cost sometimes completed faster searches and always provided more optimal solutions. The workers movement to between two states is calculated using the "Manhattan Distance," since the worker cannot move diagonally. A heuristic using the path cost equal to the amount of steps the worker took, and the value discussed in section 1.1 shows the best results: being a reasonably fast search, but most importantly always finding the shortest path to a goal.

2 State

This `SokobanSolver` uses a tuple of the coordinates of the worker and the boxes to represent a state of the problem, which is both an accurate and efficient representation of a state. We initially used instances of the warehouse as states, but realized this is inefficient as additional information was constantly being passed around to methods and in the search tree, such as the locations of the walls (which never change throughout the search). A more efficient representation would be to use only the locations of boxes as the state, ignoring differences in worker location which don't affect the boxes. This would shrink our search tree drastically since we are no longer expanding nodes which only vary in worker's location, and would eliminate the exploration of multiple branches which vary only in the worker's path to a box.

3 Taboo Cells

To find the taboo cells, the `taboo_coordinates` method first creates a list of all possible x and y coordinates in the warehouse. It then finds the corners in warehouse by checking the possible coordinates with the `is_corner` method, and adds the corner coordinates to the taboo list. The method accounts for a corner being non-taboo if there is a target located on that corner cell. We then check to see which corners are connected by a continuous, straight wall. Where there is no target cell between these corners, all floor cells between the corners are added to taboo. If corners are not connected by a continuous, straight wall, meaning perhaps they are in different parts of the warehouse or there is a gap in the wall, the floor cells between them are not added to taboo.

The `taboo_cells` method takes the list of taboo coordinates returned by the `taboo_coordinates` method and returns a string of the warehouse with the taboo cells marked with an 'X'.

4 `action` & `check_action_seq`

The `action` method checks the validity of all possible actions the worker can take (Up, Down, Left, Right) and returns a list of valid actions. It first checks if the cell the action would move the worker to is either wall or a box, if neither then the cell is an empty floor cell and the action is added to a list of valid actions. If the cell the worker would move to is a box, it checks if the action would move the box into another box, a wall, or a taboo cell. If the action would not move the box into an invalid space, it is added to the list of valid actions. The `action` method returns the list of valid actions.

The description of the `check_action_seq` method says that an action is legal even if it pushes a box into a taboo cell, which we found contradictory to the definition used in the `action` method described above. We created a method following this instruction: `check_taboo_allowed_action_seq`. However, we wrote `check_action_seq` to recognize valid actions in agreement with the `action` method--pushing a box into a taboo cell is not recognized as a valid action. The method goes through each action in the action sequence and checks if it is valid using the current state and the `action` method, then updates a temporary state for that action and repeats this process until all actions in the sequence have been checked.

5 Elementary Solver

The puzzle is solved with a series of elementary steps the worker makes in the `solve_sokoban_elem` method, which uses the A* search defined in the searches.py file
Sections 1 and 2 explains the heuristic for our A* search.

6 `can_go_there`

The `can_go_there` method figures out if the worker can move through the warehouse, without moving a box, to a given destination coordinate. To avoid searching unnecessarily, the method first checks if the worker is already at the destination cell. Otherwise, the method uses a miniature variation of a breadth first graph search of the empty floor cells, where the worker's location is the root node state, and each search state, or floor cell, is expanded to the surrounding cells that can be reached in one action. The search does not expand states (cells) which are boxes, walls, or already explored cells, so it never reaches the destination if it has to move a box to get there. The search finishes after finding the goal (destination) cell by only traversing empty floor cells, or after exploring every empty floor cell reachable from the worker's location and not reaching the goal (destination).

7 Macro Solver

Our `solve_sokoban_macro` returns a macro solution of box movements by calling the `solve_sokoban_elem` method and shortening the list elementary actions returned to a list of macro actions. The method makes a list of macro actions from the list of elementary actions by finding when an elementary action moves a box, and adding the box moved and direction to a macro solution list.

Ideally, `solve_sokoban_macro` would return a list of macro actions and do so faster than `solve_sokoban_elem`, but our `solve_sokoban_macro` uses `solve_sokoban_elem`, it takes at minimum the same amount of time as its elementary counterpart. This is because the way the method is written, it must first wait for `solve_sokoban_elem` to fully execute, and then represent the returned directions in a different way.

