Alex Nguyen

Project 1: A* On Terrain Maps

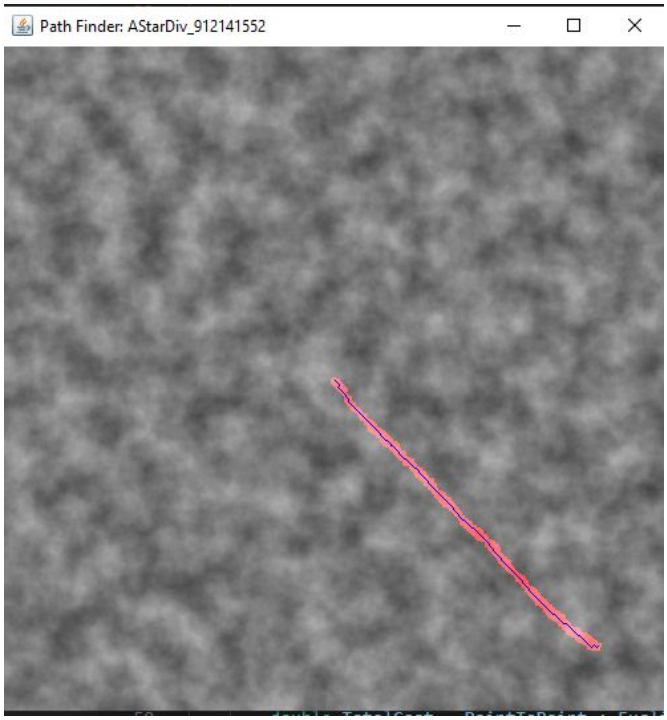
Part 1: Creating Heuristics

1) Cost Function: New height divide by old height

My admissible heuristic I implemented for the new height divided by old height is created by using the cost function + euclidean distance. This is an admissible heuristic because using the euclidean distance the diagonal path cost is less than the cost from traveling along the x-axis + the cost from traveling along the y-axis. We know our start point is higher up in the grid than our endpoint. This means that we traverse our path down the grid. This means that p2 will always be smaller than p1. When we calculate p2/(p1+1) This will always give us a cost of less than 1, which is an underestimate of the cost each time we visit a new point. Since both the cost function and the euclidean distance are both underestimated, I can call this an admissible heuristic.

```
private double getHeuristic (final TerrainMap map, final Point pt1, final Point pt2)
{
    int xDiff = pt1.x - pt2.x;
    int yDiff = pt1.y - pt2.y;
    double PointToPoint = map.getCost(pt1,pt2);
    double MultAdd = xDiff*xDiff + yDiff*yDiff;
    double Euclidean = Math.sqrt(MultAdd);
    double TotalCost = PointToPoint + Euclidean;
    return TotalCost;
}
```

Output:



2) Cost Function: Exponential of height difference

My admissible heuristic that I implemented for the exponential of height difference is created by using the cost + euclidean distance. I proved the euclidean distance is admissible in the section above. We know that p2 will always be smaller than p1 because the starting point is bigger than the end point, therefore the path along the way will always have points smaller than p1. When we take p2 - p1 for each step, we will either get a negative value or a 0. The exponential function takes e^(p2-p1), which will always give us a value less than 1. The bigger the difference is between p2 and p1 the closer the cost gets to 0, which is an underestimate of the cost. Since both the cost function and the euclidean distance are both underestimated, I can call this an admissible heuristic. (I was not able to implement this so my AStarExp is still the same as AStarDiv).

Part 2: A* Algorithm

In my A* algorithm code, I first initialize the class called Index, which will save a path array that will update the longest path, the current point I am at, the g(n) stored in the Cost value, and the f(n) stored in the FullCost value as seen below:

```
class Index {
    private ArrayList<Point> PathArray;
    private Point Point;
    private double Cost;
    private double FullCost;
```

After all the initialization is done, we get into the createPath function, which will use the getHeuristic function to create the path. In this function I initialize the CurrentPoint as the start point, EndPoint, as well as calling the getHeuristic function. My code utilizes a priority queue to traverse the path using both an open list and closed list to manage the points I have visited and the new points. I also initialize an array list called PathArray, which will store the new path for each time I visit a new node, which I can always refer back to using the same index. After that my code will initialize the first point, its cost, and its path to my OpenList. It will also create an empty ClosedList that will later on be used to keep track of the points that were visited. As we go down the while loop it will now take that path, point and cost out of the open list and into the closed list. It then sets the first point as the NewCurrentPoint and uses the getNeighbors function to find all the neighboring points from this current point. After this we will compare all the new neighbors we found to the closed list and if the neighbor is the same it will break out of the code. If they aren't the same then we can continue to create the path using the new neighbor we found as well as calculating the cost and add this to the OpenList. We then go to a new NewCurrentPoint, which is the shortest path so far and continue the while loop process. This loop continues until the NewCurrentPoint is the same as the EndPoint, which the code then returns the path at this index to the main function.

Alex Nguyen

Output for MtStHelens: