CS3610 Project 5

Dr. Jundong Liu

Shortest-path algorithm and implementations

In order to find a single source shortest path, you are asked to implement Dijkstra's algorithm we went through in class (the PowerPoint Ch12_part2.pptx is attached in this assignment).

Array version

The array version of Dijkstra's algorithm can be found on slides 23 and 24 in the lecture notes. If you recall, this implementation runs in $O(V^2+E)$ time, where V is the number of vertices in the graph and E is the number of edges. The $O(V^2)$ component refers to the number of operations carried out to find every minimum distance vertex extracted from the set of unvisited vertices at the beginning of each iteration of the while loop. The $O(E)$ component results from comparing and possibly updating the distances of all vertices adjacent to the minimum distance vertices. In other words, the for loop within the while loop runs $O(E)$ operations in total.

Heap version: bonus implementation

If your graph is rather sparse (meaning the graph does not contain an overwhelming number of edges), you may want to consider storing the set of unvisited vertices in a min heap using distance to the source vertex as a key. This will help you find all the minimum distance vertices extracted at the beginning of each iteration of the while loop in $O(V \log(V))$ time as opposed to $O(V^2)$ time. Of course, when you now update the distance of a neighboring vertex in the for loop below, you must also update that vertex's position in the min heap. As you already know, bubbling up an element in a min heap of n elements takes just $O(\log(n))$ time, but the initial searching for the element takes $O(n)$ time. In order to avoid the $O(n)$ search, you must implement a lookup table that returns a vertex's index in the min heap in $O(1)$ time. If implemented correctly, the cumulative time complexity of updating the distance values of vertices in the min heap version of Dijkstra's would be $O(E \log(V))$. In other words, using a lookup table and a min heap, the for loop within the while loop runs in $O(E \log(V))$ time as opposed to the $O(E)$ time seen in the version of Dijkstra's algorithm described in the previous paragraph. Thus, the total time complexity of this modified Dijkstra's algorithm is $O(V \log(V) + E \log(V))$. As said early, it is more advantageous to use the min heap version if your graph is sparse.

Lookup table and min-heap: There could be different ways to implement this lookup table. One setup could be an array of pointers, where each element stores the address of the corresponding vertex in the min-heap. In order to communicate back to the lookup table, the elements in the min-heap should be designed as the combinations of (distance, vertex-index) or (distance, vertex-index, predecessor), which is similar to Elon Musk's ($251B, Texas) setup in Project 4. In project 4, I suggested you use STL's

priority_queue to implement your max-heap. In this project, however, you may have to implement a heap class by yourself, as the needed operations are not available in STL *priority_queue* or *heap*.

Assignment and Bonus

In this project, the 5 final grade points will be awarded if you successfully implement the array version of Dijkstra's algorithm, as described in lecture. If you successfully implement the min heap version, you will be awarded additional 4 bonus points. For the students who decide to implement the heap version, I would suggest you start with the array version for a correct baseline. Also, please include a note in your submission to inform the TA that your implementation is the heap version.

Input

Input is read from the keyboard. **The first line of input is the number of test cases** K. Each of the K test cases is written in the following format:

Individual Test Case Format

```
n
city_1
city_2
.
.
.
city_n
d_11 d_12 ...  d_1n
d_21 d_22 ...  d_2n
.
.
.
d_n1 d_n2 ...  d_nn
```

The first line of each test case is the number of cities n in the graph. The next n lines are the names of each city. City names consist only of alphabetic characters. Following the list of n city names is an nxn distance matrix where each distance `d_ij` is an integer value in the range $[0, 10000]$ representing the distance of the road connecting city i to city j. A distance `d_ij` $= 0$ indicates that there does not exist any road connecting city i to city j. For this project, all roads will be undirected, which means `d_ij` $=$ `d_ji` for all cities i and j. As a result, every input distance matrix will be symmetric.

Output

For each test case, output a space delimited list of all the city names in the shortest path connecting `city_1` and `city_n` followed by the integer distance of the path. City

names should be listed in order of when they are to be visited starting from `city_1` and ending with `city_n`. If `city_1` has multiple shortest paths to `city_n`, just output one of them. Also note that in every test case, there will always be a path connecting `city_1` to `city_n`.

Sample Test Cases

Use input redirection to redirect commands written in a file to the standard input, e.g. `$ ./a.out < input1.dat`.

Input 1

```
1
4
Akron
Athens
Columbus
Cleveland
0 1 2 0
1 0 5 6
2 5 0 7
0 6 7 0
```

Output 1

```
Akron Athens Cleveland 7
```

Turn In

Submit your source code under blackboard. If you have multiple files, package them into a zip file.

Grading

Total: 100 pts.

- **10**/100 - Code style, commenting, general readability.
- **10**/100 - Compiles.
- **10**/100 - Follows provided input and output format.
- **70**/100 - Successfully implemented Dijkstra's algorithm.
- **80**/100 - Bonus points for heap implementations.