To make `solve_sokoban_macro` faster than the elementary solver we would have implemented `can_go_there` in this method. By checking if the worker can move to a cell, specifically to the cells immediately next to boxes that allow the box to be pushed, we can teleport to that cell without expanding every node that has as its state the location of the worker between the current cell and cells which allow the worker to move the boxes. Each node state would essentially be concerned with the locations of the boxes, not the worker, and node states in which boxes are not moved are almost completely excluded from the search tree. Nodes which have states that differ only in the location of the worker, and all of their subsequent child nodes, which for instance make up branches of the search tree that explore how to move the worker along different paths to achieve the same box movement, would no longer be included. This would trim down our search tree immensely, making the macro search much faster than its current performance using the elementary move search tree. We predict that the version of `solve_sokoban_macro` which implements these theories together with the ideal state described in section 2 would make the macro search complete in a fraction of the time it takes for the elementary search to complete.

Appendix

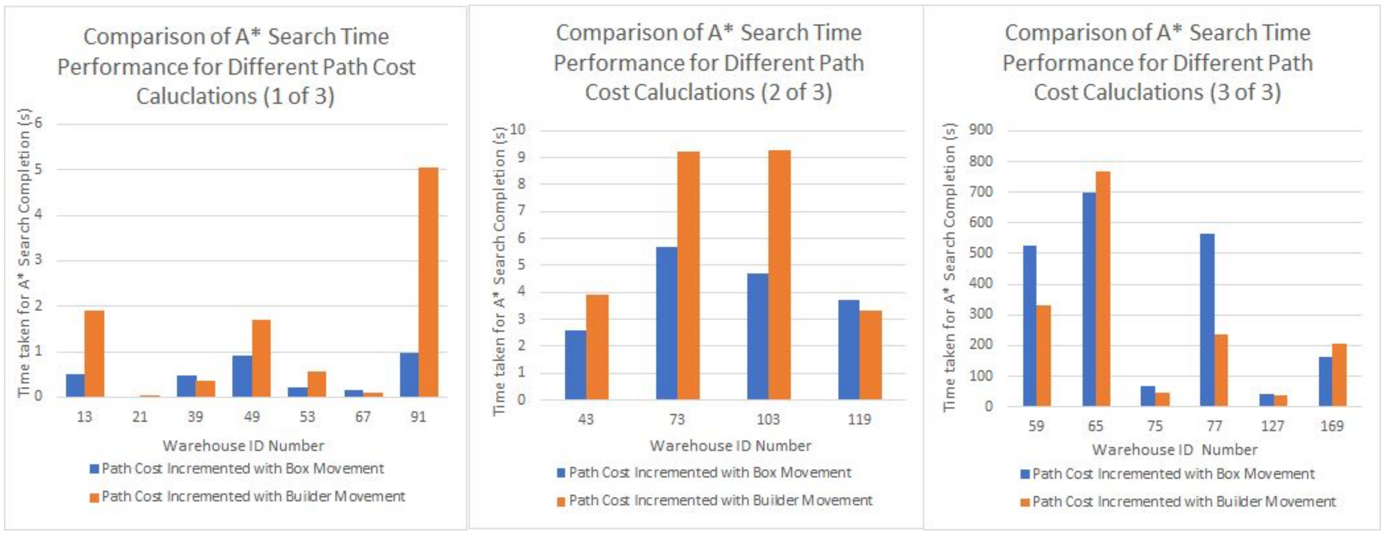

Figure 1*: The above graphs show the times taken for an A* search to solve the Sokoban Puzzle for a collection of warehouses of varying difficulty. The blue, or left-hand, bar for each warehouse shows the time taken for A* to complete where path cost was calculated by adding 1 for every step a box was moved. The orange, or right-hand, bar for each warehouse shows the time taken for A* to complete where path cost was calculated by adding 1 for every step the builder moved. The data shows that in a majority of cases, calculating the path cost using the builder's steps leads to a slower performance for A*.

*While all the data could have been plotted on a singular graph, the figure shows the data split into three graphs to allow clear visibility of value differences for all scales; the left hand graph shows puzzles solved in seconds or less, the middle shows puzzles solved in a matter of seconds, and the right hand shows puzzles solves in a matter of minutes.

