project 1

Running the code

```
python search.py <filename.json>
```

---

1. How did you implement the A* search? Discuss implementation details, including choice of relevant data structures, and analyse the time/space complexity of your solution.

There are a number of classes that have been written to abstract some of the board pathfinding algorithm, these are:

- A Input class to parse the raw json (not strictly needed but gives type hints and makes development easier)
- A Hexagon class that contains coordinates, color, the total cost to get to this node (initially set to infinity) and the incremental path cost to go onto this node (initially set to 1), and the previous node (set to None)
- A Board class that contains a 2d list of Hexagons, and the start and end hexagon and the dimensions of the board.

The Hexagon class is responsible for things like returning the incremental path cost based on the color of the piece, and finding the distance between two different Hexagons using the formula set out in https://www.redblobgames.com/grids/hexagons/#distances

The Board class is the coordinator of the algorithm, once initialised, pieces can be colored in and their incremental path cost will be set to infinity (inf). Once this process is completed the board can run the a star algorithm with the `a_star` method. This function does the following:

- Sets the start node in an `Open` list, and sets `Current` to the first node
- While the current node is not equal to the end node
- sort the list with the calculated expected path cost to the goal node and get the node that is the smallest, set current to this node
- explore all of the neighbours of the current node
    - calculate the cost to get to this neighbour node through the current node,
        - if this is larger than what the neighbour already has then the path will go through the neighbour to the current node, so set the currents previous node to the neighbour.
        - if the cost to get to this neighbour through the current node is less then the neighbours previous node will be set to the current node
        - if the node is not in closed or open add it to the open list
- Once we exhaust the open list we won't have any more nodes that could have less path cost, so return this.
- If we are not yet at the goal node and there aren't any more nodes in the open list then there is not any path from start to finish, so we should return an empty list indicating that there is no valid path.

The maximum branching factor in this network is 5, which is 6 neighbours minus the current node, which means that the worst case time complexity of this algorithm is going to be O(b^d) where be would be the branching factor (5) and d would be the depth, which would be on the order of n, the dimensions of the graph. This means that the overall a star time complexity would be O(b^n).

Whilst we do sort our open list on every iteration of the while loop, and sorting algorithms have a O(nlogn) the maximum amount of nodes that will be appended in one iteration if 5 (6 neighbours - current node) and so the sorting will be a constant operation per iteration. An optimisation of this would be to just use a weighted queue, and this way the queue will be sorted when the items are added to the queue. Also in the sorting, the `distance` function is called, but because this function calculates the distance between two nodes in one operation, the time complexity of calling this function is always O(1).

The space complexity of our algorithm is on the same order of O(b^n), and our queue implementation does at most 5 shuffles per cycle, so the space complexity of this implementation would be on the same order.

---

2. What heuristic did you use and why? Show that it is admissible, and discuss the cost of computing the heuristic function, particularly in relation to the overall cost of the search.

The heuristic we used was the shortest path from the current node to the end node. The way we calculated that was with the adaptation of the algorithm from https://www.redblobgames.com/grids/hexagons/#distances. The heuristic is admissible as it assumes positive weights for paths and the heuristic is weighted to move towards the goal. Because this distance between the current node and the end node is **always** the shortest path, when we sort our queue we are guaranteed to start exploring the nodes that have the shortest paths. If the expansion cannot be explored to the end node (for example there are pieces blocking the way) then the updated path cost will mean that we explore other shorter estimated nodes. When we get to the finish node it is guaranteed to be the shortest path because there are no other possible paths that are less than the one we just took. This heuristic will need to be modified to be permissible in the case of already colored pieces, but this it not to be answered here.

The formula for the heuristic is h(s) <= cost(s,a) + futurecost(s`) where h is the heuristic, s is the current state, a is the action and s` is the next state. The heuristic function that was used is also consistent as the estimate distance is always <= to the actual path used. Hence the heuristic that was used can be said to be admissible.

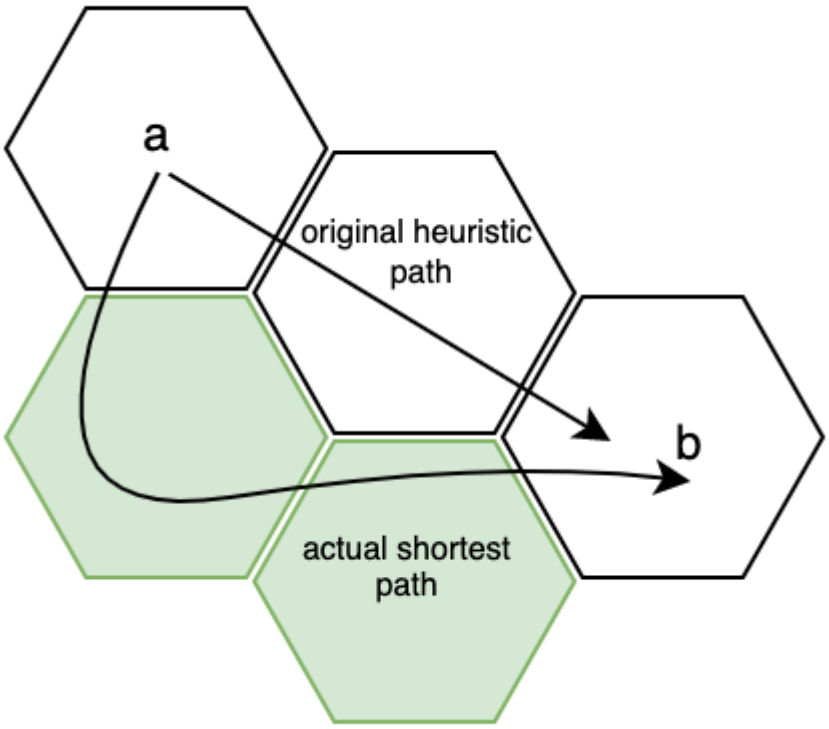
The cost of computing the heuristic function is O(nlogn) as we sort the list every time we go through the while loop, although because we are appending 6 neighbours per loop there are only 6 elements that would need to be sorted on each iteration. This means that the actual time complexity would be at most 6 * 1 as calculating distance between two hexagons is a constant operation.

---

3. (Challenge.) Suppose the search problem is extended such that you are allowed to use existing board pieces of a specific colour as part of your solution path at no cost. An optimal solution is now defined as a minimal subset of unoccupied cells that need to be 'captured' by this colour in order to form a continuous path from the start coordinate to the goal coordinate. How would you extend your current solution to handle this? Discuss whether the heuristic you used originally would still be admissible

Currently in our code each piece holds its path cost in the field `incr_cost`. When the board is being preprocessed and a piece is coloured in this cost is set to `inf`. To implement the described feature it would be simple enough to set this `incr_cost` to 0 instead of setting it to `inf`.

After this the heuristic function would need to be changed to take into account that some pieces on the board have zero cost. Currently this heuristic would fail as it would explore nodes that are the shortest distance, and paths that are out of the way to start with would be unexplored by the time the solution is returned.


a
original heuristic
path
b
actual shortest
path