Programming Assignment 2

Department of Computer Science, University of Wisconsin – Whitewater
Algorithms in the Real World (CS 738)

Instructions For Submissions

- **This assignment is to be complete individually.**
- Submission is via Canvas as a single zip file. Submit code and a brief report. No need to include the algorithm description in the report.
- **Your choice of programming languages are limited to C++ or Java.** Make sure you cite everything. If you use online code resources, please make sure that it is not a verbatim copy; you really should avoid this in general.

---

1 Introduction

In this assignment, we will carry out a comparative analysis of P2P shortest path algorithms:

- Dijkstra's algorithm
- Bidirectional algorithm
- A* using 2 heuristics: Euclidean distance, and Landmarks

Data Description. The graph data can be found in the assignment folder. All graphs are undirected. This is a modified version of the data from DIMACS. Each dataset comprises of 2 files:

- `.len files`: First line is the number of vertices and edges respectively for this region. Second line onwards are the edges in the graph; in particular, each line contains three entries: the source vertex, the target vertex, and the length of the edge.

  The graph class contains the graph as an adjacency list. Also, provided is the reversed graph as a reverse adjacency list, which will be the same as the adjacency list when the graph is undirected. Since graphs are undirected for this project, the two adjacency lists are identical.

- `.co files`: You will use this for the A^* euclidean heuristics and computing the grid landmarks. First line is the number of vertices for this region. Second line onwards, each line contains three entries: a vertex, its x co-ordinate, and its y co-ordinate. The distance between two points (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

  The graph class stores these co-ordinates using a class `Vertex`, which has the following fields: `int` id, `double` x, and `double` y. The `Graph` class uses an array/vector `coords` of type `Vertex`, having length the same as the number of vertices.

2 Part 1: Bi-directional and A* implementations

Code for P2P Dijkstra's algorithm has been provided. Read the code to understand the structure of the project as most of the algorithms can be implemented by making light to moderate changes to this code. *Notice the field* `numEdgesRelaxed`. This will keep track of the number of edges that are being relaxed by the algorithm. This is incremented by the number of outgoing edges of a vertex which is being relaxed.

2.1 Task 1: Bidirectional Algorithm

Pseudo-code for this part has been provided in the notes. Use it to implement the `execute` method in the `Dijkstra_Bidirectional` class. Update the field `numEdgesRelaxed` as described in P2P Dijkstra. Notice that this variable must be updated both in the forward and reverse directions.

2.2 Task 2: A*

In the `AStar` class, implement the A* algorithm with the abstract/virtual heuristic function. To be honest, the code will be pretty much identical to P2P Dijkstra, except with one change – when adding a vertex v to the priority queue *open*, use $dist + heuristic(adjVertex, target)$ as the priority, where heuristic is a function within the file to be described subsequently. (When you are initially adding the source to *open*, use $heuristic(source, target)$ instead of 0.)

2.2.1 Task 2.1: A* with Euclidean Distance

In the `AStar_Euclidean` class, $heuristic(v, target)$ provides the Euclidean distance between v and $target$. Use the coordinates from the Graph super class to find the Euclidean distance between v and $target$ and return it.

2.2.2 Task 2.2: A* with Landmarks

The `AStar_Landmark` class constructor loads the pre-computed distances from each landmark to every vertex and from every vertex to each landmark. These are stored in two 2-dimensional arrays/vectors (at the class level), which are named as `fromLandmarks` and `toLandmarks`. Each array has as many rows as the number of landmarks and as many columns as the number of vertices. The `fromLandmarks` array contains the distances from each landmark to every vertex. The `toLandmarks` array contains the distances to each landmark from every vertex in case of a directed graph, else it is null (which is the case for this project).