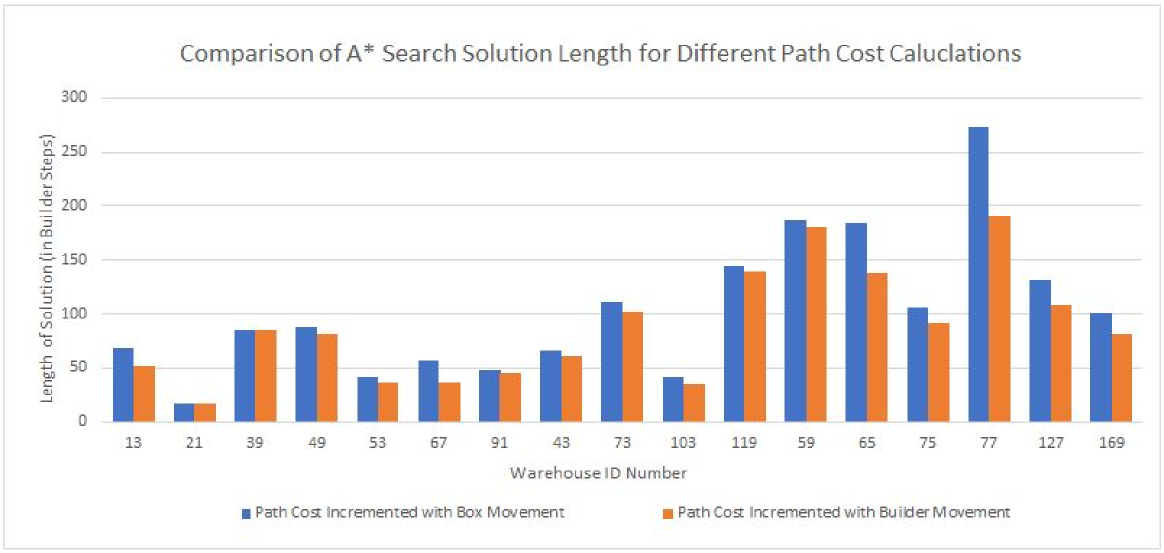

Figure 2: The above graphs show the lengths of the solution (the amount of elementary builder steps required to solve the puzzle) returned by the A* search used to solve the Sokoban Puzzle for a collection of warehouses of varying difficulty. The blue, or left-hand, bar for each warehouse shows the length of the solution where path cost was calculated by adding 1 for every step a box was moved. The orange, or right-hand, bar for each warehouse shows the length of the solution where the path cost was calculated by adding 1 for every step the builder moved. The data shows that the search using path cost calculated using box movement is not guaranteed to return an optimal solution. The data from Figure 1 and Figure 2 together show that while calculating the path cost using only the builder's steps is not always faster, it always finds a (more) optimal solution.

