CAB320 Assignment 1: Sokoban

Intelligent Search – Motion Planning in a Warehouse

Mitchell Egan n10210776
Jaydon Gunzburg n10396489
Rodo Nguyen n10603280

1 Implementation

A weighted-*Sokoban* solver was implemented in *Python*. As a starting point, an interface consisting of a `Warehouse` class, an abstract `Problem` class, and numerous generic search and complementary classes/functions were provided. A concrete sub-class of `Problem` was implemented exclusively to handle `Warehouse` puzzle representations. All methods of the abstract base-class were overridden, except for the `value` method, which was not utilised in the solution. An additional method, `h`, was declared for use in the provided `astar_graph_search` and `astar_tree_search` functions, to calculate the value of the heuristic for a given node.

To solve a *Sokoban* puzzle, the function `solve_weighted_sokoban` is called from the implementation file, with the warehouse to be solved as its parameter. Within this, the concrete `Problem` sub-class is instantiated for the warehouse. During its initialisation, the initial and goal states, as well as the static features of the warehouse are declared as attributes in coordinate form. Aside from the attributes explicitly exhibited in the warehouse string representation, a collection of so called "taboo cells" is also kept. Once instantiated, the `Problem` sub-class is passed to `astar_graph_search`, where its methods are applied to attempt finding the optimal solution of the puzzle. The result of the *A* search* is checked to determine if no solution was found, or if the solution contains forbidden movements. If either case is true, `solve_weighted_sokoban` will return as such. Otherwise, the solution action sequence and cost will be returned.

1.1 Identification of Taboo Cells

A taboo cell is defined as a warehouse cell where if a box is pushed onto it, the puzzle becomes unsolvable. A cell is classified as taboo if it conforms to either of the following provided rules:

1. The cell is a corner (within the accessible area of the worker) and is not a target.
2. The cell is between two taboo corners. It and any other cells between the two corners, are along a wall and are not targets.

Rule two is slightly ambiguous. It could be interpreted as all cells between the corners being adjacent to a singular, continuous wall. Alternatively, it could refer to each cell between being individually adjacent to at least one wall. Following an analysis of both scenarios, the second interpretation was adopted. The rationale being that the second interpretation identifies all taboo cells which would be identified by the first; but also identifies others that wouldn't be, but which would result in an unsolvable puzzle nevertheless.
Accordingly, the process for identifying taboo cells in a warehouse is as follows:

1. Identify cells inside the warehouse (those within the accessible area of the worker).
Inside cells can be identified using a stack-based recursive implementation of the flood fill algorithm. Starting at the position of the worker, with an empty set of already-identified inside cells; coordinates of cells are checked to be within the set of warehouse walls. If they are, the set of identified cells is returned. Otherwise, given a cell isn't already classified as inside (to prevent stack overflow); the function is called recursively on the cells to the left, right, top and bottom of the current cell. This results in a "flooding" effect, where all the cells within the area of the worker are identified.

2. **Identify which of those identified inside cells are also corners.**
   For each cell in the set of inside cells, determine if the cell has at least one adjacent wall on each axis. If it does, it is included in the set of inside-corner cells.

3. **Identify taboo cells conforming to rule one.**
   Unique permutations of cell pairs are generated from the set of identified inside-corner cells. For each pair, each cell in the pair is checked to be in the set of target cells. If not, the corner is added to a set of identified taboo cells. If both corners in a pair are determined to be taboo and are aligned on an axis, the next step is executed.

4. **Identify taboo cells conforming to rule two.**
   Determine the range of cells of between the corners on their non-aligned axis. Iterate through this range and check if each cell is a target or wall (important to consider in the case of there being a segment of "outside" cells between the corners). If not, the cell is added to a buffer set of taboo cells between corners. Otherwise, cells previously added to the buffer set are voided and no more cells in the range are considered.

