CPS 530: Programming Assignment

150 pts

This is NOT teamwork. No submission will be accepted after the deadline

Receive an F for this course if any academic dishonesty occurs

Receive 5 bonus points if submit it without errors at least one day before deadline

1. Purpose

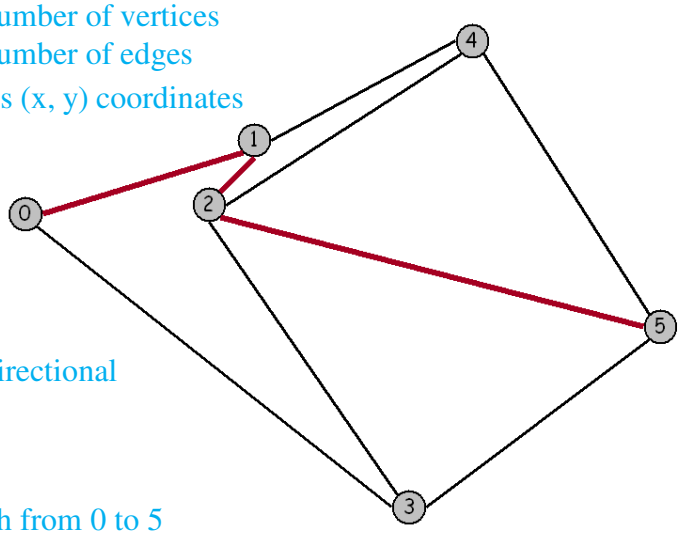
Implement the classic Dijkstra's shortest path algorithm and optimize it for maps. Such algorithms are widely used in geographic information systems (GIS) including MapQuest and GPS-based car navigation systems.

2. Description

Maps. For this assignment we will be working with *maps*, or 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), and finally the source and sink vertices. For example, `input.txt` represents the map below:

```
6 9          ← 6 is the number of vertices
             ← 9 is the number of edges
0 1000 2400
1 2800 3000  ← node 1's (x, y) coordinates
2 2400 2500
3 4000    0
4 4500 3800
5 6000 1500

0 1
0 3
1 2
1 4
2 4          ← each edge is bidirectional
2 3
2 5
3 5
4 5
0 5          ← find shortest path from 0 to 5
```



Dijkstra's algorithm. Dijkstra's algorithm is a classic solution to the shortest path problem. The basic idea is not difficult to understand. We maintain, for every vertex in the graph, the length of the shortest known path from the source to that vertex, and we maintain these lengths in a priority queue. Initially, we put all the vertices on the queue with an artificially high priority and then assign priority 0.0 to the source. The

algorithm proceeds by taking the lowest-priority vertex off the PQ, then checking all the vertices that can be reached from that node by one edge to see whether that edge gives a shorter path to the node from the source than the shortest previously-known path. If so, it lowers the priority to reflect that fact.

Here is a step-by-step description that shows how Dijkstra's algorithm finds the shortest path 0-1-2-5 from 0 to 5 in the example above.

```
process source node 0 (distTo[0]=0.0)
    lower distance to node 3 distTo[3] to 3841.9
    lower distTo[1] to 1897.4
process 1 (1897.4)
    lower distTo[4] to 3776.2
    lower distTo[2] to 2537.7
process 2 (2537.7)
    lower distTo[5] to 6274.0
process 4 (3776.2)
process 3 (3841.9)
process 5 (6274.0)
```

This process gives the length of the shortest path. To keep track of the path, we also maintain for each vertex, its predecessor (i.e., parent) on the shortest path from the source to that vertex.

Your goal. Optimize Dijkstra's algorithm so that it can **process thousands of shortest path queries for a given map (i.e., for each pair (source, destination), your program should print its shortest path)**. 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 the results. So, your goal is to reduce the amount of work involved per shortest path computation. We suggest a number of potential ideas below which you may choose to implement. Or you can develop and implement your own ideas.

Idea 1 (receive 20 points of 70 points. See grading notes). 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.

Idea 2. Change the search to use a symmetric strategy: keep two priority queues, one for the source and one for the destination, and alternate picking the best point off each queue until they "meet in the middle." Determining exactly what to do at this point

(and what is meant by when they meet the middle) to guarantee that you have a shortest path is the most difficult part of this task.

Idea 3. Use a faster priority queue. Consider a multiway heap (each non-leaf node has d children, where $d \geq 2$).

Idea 4. Preprocess the graph by dividing it into a W-by-W grid and precompute the distance between each pair of grid cells i and j and use these distances to guide the search. This idea is probably most useful when the source and destination are far apart.

Idea 5. Many of the vertices have exactly two neighbors. If a vertex `v` has exactly two neighbors `u` and `w`, then you can replace the two edges `u-v` and `v-w` with a single edge `u-w` whose length is the sum of the lengths of the two original edges. This reduces the number of vertices and edges in the graph. The main challenge here is to print out the shortest path in the original graph after you've compute the shortest path in the reduced graph.

Your idea: receive 10 bonus points if you come up with your idea (not same as the ideas listed above)

Testing. The file `usa.txt` contains 87,575 intersections and 121,961 roads in the continental United States. The graph is very sparse - the average degree is 2.8. Your main goal should be to answer shortest path queries quickly for pairs of vertices on this network. Your algorithm will likely perform differently depending on whether the two vertices are nearby or far apart. We provide input files that test both cases. You may assume that all of the x and y coordinates are integers between 0 and 10,000.

Deliverables. Improve the bare bones implementation (i.e., the shortest path algorithm discussed in class) by using some of the ideas described above together with your ingenuity. Zip and submit your entire project (java, C++ or Python is fine) needed to compile your program. Also, submit a **`readme.txt` file**, **where you should explain how to compile/run your program, describe and justify your approach, and explain what difficulties/challenges you meet in this assignment.** Also, compare your approach to the bare bones implementation (i.e., Dijkstra's algorithm. See slides for implementation details).

3. Grading notes

- If your program does not compile, you receive **zero** points for this assignment.
- (15 pts) You should implement the adjacency list data structure for the graph.

- (40 pts) Be sure to test the correctness of your bare bones implementations (i.e., the original Dijkstra's shortest path algorithm).
- **(70 pts)** Be sure to show how you improve the implementation.
- (15 pts) Be sure to complete readme.txt file.
- (10 pts) Your code will be graded based on whether or not it compiles, runs, produces correct output, and your coding style (does the code follow proper indentation/style and comments).