Once all the landmarks have been loaded, we compute $heuristic(v, target)$ as follows:

- Set $H = 0$. Here, H is the heuristic score.
- Run a loop from $\ell = 0$ to $\ell <$ number of landmarks. Within this loop,
  - compute $h = \left|\texttt{from}[\ell][target] - \texttt{from}[\ell][v]\right|$
  - if $(h > H)$ set $H = h$
- return H

2.3 Task 3: Test your implementation

I have provided the landmark files for NewYork ($regionIndex = 0$). At this point, you should be able to test the code by running the main function in the `Test` file. Depending upon how you have stored the data, you may need to modify the `DATA_FOLDER`, `GRID_LANDMARK_FOLDER`, and `RANDOM_LANDMARK_FOLDER` paths at the top of the class. Try with different `NUM_PAIRS` to ensure that the code is correctly implemented. Here's a sample output.

Java

```
Loading NewYork...Done!

Testing with 25 source-target pairs for NewYork

P2P: Avg. Time = 37.8, Avg. no. of edges relaxed = 409919.2
Bi-directional: Avg. Time = 24.44, Avg. no. of edges relaxed = 255724.44
Euclidean: Avg. Time = 37.92, Avg. no. of edges relaxed = 384328.08

Grid Landmark(4): Avg. Time = 5.6, Avg. no. of edges relaxed = 58826.72
Random Landmark(4): Avg. Time = 7.88, Avg. no. of edges relaxed = 84450.4

Grid Landmark(9): Avg. Time = 3.48, Avg. no. of edges relaxed = 35376.24
Random Landmark(9): Avg. Time = 4.12, Avg. no. of edges relaxed = 43487.44

Grid Landmark(16): Avg. Time = 2.6, Avg. no. of edges relaxed = 24158.0
Random Landmark(16): Avg. Time = 3.16, Avg. no. of edges relaxed = 30068.72

Grid Landmark(25): Avg. Time = 2.36, Avg. no. of edges relaxed = 18754.68
Random Landmark(25): Avg. Time = 3.48, Avg. no. of edges relaxed = 24605.44

Grid Landmark(36): Avg. Time = 2.44, Avg. no. of edges relaxed = 15935.72
Random Landmark(36): Avg. Time = 3.4, Avg. no. of edges relaxed = 20682.4
```

C++

```
Loading NewYork...Done!

Testing with 25 source-target pairs for NewYork

P2P: Avg. Time = 104.222, Avg. no. of edges relaxed = 355532
Bi-directional: Avg. Time = 73.4509, Avg. no. of edges relaxed = 212592
Euclidean: Avg. Time = 102.331, Avg. no. of edges relaxed = 329887

Grid Landmark(4): Avg. Time = 23.3057, Avg. no. of edges relaxed = 47324.4
Random Landmark(4): Avg. Time = 29.8893, Avg. no. of edges relaxed = 69199.2

Grid Landmark(9): Avg. Time = 20.1087, Avg. no. of edges relaxed = 34486.6
Random Landmark(9): Avg. Time = 24.906, Avg. no. of edges relaxed = 48960.6
```

```
Grid Landmark(16): Avg. Time = 17.8145, Avg. no. of edges relaxed = 25986.1
Random Landmark(16): Avg. Time = 18.9017, Avg. no. of edges relaxed = 28606.8

Grid Landmark(25): Avg. Time = 16.0929, Avg. no. of edges relaxed = 19047.5
Random Landmark(25): Avg. Time = 15.8818, Avg. no. of edges relaxed = 18507.2

Grid Landmark(36): Avg. Time = 16.223, Avg. no. of edges relaxed = 17538.6
Random Landmark(36): Avg. Time = 17.9212, Avg. no. of edges relaxed = 21087.5
```

If you notice that your code breaks or the output is substantially different, then you should revisit the code that you have written before proceeding to the next part.

3 Part 2: Landmark Creation

Now, we focus on landmark creation. The code for creating random landmarks has been provided; scan it briefly as it may make the following discussion easier to follow.