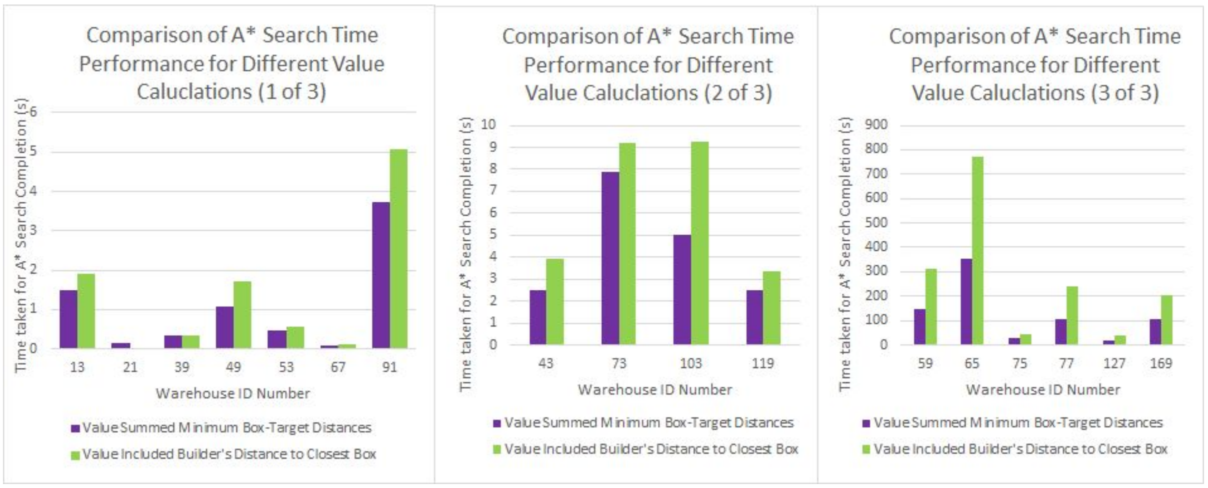

Figure 3*: The above graphs show the times taken for an A* search to solve the Sokoban Puzzle for a collection of warehouses of varying difficulty. The purple, or left-hand, bar for each warehouse shows the time taken for A* to complete where the value added to the path cost to obtain a state's heuristic was calculated by taking the sum of the distances of each box to its closest target, where no box or target is counted twice. The green, or right-hand, bar for each warehouse shows the time taken for A* to complete where the value is calculated the same way, but also adds the distance of the builder to its closest box. The data shows that in almost all cases, calculating the value using only the distances of boxes to targets led to faster performance of A* then the alternative method.

*While all the data could have been plotted on a singular graph, the figure shows the data split into three graphs to allow clear visibility of value differences for all scales; the left hand graph shows puzzles solved in seconds or less, the middle shows puzzles solved in a matter of seconds, and the right hand shows puzzles solves in a matter of minutes.

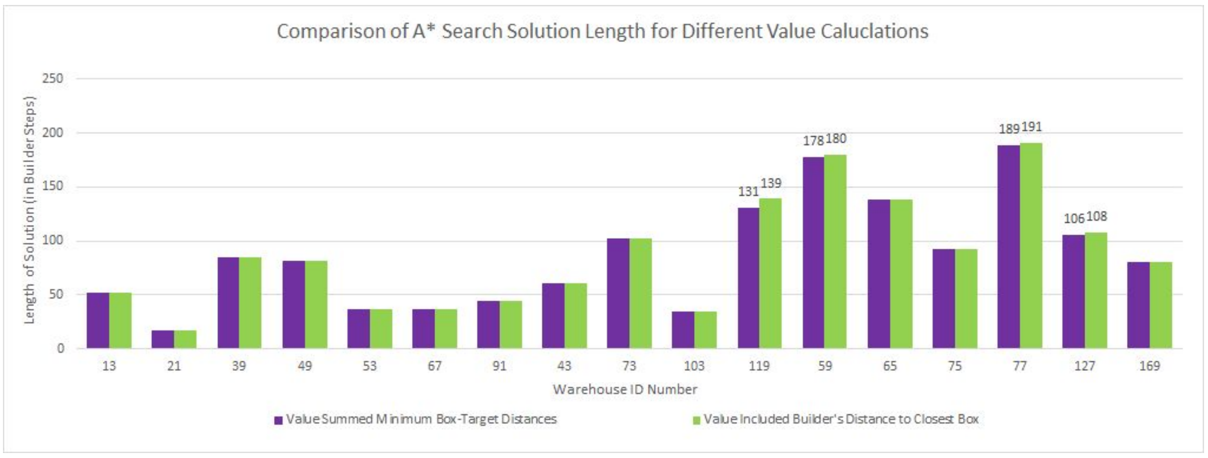

Figure 4*: The above graphs show the lengths of the solution (the amount of elementary builder steps required to solve the puzzle) returned by the A* search used to solve the Sokoban Puzzle for a collection of warehouses of varying difficulty. The purple, or left-hand, bar for each warehouse shows the length of the solution returned by A* where the value added to the path cost to obtain a state's heuristic was calculated by taking the sum of the distances of each box to its closest target, where no box or target is counted twice. The green, or right-hand, bar for each warehouse shows the length of the solution returned by A* where the value is calculated the same way, but also adds the distance of the builder to its closest box. By looking at the data provided in both Figures 3 and 4, one can see that calculating the value using only the distances of boxes to targets led to both a faster performance of A* then the alternative method and provides a (more) optimal solution.

*Data labels are provided to highlight the narrow differences in solution lengths that might otherwise easily be overlooked.