IT3105 Module 1

Iver Jordal

Aspects of the A* program

Generality

The general A* algorithm has its own core class (AStar) which can be used for multiple problems. It has a constructor, where settings are set, and a run function which accepts the start node. The start node must be a subclass of BaseNode. BaseNode is a class which sets up common, general code for a node. It has methods such as calculate_f, which is common for most node implementations. There are also methods such as generate_children and calculate_h, which are problem-specific. These methods must be implemented by subclasses of BaseNode. The navigation problem-specific classes are Main (parse input, start running the algorithm etc), NavNode (subclass of BaseNode), Board (keeps info about the grid and its walls), Gfx (visually display a state), Point (has a position and some math for calculating distances and such), Rect (used for representing the walls and the outer bounds of the grid).

The agenda loop

The A* algorithm implementation is based on pseudo code which I found in slides (page 11-12) here:

http://www.idi.ntnu.no/emner/it3105/lectures/astar-search.pdf

I might have implemented path propagation slightly differently, though. In my implementation, each node has a field for the parent node. If a better path to a node has been found, then that node's reference to the parent is simply updated as a result of running attach_and_eval again on that node.

Data structures

open_list has a special data structure that is a dictionary combined with a heap. I created a class (NodePrioritySet) for handling this in a convenient way. The heap acts as a priority queue where the first element always is the element with the smallest f value. This makes inserting and popping nodes efficient: O(log n). However, a heap is not efficient for checking whether or not an element exists. Therefore the collection is also stored in a dictionary. This allows quick lookup and retrieval, which the A* implementation does. Specifically, that happens in the part where the algorithm checks the current node has been previously generated, and if that is the case, then that node should be re-used.
closed_list does not need to be sorted, but we need to quickly insert, look up and retrieve nodes. Therefore I chose Python's dictionary data structure (it's actually a kind of hashmap), which does all these operations in O(1).

BFS and DFS

In order to make the algorithm run as if it was a breadth first search, I set H_MULTIPLIER to 0 while leaving ARC_COST_MULTIPLIER = 1, which basically means that only G, the "length of the path so far", will be considered when sorting nodes.

For DFS-like search, I set H_MULTIPLIER and ARC_COST_MULTIPLIER to 0. Also, I had to modify my heap insertion function so that it acts as a LIFO for nodes that have same priority. In other words, if two nodes with the same priority are pushed, then when I pop a value, it's going to return the "newest" one of the two nodes.

The program can be run in BFS or DFS mode by simply passing arguments --mode=bfs or --mode=dfs respectively.

Limitation

The A* algorithm will halt if it hits the upper limit (50000000) for generated nodes, in order to avoid very long runs.

GUI

The GUI is created with the help of pygame. The frame rate is limited to 30 by default, but this can be increased by pressing the up arrow key and decreased by pressing the down arrow key. Also, there's a command line argument --fps for setting the default.

Heuristic function

I ended up choosing euclidean distance. I could've chosen manhattan distance because it generally outperforms euclidean distance, but I did not. The reason I didn't choose manhattan distance was that it erroneously produced a slightly longer than optimal path in a huge problem (1600x1200 grid, see test_huge.txt) that I constructed to test the efficiency of the algorithm. Anyway, euclidean distance works well because it is admissible, so the obtained path is guaranteed to be the shortest path.

Generating successor states

As I said earlier, the core code for a node does not have an implementation of generate_children. This must be implemented for each problem, i.e. for each subclass of BaseNode. In NavNode, which is the navigation problem-specific subclass of BaseNode, I implemented generate children like this:

```
candidate_positions = {
    Point(x=self.position.x, y=self.position.y + 1),
    Point(x=self.position.x, y=self.position.y - 1),
    Point(x=self.position.x + 1, y=self.position.y),
    Point(x=self.position.x - 1, y=self.position.y)
}
```

…and then for each candidate position, the program checks if that position is accessible (i.e. is not a wall and is not outside the bounds of the grid). These checks should be quick, since inaccessible_tiles is a set with O(1) lookup time. By the way, inaccessible_tiles only containts tiles that are the outer bound of a wall, in order to be quicker and less memory-intensive, while still behaving correctly.

When the accessible positions are determined, a collection of these child nodes is returned. In the cases where this method generates nodes that have been previously generated, the general A* algorithm will graciously handle this. This is made possible by implementing __eq__ and __hash__ in the node class.