1.2 State Representation

An intuitive but naïve state representation would be the warehouse string representation, which is retrievable from a `Warehouse` object. However, such a representation retains static features of the warehouse, and requires complex string manipulation procedures or parsing of the string into coordinate forms; all of which are computationally wasteful. This isn't even considering that such a representation could not manage weights, since they are assigned per the ordering of the boxes in the string representation.
The most appropriate representation of state would be a container consisting of the dynamic features of the warehouse – the worker and the boxes. As for expressing the goal state in this format, the worker can remain empty (note that they could be in any cell), and the target cells can be substituted for the boxes. Set equality between the boxes of a state and the boxes of the goal state can be used to test for the goal, as such an equality disregards ordering (consider that any box could be on any target).
A suitable container would be a collection, such as a `tuple`. A `tuple` has a minimal footprint and would ensure no change in its items' state – preserving a warehouse state when determining the next. However, a `class` may be preferable to allow state-variable storage and access as attributes, despite its larger impression. Fortunately, *Python* has a hybrid collection type – the `namedtuple`; which allows for item access as attributes, whilst retaining immutability and a smaller footprint. Accordingly, the conceived state representation was:

$$\text{state} = \text{State}(\text{worker: } (x, y), \text{boxes: } ((x, y), ...))$$

where State is the `namedtuple` sub-class, worker & boxes are the sub-class attributes, and x & y represent coordinate values.

1.3 Heuristic

The value of the solver's heuristic function (`h`) for a given node is determined by firstly calculating the *Manhattan distance* (taxicab metric) between each box and each target (box in goal state) of that node's state. The formula for calculating the Manhattan distance between two points is expressed as:

$$d = |x_1 - x_2| + |y_1 - y_2|$$

The distance is multiplied by the sum of its corresponding box's weight and the base movement cost (1). Then for each box, the minimum of its distances-to-targets is identified. Subsequently, all the minimum box-to-target distances are summed to give the heuristic value:

$$h = \sum_{i=0}^{n_{\text{boxes}}} \min(\mathbf{d}_{\text{box}_i})$$

The conception of this heuristic followed consideration of search optimality from the very beginning. In the case of an *A* tree search*, optimality requires admissibility. That is,

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to the goal. Put simply, $h(n)$ must be the cost to the goal when the problem is relaxed (constraints removed). For this heuristic, the relaxed problem eliminates the distance required for the worker to walk to the boxes. Additionally, it allows for the worker walking & pushing boxes through walls/other

boxes and allows more than one box to be allocated to any single target. No matter the state of the node, the heuristic is guaranteed to be less than or equal to true cost to the goal.
For *A* graph search*, optimality requires consistency (which also implies admissibility). Consistency meaning that the heuristic value is less than the true cost for each arc in the graph. This indicates that the combined heuristic value and path cost never decreases, that:

$$h(x) \leq c(x, y) + h(y)$$

where x & y are any nodes (given x precedes y), and c is the path cost between them. The path cost between two nodes is the sum of the action costs between the nodes' states: $1 + \text{weight}$ when pushing a box, otherwise just 1. Given this, the heuristic can be proven to be consistent as:

When not pushing a box - the path cost is 1 and the heuristic value remains the same,

$$h(n_{i-1}) \leq c(n_{i-1}, n_i) + h(n_i)$$
$$h(n_{i-1}) < 1 + h(n_{i-1})$$

When pushing a box closer to its nearest target - the path cost is $1 + \text{weight}$ and the heuristic value decreases by $1 + \text{weight}$,

$$h(n_{i-1}) \leq c(n_{i-1}, n_i) + h(n_i)$$
$$h(n_{i-1}) = (1 + \text{weight}) + (h(n_{i-1}) - (1 + \text{weight}))$$

When pushing a box away from its nearest target - the path cost is $1 + \text{weight}$ and the heuristic value increases by $1 + \text{weight}$,

$$h(n_{i-1}) \leq c(n_{i-1}, n_i) + h(n_i)$$
$$h(n_{i-1}) < 2(1 + \text{weight}) + h(n_{i-1})$$

2 Testing Methodology

To ensure the solver was functioning as intended, a suite of unit tests was implemented using the `unittest` module and covesr the solver's primary functions. These tests could be run by executing the implementation file. For collaboration, the project files were kept on *GitHub*; to guard against functionality-breaking changes, the test suite was automatically run via *GitHub Actions* during pushes to the repository. Following are the four test classes and their methodologies.

