IT3105 Module 1

Implementing and Testing the A* algorithm on a Navigation Task

Justification of programming language choice

We have decided to solve this module using Python as programming language for three main reasons;

(1) Python is the language we are both most comfortable with,
(2) The expressiveness of Python clearly demonstrates the steps in an algorithm,
(3) Python was recommended to use because the last project will include a Python framework for neural nets.

For the graphical user interface we have decided to use Pythons Tkinter framework.

Classes, methods and data structures

This project consists of six classes; Astar, AstarMod1 (subclassing Astar), State, StateMod1 (subclassing State), Board and Gui. Due to the way Tkinter's event loop works we had to separate the A* algorithm into two parts.

Initialization

The first part is the initialization part which is called in the Astar class' constructor. During this phase four data structures (and one counter to keep track of statistics) are created: *closeddict, opendict, closedlist* and *openlist*. The *closeddict* is a dictionary and keeps track of all the search nodes that have been expanded. The entries in the dictionary take the form *{hash(node): node}.* The built-in *__hash__* function has been overridden to hash a string with the coordinates of the node separated by a comma, such that (x=1, y=11) is not equal to (x=11, y=1), which we learned the hard way. This is done so that when the algorithm generates a search node that is already in the set of expanded nodes, we can extract that node and check if we have found a shorter path to it.

The *opendict* data structure is also a dictionary and is used for fast access to elements in the *openlist*, but we have to use the list structure to make the priority queue. We have a dedicated function to create the list structure so that it creates an ordinary array if the selected search algorithm is *A** or *DFS,* or a *deque (double-ended queue)* used as a stack for *BFS*. In keeping with this style we have search type specific methods to add and remove elements from the *openlist*; *pop_from_open* returns *heappop(openlist)* for A*, *openlist.popleft()* for BFS and *openlist.pop()* for DFS, while *append_to_open* uses *heappush* for A*, and *append* for both BFS and DFS. When the list is sorted as a queue the objects are compared using the built-in *__eq__, __lt__* and *__gt__* functions. These are overridden in the abstract State class, as they are general, and all they do is simply to compare *f* values. In the case of *__lt__* and *__gt__* they also take into account the *h* value if the *f* values are equal.

The initial search node is then generated by calling *generate_initial_searchstate*, which has to be overridden in problem-specific subclasses of Astar. In the case of module 1 it is overridden by a method that calls a method in Board that creates a StateMod1 object, which is a problem-specific subclass of State. Its *g*-value is set to zero as it is the initial state. Next, its *f*-value is set via a call to the *update_f* function. This object is then added to *openlist* via the *append_to_open* function and the support structure *opendict.*

Agenda loop

The next part is the part one would usually go through the agenda loop until a solution is found. Instead, this is done by a function in the Gui class called *do_one_step*, which calls the Astar object's *do_one_step* function, which does the following:

(1) Check that *openlist* is not empty
(2) Pop a search node from the *openlist* using *pop_from_open* and remove it from the *opendict* support structure
(3) Check if it is the goal state. If it is; run the method *find_path* which follows the "parent" pointers and puts them in a list and reverses them, then returns the list. If it is not the goal state, continue.
(4) Put search node in *closeddict*
(5) Generate successors of search node by calling *calculate_neighbors*, which is specific for each problem so it is in the subclass of State.
(6) For each successor
    a. increase statistics counter
    b. If it is in *opendict* or *closedlist* it fetches the preexisting state (the state with identical x, y values) and uses that
        i. It is then put in the search node's children list
    c. If it is not in either of the data structures, the function *attach_and_eval* is called to update the child to point to the search node as its parent and its *g* and *f* values are calculated.
    d. Else if we have found a shorter path to said child, we *attach_and_eval.* If in *closedlist*, we *propagate_path_improvements*.

It then returns the path from the search node to goal. This is not strictly part of A*, but is necessary for visualization. This part is then run again and again until the goal state has been found.

