Map Routing

Problem Definition 2

Goal and Suggestions 2

Project Requirements 3

- Input 3
- Output 3
  - How to calculate execution time? 3
- Test Cases 3

Deliverables 5

- Implementation (60%) 5
- Document (40%) 5

BONUS TASK 5

Problem Definition

The goal is to implement the classic Dijkstra's shortest path algorithm and optimize it for maps. Such algorithms are widely used in geographic information systems (GIS) including MapQuest and GPS-based car navigation systems.

Maps are graphs whose vertices are points in the plane and are connected by edges whose weights are **Euclidean distances**. Think of the vertices as cities and the edges as roads connected to them. To represent a map in a file, we list the number of vertices and edges, then list the vertices (index followed by its x and y coordinates), then list the edges (pairs of vertices). For example, Sample Map represents the map below:

```
6 9
0 1000 2400
1 2800 3000
2 2400 2500
3 4000    0
4 4500 3800
5 6000 1500

0 1
0 3
1 2
1 4
2 4
2 3
2 5
3 5
4 5
```

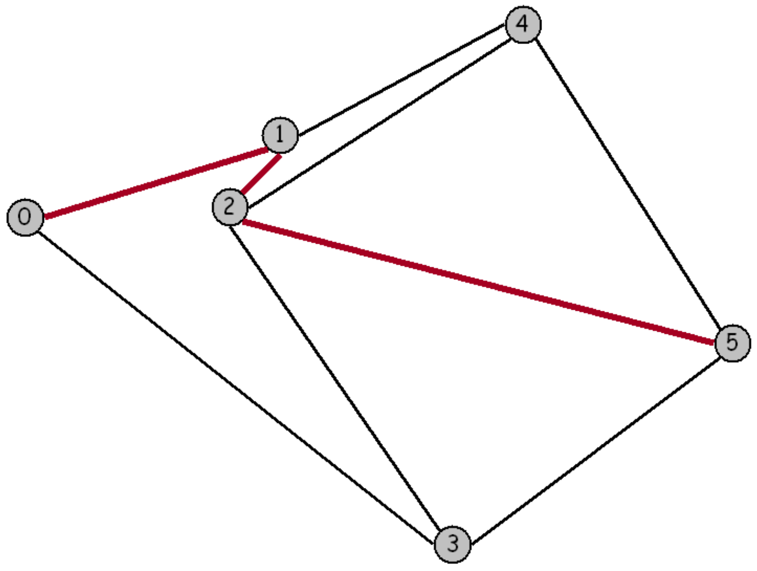

Figure 1: Sample map file

Dijkstra's algorithm is a classic solution to the shortest path problem. It is described in Textbook, Section 24.3. The basic idea is not difficult to understand. It maintains, for every vertex in the graph, the length of the shortest known path from the source to that vertex, and it maintains these lengths in a **priority queue** (described in textbook, Section 6.5). Initially, it put all the vertices on the queue with an artificially high priority and then assigns priority 0 to the source. The algorithm proceeds by taking the lowest-priority vertex off the priority queue. Then checking all the vertices that can be reached from that vertex by one edge to see whether that edge gives a shorter path to the vertex from the source than the shortest previously-known path. If so, it lowers the priority to reflect this new information.

This method computes the length of the shortest path. To keep track of the path, it also maintains for each vertex, its predecessor on the shortest path from the source to that vertex.

Goal and Suggestions

Your goal is to optimize Dijkstra's algorithm so that it can process thousands of shortest path queries for a given map. Once you read in (and optionally preprocess) the map, your program should solve shortest path problems in *sublinear* time. One method would be to precompute the shortest path for all pairs of vertices; however you cannot afford the quadratic space required to store all of this information. Your goal is to reduce the amount of work involved per shortest path computation, without using excessive space. There are three suggestions of potential ideas below which you may choose one to implement. Or you can develop and implement your own ideas.

Idea 1. The naive implementation of Dijkstra's algorithm examines all V vertices in the graph. An obvious strategy to reduce the number of vertices examined is to stop the search as soon as you discover the shortest path to the destination. With this approach, you can make the running time per shortest path query proportional to E' log V' where E' and V' are the number of edges and vertices examined by Dijkstra's algorithm. However, this requires some care because just re-initializing all of the distances to ∞ would take time proportional to V. Since you are doing repeated queries, you can speed things up dramatically by only re-initializing those values that changed in the previous query.

Idea 2. You can cut down on the search time further by exploiting the Euclidean geometry of the problem. For general graphs, Dijkstra's relaxes edge v-w by updating wt[w] to the sum of wt[v] plus the distance from v to w. For maps, we instead update wt[w] to be the sum of wt[v] plus the distance from v to w *plus* the Euclidean distance from w to d *minus* the Euclidean distance from v to d. This is known as the *A* algorithm*. This heuristics affects performance, but not correctness.

Idea 3. Use a faster priority queue. There is some room for optimization in the supplied priority queue. You could also consider using a multiway heap.

Project Requirements

1. Implement the Dijkstra's Single-Source Shortest Path based on Priority Queue
   - Dijkstra's shortest path algorithm **Sec.24.3**
   - Priority queue (heap data structure) **Sec.6.5**
2. Implement one of the optimization ideas for map-routing requests

Input

- **Map file** containing
  1. the number of vertices and edges,
  2. list of vertices (index followed by its x and y coordinates),
  3. list of edges (pairs of vertices).
- **Routes** File contains
  1. # required routes
  2. Source & sink of each route

Output

Your program should print

1. shortest path of each route
2. Total execution time of routing requests

How to calculate execution time?

- To calculate time of certain piece of code:
  1- Get the system time before the code
  2- Get the system time after the code
  3- Subtract both of them to get the time of your code
  To get system time in milliseconds, you can use `System.Environment.TickCount`

Test Cases

- Sample Case:
  1. 6 vertices and 9 edges
  2. One file contains 5 routing requests
- USA Case:
  1. 87,575 vertices and 121,961 edges
  2. Two files with long routes
  3. Three files with short routes

Deliverables

Implementation (60%)

1. Dijkstra's shortest path algorithm **(Sec.24.3)**
2. Priority queue (heap data structure) **(Sec.6.5)**
3. One of the optimization ideas

Document (40%)

1. Source code
2. Analysis of your code
3. Execution time before and after applying the optimization idea

BONUS TASK

- Implement more than one optimization idea and compare their execution times.