3.1 Task 1: Pre-compute Distances

In the `LandmarkGenerator` class, complete the `precomputeLandmarkDistances` function as follows:

- First create a `Dijkstra_SSSP` object on the graph.
- Now, create a two-dimensional array/vector (in Java/C++). This will have as many rows as the number of landmarks.
- Now, for each landmark in the hashset:
  - Run Dijkstra's algorithm from the landmark.
  - Store the distance array returned in the previous step in the 2d array/vector.
- Call the `writeDistances` function of the `LandmarkReaderWriter` class with the arguments – the two dimensional array/vector, the number of landmarks, the number of vertices in the graph, and `fromLandmarkPath`.

P.S. Since the graph is undirected, we do not pre-compute from vertices to landmarks; hence, `toLandmarkPath` *is not needed.*

At this point, we can create random landmarks. Once you uncomment the function call `generateRandomLandmarks(regionIndex, g)` in main, you should be getting the following output.

```
Loading NewYork...Done!

Creating 4 random landmarks for NewYork...Done!
Creating 9 random landmarks for NewYork...Done!
Creating 16 random landmarks for NewYork...Done!
Creating 25 random landmarks for NewYork...Done!
Creating 36 random landmarks for NewYork...Done!
```

3.2 Task 2: Grid Landmark Creation

In the `LandmarkGenerator` class, complete the `makeGridLandmarks` function to create $gridDim^2$ many grid landmarks as follows:

- Obtain the co-ordinates from the graph.
- Find the following:
  - maximum x value: $maxX$ and minimum x value: $minX$
  - maximum y value: $maxY$ and minimum y value: $minY$
- Then compute

$$gridXSize = (maxX - minX)/gridDim$$
$$gridYSize = (maxY - minY)/gridDim$$

- Compute the center point of each grid using the above information. Thus, you should be computing $gridDim^2$ many center points.
- Create a hashset to store the landmarks.
- For each center point,
  - find the vertex in the graph that is closest to it based on Euclidean distance.
  - add this vertex to the hashset as the grid landmark for that center point.
- Finally, call the `precomputeLandmarkDistances` function as described at the end of the `makeRandomLandmarks` function.

At this point, we can create random landmarks. Once you uncomment the function call `generateGridLandmarks(regionIndex, g)` in main, you should be getting the following output.

```
Loading NewYork...Done!

Creating 4 grid landmarks for NewYork...Done!
Creating 9 grid landmarks for NewYork...Done!
Creating 16 grid landmarks for NewYork...Done!
Creating 25 grid landmarks for NewYork...Done!
Creating 36 grid landmarks for NewYork...Done!
```

4 Part 3: Comparison

Run the code for the other regions; remember to generate the landmarks before comparative analysis. If it is too slow, then revisit the algorithms and/or try with a fewer number of source-target pairs (no less than 10 pairs to ensure an unbiased average).

Submit the output for each region along with your code.

5 Experimental Setup

You don't have to do anything for this part, but for future assignments, especially the project, here are a few things to keep in mind.

- **Make sure that the algorithms are correct.** The correctness phase as written in the `Test` file should take care of this, but we should still check our code carefully.
- **Whenever we are time-testing an algorithm, we must ensure that we are only running the algorithm between the start and end time of the experiment.** For example, we do not include graph loading time and graph reversing (for the bi-directional algorithm), or landmark generation time/loading time of pre-computed landmark distances. Obviously, we must include the heuristic computation time in case of the A^* algorithms.
- **Don't do something within the algorithms which they are not supposed to do.** For example, you MUST use a priority queue for finding the vertex with the minimum estimate, and you should NOT scan through the entire distance array. (A priority queue based implementation of Dijkstra has been given to you.) Do not write to a file, or print within the algorithm as they will result in I/Os, which will increase time.
- **Use random generation for the source and target vertices.** Ideally, to avoid biasing the experiment, all source and target vertices should be distinct; however, I have skipped this. However, we must use the same set of source and target vertices in all the algorithms; otherwise, the experiment is incorrect. Remember that when we compare algorithms, the test bed must be identical.

6 References

The following papers form the basis of this assignment (and experiments).

- Computing the Shortest Path: A* Search Meets Graph Theory by Andrew V. Goldberg and Chris Harrelson
- Reach for A*: Efficient Point-to-Point Shortest Path Algorithms by Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck