Summer Orienteering

Maksim Pikovskiy

CSCI-331: Introduction to Artificial Intelligence

2/6/2022

Table of Contents

Summer Orienteering is a program which finds an optimal path from one point to another using A* search algorithm. Using the provided data files (terrain map, elevation file, and points to visit file), A* search algorithm uses cost and heuristic functions to find the most optimal point to explore next. It continues the process until it encounters the goal point and returns the optimal path it found.

State space:

- Dynamic - the pixel location on the map.
- Static - the map data, such as terrain (and its colors) an elevation.
  the size of the pixels (Longitude of 10.29 meters and Latitude of 7.55 meters).

Initial State:

- Starting location of the search, represented in x and y coordinated (corresponds to pixel on the map).

Goal State:

- Last location after all other locations were visited (in the provided path file).

Actions:

- Move to the neighboring location of the explored location (4 directions: North, South, East, West).

Glossary

Term	Definition
fScore	the total estimated cost to get to goal point
gScore	the total cost of the current path
hScore	the cost of the path from selected point to goal point

lab1 Class

`lab1` Class is the core of this program. It requires four arguments, which are terrain image, elevation file, path file, and output image file path.

Firstly, `lab1` initiates the terrains with their respective names, colors, and modifiers and checks for valid number of arguments. It stops the execution of the program if incorrect number of arguments is received.

After getting the four required arguments, it does the following:

1. Reads from the provided terrain image (for `aStarSearch` to check for colors),
2. Converts the elevations file into 2D array,
3. Reads from the provided path file,
4. Calls on `aStarSearch` to find a path from one point to another.

After `aStarSearch` has finished finding paths between all provided points, `lab1` does the following:

1. Calculates and prints the total distance of the path,
2. Writes the total distance into a `totalDistance.txt` in `output` folder in the directory,
3. Draws a path on the provided terrain map, saving it as separate image with the path specified as last argument (still saves it in `output` folder),
4. Calculates and prints the total amount of time it has taken to calculate the optimal paths between the provided points.

aStarSearch Class

`aStarSearch` Class is another core of this program. It finds the most optimal path between two points using the data that `lab1` has got. It uses a formula f(n) = g(n) + h(n), where g(n) is a cost of a path from one point to another and h(n) is a cost of a path from next point to goal point.

findPath()

`findPath()` implements the core of the `aStarSearch` algorithm and returns the best path to the provided goal point from the given starting point. It calculates the direct neighbors of the point it's currently exploring. That is, it gets the North, South, East, and West neighbors of the currently explored point, but not North-East, North-West, South-East, and South-West neighbors. It puts the neighbors into a `PriorityQueue`, which sorted the points to explore by their fScores produced by the `costFunction()` and `heuristicFunction()` (See their respective sections below). After finding the goal point, the `findPath()` builds an `ArrayList` by getting parents of the point (represented in the `Node` data structure, described in "Node Class" Section) and returns that `ArrayList` back to `lab1` for further processing.

calculateDistance()

`calculateDistance()` is a helper function, which calculates the 3D Euclidean distance between two points.

- The difference of x coordinates is multiplied with Longitude value of 10.29 meters to illustrate actual distance.
- The difference of y coordinates is multiplied with Latitude value of 7.55 meters to illustrate actual distance.
- The difference of x coordinates is the difference between elevation values of two locations, provided by the elevation file.

costFunction()

`costFunction()` is a function that calculates the gScore of the `Node`. That is, the gScore of the `Node` is the cost of the path from one `Node` to its neighbor. It uses `calculateDistance()` to find the distance between two given nodes. It then finds the terrain modifier for both nodes by finding the color of the node and iterating over the list to match the colors and retrieve the terrain modifier (this is done by the `getTerrainModifier()` helper function). After getting both, the distance and modifiers for both nodes, it uses the formula "*(Distance / 2) / TerrainMod1 + (Distance / 2) / TerrainMod2*", to get the cost. That cost is later added to the gScore of the parent `Node`, which is then attributed to the neighbor `Node` for the total path cost, went through thus far.

The terrain modifiers were chosen based on the pictures provided in the Assignment Writeup. The chosen modifiers are shown in the table below.
The modifiers are represented as follows:

- Lower value of modifier means the terrain is worse (slower speed).
- Higher value of modifier means the terrain is better (faster speed).

Thus, the distance in above formula is divided by the terrain modifier. When the distance is divided by a higher value, the cost will be less, and when the distance is divided by a lower value, the cost will be more.

The `costFunction()` is thus admissible, as it calculates the cost of path between neighboring nodes and then the other part of `aStarSearch` algorithm adds that cost to the total cost of the path.

