Spring 2021 - BBM204: Software Practicum Resit Assignment

Hacettepe University Computer Engineering Department

Due Date: June 24, 2021 23:59

Introduction

The shortest path problem is defined as finding the shortest path between a starter vertex (source) and a target vertex (destination) in a digraph data structure, and it plays an important role for many practical applications; for example, in network routing, to send internet packets through the shortest path between routers; in chip design, to produce the most efficient circuit; in shipping, to transmit shipments in the shortest time, etc. There are various shortest path algorithms, such as Breadth-First Search, Dijkstra, and A*, which find the least cost path to reach the target vertex from the starter vertex.

However, in real-life problems, it is possible to encounter different constraints while finding the shortest path. Therefore, these shortest path algorithms must be modified according to the given constraints. Suppose you need to develop a navigation system that offers the shortest or least-cost path between the source and destination locations defined by a user. In this case, we can use standard shortest path algorithms; however, if we want to visit certain locations, we must make some changes to the existing algorithms or develop a new strategy to meet the given constraints.

In this assignment, you are expected to present and develop your own strategy for a shortest path problem with constraints. You can modify any existing algorithm, develop your strategy or even build your own algorithm. All details are given below.

Problem Definition

Given a weighted digraph that represents locations as vertices and roads as edges with related distances and must-visit vertices as constraints, you will develop or look for a solution to find the shortest path between the source and destination vertices, provided that your shortest path **must contain the given must-visit vertices**.

Step 1: Reading the Input File

Your program should accept the name of the input file as the first argument. The input file consists of the necessary information for building the digraph and running your algorithm. The sample input looks like the following figure:

```
S: 1, D: 12
1. 2(2), 5(13), 6(8)
2. 3(20), 6(27)
3. 4(14), 7(14) must-visit
4. 8(15)
5. 6(9), 9(15)
6. 3(12), 7(10), 10(20)
7. 8(25), 10(9), 11(12)
8. 3(11), 12(17)
9. 6(12), 10(18) must-visit
10. 5(11), 11(8)
11. 8(13), 12(11)
12. ()
```

Figure 1: Sample input file

The first line describes the source and destination vertices, while the following lines have the necessary information to build the digraph and to mark the required vertices as must-visit. The input format of a must-visit vertex is given below:

```
3. 4(14), 7(14) must-visit
```

The first number followed by a dot indicates the number of the vertex in the digraph, while the "must-visit" tag indicates the vertex that should be visited before reaching the destination vertex (in this sample, vertex 3 is a "must-visit" vertex). The middle elements separated by a comma indicate the adjacency status with the corresponding weight. The first number indicates the destination vertex of the defined edge, while the number inside the parentheses indicates the weight of the edge.

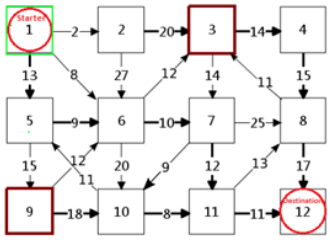

Figure 2: Illustration of the graph regarding the sample input

You should use a *Scanner* instance to perform line-by-line reading of the input file. You should come up with a *regular expression* to parse the first line, and use *groups* to get the number of the start and end vertices.

S: 1 , D: 12

Figure 3: An example of matching text as groups using the first regular expression, match marked in blue, groups in green and orange respectively

After this step, your program should start parsing the digraph construction information stated in the remaining lines. To parse each line and construct your digraph, you should again come up with a *regular expression* to find every instance of the following format:

```
4(14)
```

You should add vertices and edges with respect to the vertex number and adjacency information.

3. 4 (14) , 7 (14) must-visit

Figure 4: An example of matching text as groups using the second regular expression, match marked in blue, groups in green and orange respectively

If you encounter a "must-visit" keyword in your digraph, you should add that to your list of must-visit vertices which will be used later to generate permutations.

For a less painless testing experience, you can use regex101.com for testing your regular expressions.

Step 2: Generating Permutations

To find an optimal solution to this particular problem, you are expected to generate all permutations of your must-visit list. When the vertices 3 and 9 are given as must-visit vertices, vertex 1 given as the source, and vertex 12 given as the destination, the permutations of the must-visit array are as follows:

```
[1, 3, 9, 12]
[1, 9, 3, 12]
```

Figure 5: Permutations of the must-visit list using the sample input

```
[1, 3, 9, 10, 12]
[1, 3, 10, 9, 12]
[1, 9, 3, 10, 12]
[1, 9, 10, 3, 12]
[1, 10, 3, 9, 12]
[1, 10, 9, 3, 12]
```

Figure 6: Permutations of the must-visit list when the vertex 10 is added

Step 3: Finding the Optimal Solution to the Problem

After generating the permutations of the must-visit list, you should try each of them in the following way to get the solution with the minimum total weight, therefore optimal:

1. Generate the concatenated shortest path using the vertices in the permutation (e.g., 1 to 3, 3 to 9, ...).
2. Use the concatenated path as the final path and check if it has the minimum total weight.
3. Repeat the first two steps until you have depleted all of the permutations.
4. The optimal solution is the one with the minimum total weight. Make sure to save this solution for outputting it later.

Step 4: Output

At every permutation of the must-visit list, you should print which permutation is being investigated along with the corresponding concatenated path and its total weight. The output regarding the sample input given in Figure 1 can be seen below:

```
Investigating: [1, 3, 9, 12]
Path found: [1, 6, 3, 7, 10, 5, 9, 10, 11, 12]
With weight: 106.0

Investigating: [1, 9, 3, 12]
Path found: [1, 5, 9, 6, 3, 7, 11, 12]
With weight: 89.0

Optimal solution is: [1, 9, 3, 12]
With optimal path: [1, 5, 9, 6, 3, 7, 11, 12]
With optimal weight: 89.0
```

Figure 7: Sample output

Figure 8 illustrates the optimal solution regarding the sample input given in Figure 1.

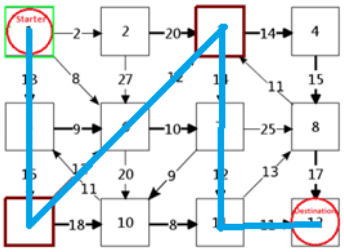

Figure 8: Illustration of the optimal solution regarding the sample input

Notes and Restrictions

- Your work will be graded over a maximum of 100 points according to the grading policy.
- Do not miss the submission deadline.
- Pay attention to the comments and the structure of the coding template.
- Submit your work using codepost.io.
- Save all your work until the assignment is graded.
- Regardless of the length, use **understandable** names for your variables, classes and functions. The names of classes, attributes and methods should obey Java naming convention. This expectation will be graded as **coding standards**.
- Source code readability is of a great importance for us. Thus, write a **readable code** and **comments**. This expectation will be graded as **clean code**.
- All assignments must be original, individual work. Duplicate or very similar assignments are both going to be considered as cheating.

Grading

Task	Points
Submitted	1
Coding standards	5
Clean code	10
Reading the input file correctly	10
Generating correct permutations	20
Calculating each concatenated path correctly	20
Calculating total weight associated with the each concatenated path	20
Finding the optimal solution correctly	14
Total	100