2.1 TestTabooCells

A specific test warehouse was created for the purpose of testing the identification of taboo cells (refer to Figure 1). It was designed to verify all the conditions for both taboo cells and non-taboo cells.

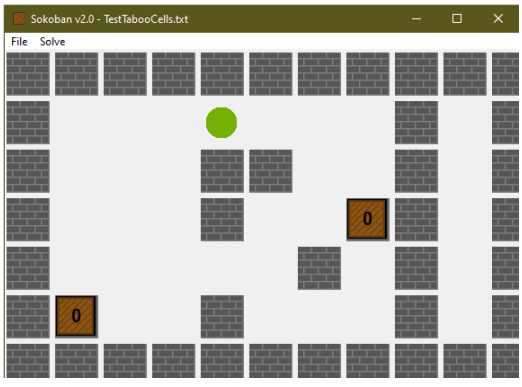

Figure 1: Taboo Cells Test Case

The expected answer to this warehouse was determined manually, without aid of the solver as to not introduce confirmation bias. The test was performed by asserting that the string returned by `taboo_cells` was equal to the expected answer string.

2.2 TestSokobanPuzzle

Testing of the *Sokoban* puzzle class was performed on one of the example warehouses, warehouse 8a. For each of the methods within `SokobanPuzzle`, multiple sub-tests were created to test each of the possible cases for that method. For example, in `actions` there were sub-tests for a wall adjacent to the worker, a wall adjacent in the same direction as a box adjacent to the worker, a taboo cell adjacent in the same direction as a box adjacent to the worker, and a box adjacent in the same direction as a box adjacent to the worker. The tests were performed by asserting each of the methods' sub-tests' actual answers were equal to their expected answers.

2.3 TestCheckElemActionSeq

Testing of the `check_elem_action_seq` function was also performed on warehouse 8a. Likewise, multiple sub-tests were created for the worker walking into a wall, pushing a box into a wall, and pushing a box into another box. Again, the tests were performed by asserting each of the sub-tests' actual answers were equal to their expected answers.

2.4 TestSolveWeightedSokoban

Warehouses 09 and 5n were used as inputs to test the `solve_weighted_sokoban`, 09 being solvable with a known solution cost and 5n being impossible. A sub-test was created for each, and each actual solution cost answer was asserted to be equal to the expected solution cost answer.

3 Performance and Limitations

3.1 Comparison of Analysis Times and Costs

Accompanying the provided interface was a collection of 6 example solution run times and expected costs for different warehouses. Table 1 illustrates that the solver implementation performed as expected, and occasionally better than the example times given. Although this could be solely due to the differences in testing hardware. Nonetheless, the times are in the same order of magnitude and can give, albeit prone to variability, some comparison of performance. The cost for each warehouse was the same, showing that the implementation successfully found the optimal path.

Table 1: Analysis Time and Cost Comparison

Warehouse	**Comparison Example**		**Our Implementation**	
	Analysis Time [s]	*Cost*	*Analysis Time [s]*	*Cost*
7	300.645556	26	172.69176	26
9	0.009575	369	0.003492	396
47	0.122561	179	0.100814	179
81	0.17024	376	0.299447	376
147	197.910769	521	205.869627	521
5n	1.504564	Impossible	1.371451	Impossible

3.2 Heuristic Limitations

A search* is only as good as its heuristic; underestimating a heuristic as close to the true cost of an optimal path guarantees admissibility, whilst ensuring high search efficiency. For this reason, ideally, the *Manhattan distance* between two points should consider walls and return a constrained *Manhattan* path through the warehouse between boxes and targets. To implement this however, the program would be essentially performing an additional search algorithm such as Breadth-First Search, just to find this ideal path. Another consideration would be to consider that a box can only be assigned to a single target, and vice versa. A combinatorial optimisation algorithm such as the Hungarian method could be utilised to determine the optimal box-target combination to ensure minimal cost.

Considering the heuristic is purposed with making the search more efficient, it may (dependent on problem size) become redundant to utilise either of these methods; as they require much greater computation power just to calculate the heuristic. For this reason, neither a constrained Manhattan distance calculation nor an optimal box-target combination for minimal cost were implemented; creating a sub-optimal path estimation but with a computationally cheap calculation.