Terrain Name	Terrain Modifier
Open land	1
Rough meadow	0.45
Easy movement forest	0.8
Slow run forest	0.7
Walk forest	0.6
Impassible vegetation	0.001
Lake/Swamp/Marsh	0.001
Paved road	1
Footpath	0.9
Out of bounds	0.00001

heuristicFunction()

`heuristicFunction()` finds the cost of the path from one `Node` to the goal `Node`. It used to calculate which node is best to go on for the most optimal path.

`heuristicFunction()` behaves similarly to the `costFunction()`. The most notable difference is that it divides the distance by the best terrain modifier present in the terrain `ArrayList` to get the cost of the path. It retrieves the distance between the neighbor `Node` and the goal `Node` from `calculateDistance()`.

The `heuristicFunction()` is thus admissible, as it calculates the cost of the path from the one `Node` to the goal `Node`. As `PriorityQueue` prioritizes lower fScore (which is calculated by adding gScore and hScore together), it will place the `Node` with lower hScore first compared to the one that has a higher hScore (as long as their gScores are similar). The `heuristicFunction()` ignores the worst terrains and obstacles to get the direct path to the goal `Node`.

Terrain Class

`Terrain` Class is a data structure representation for the terrains. It includes the name of the terrain (for code readability purposes), a color (which represents the color of the terrain on the terrain map), and modifier (which is the speed of the person on that terrain).

This `Terrain` Class is used to store all terrains that are present on the map, notably in `lab1` Class where terrains are stored in `ArrayList`. `aStarSearch` then utilizes it in its `getTerrainModifier()` method to match the colors and retrieve the terrain modifier that the `Node` will need to estimate its cost.

Node Class

`Node` Class is data structure representation for individual pixels, with their respective fScores, gScores, and hScores. This class is used in `aStarSearch` to compare them and get the optimal path.

Each individual `Node` has a location (represented in `Point` Class), fScore (which is gScore + hScore), gScore (which is the cost of path between two nodes), and hScore (which is the cost of path of this `Node` to goal `Node`), and parentNode (which is the `Node` that preceded this `Node`; used for path `ArrayList` building).

`Node` has functions `compareTo` for use in `PriorityQueue` (compares fScores to place the best one in front of queue), `equals` for use in `[Queue/List/Set].contains(...)`, and `toString` for testing purposes.

Run The Program

To run the program:

1. Place the required files somewhere in the directory `summer-orienteering`.
2. Run the command `java lab1.java [path-to-file]/terrain.png [path-to-file]/mpp.txt [path-to-file]/red.txt [path-to-file]/redOut.png`
3. See the results in `output` directory, with `totalDistance.txt` showing the total distance of the path and `redOut.png` to see the path.

For example: To run the "normal" testcase with brown path selected, this is the command:

> java lab1.java testcases/normal/terrain.png testcases/normal/mpp.txt testcases/normal/brown.txt output/normal/redOut.png

Other

Bugs

One of the current bugs occurs when all terrain modifiers are the same (for example, all of them are 1). It seems that the fScores become very similar the longer the `aStarSearch` algorithm runs and as such, it has to search many more nodes. This bug occurs when the starting point and goal point are on diagonal of each other. To temporarily fix this issue, it is possible to multiply the heuristic cost by 1.5 in `heuristicFunction()`. Multiplying the heuristic cost by 1.5 makes the A* search algorithm more greedy, and as such, allows it to efficiently select the path (it might or might not be the optimal path).

By reading up on A* search algorithm, I can point the issue to the same paths with the same length. As such, there are many tiebreakers and A* search has to search all paths, making it more inefficient. This issue is described in Standford University website called Heuristics,

http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#breaking-ties. They state that to fix such tiebreakers, we can nudge the hScore slightly, which is why I decided to multiply 1.5 whenever all terrain modifiers are exactly the same. It is the lowest value, where A* search suffers the least from inefficiency problem.

While it slightly breaks admissibility of the heuristic, it allows A* search algorithm to explore far less of the map compared when such change is not implemented.

With this bug, the "elevation" testcase is unfortunately unsolvable in a short amount of time, as the algorithm will take a lot of time to solve it by exploring every single `Node`. If heuristic cost is multiplied by 1.5, the algorithm will directly path towards the goal, ignoring the terrain differences (e.g. Mountains with black color). Unfortunately, I have yet to solve this issue.

Output

`output` directory contains all the outputs that were processed after running this program. Each directory in `output` corresponds to the testcases in `testcases` directory.

`totalDistance.txt` is the only one that gets overwritten when a new search is run.