Astar class

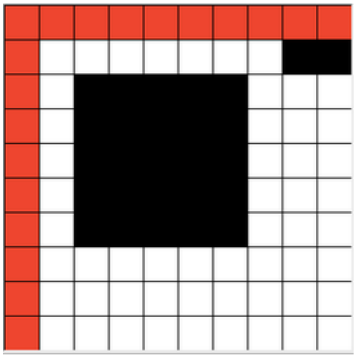
The Astar abstract class contains methods that are general for all A* problems. The constructor initializes all data structures as mentioned earlier, *find_path* returns the path from goal to node, *attach_and_eval* changes the parent of the child to the current node and recalculates *f, g* and *h*, and *propagate_path_improvements* that iterates over all children of a node and updates their values as well as calling itself with the child as an argument if it has found a better path. The subclass of Astar, called *AstarMod1*, implements all the problem-specific methods; *arc_cost* which returns the cost of moving from one state to another (in

this case it returns 1), *append_to_open, pop_from_open, generate_initial_searchstate* and *generate_openlist*, which have all been described earlier in this document.

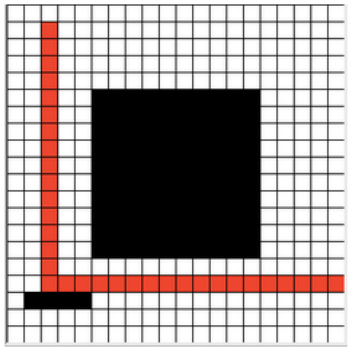
State class

The State abstract class contains all attributes and methods of a general search state in Astar. The attributes are *f, g, h, parent* and *children*. The methods are *reparent, update_f* and the built-in comparison methods *__eq__, __lt__* and *__gt__*. The problem-specific StateMod1 class contains all problem-specific attributes and methods. The attributes are *board*, *dimensions* of the board, and *x* and *y* coordinates of state. The methods to override from State are; *__repr__* which returns a string representation of the *x* and *y* coordinates, *calculate_heuristic* which returns the manhattan distance to the goal (which is why we need the *board* attribute), *calculate_neighbors* which returns a new StateMod1 instance for the directions north, west, south and east. This method also uses the helper functions *is_wall* and *node_out_of_bounds* to check if the proposed values are valid and not a wall. The last function to override is the *__hash__* function which has been mentioned earlier.

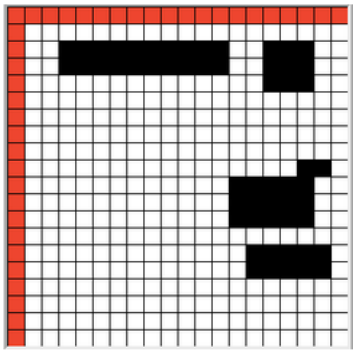
Shortest paths found by A* search
Format: (shortest path cost, nodes created)



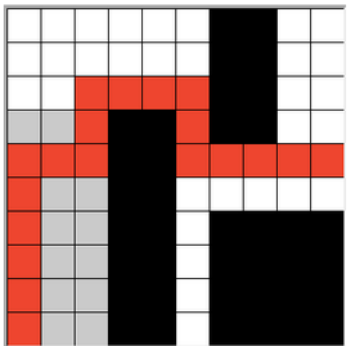
Board 1: (18, 51)



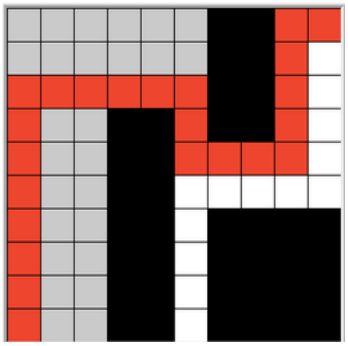
Board 2: (32, 124)



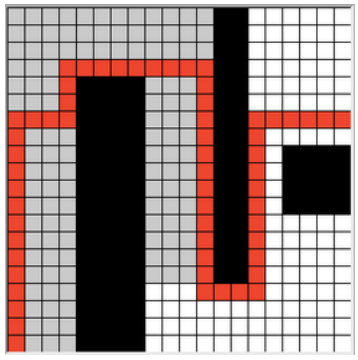
Board 3: (38, 112)



Board 4: (18, 95)



Board 5: (22, 151)



Board 6: (58, 616)