Centre for infrastructure, Sustainable Transportation and Urban Planning

Indian Institute of Science (IISc), Bengaluru
C++ Developer: Round 1

The aim of this exam is to test your problem-solving skills and basic understanding of C++. All questions are NOT compulsory, and partial solutions will also be considered while shortlisting for subsequent rounds. Hence, you are encouraged to submit the best possible answer for each question. Follow the instructions below precisely.

- Plagiarism will result in instant disqualification. You must write your own code.
- The test requires you to fork a GitHub repository and work on it. To submit your solution, follow the steps:
  - Push the updated/new files to your repository and add Prateek1009 (link) and mxthxngx (link) as collaborators. Remember that you must create a private fork of the repository. Public repositories will not be considered for evaluation.
  - Fill the following Google Form: link. You are allowed to make only one submission.
- The test commences on 10th September 2023 (08:00 AM). The last date for submission is 12th September 2023 (08:00 PM). Late submissions will not be accepted.
- If selected, it will be mandatory for you to join in an in-person capacity. Please refrain from attempting the test if you are unable to attend in person.

All the best.
C*i*STUP
Indian Institute of Science, Bengaluru

Question 1 (40 Marks)

You will be working with the GitHub repository "ULTRA: UnLimited TRAnsfers for Multimodal Route Planning"(link), which contains various state-of-the-art shortest-path algorithms. Additionally, you have been provided with a graph of Florida in the DIMACS format (link). Below is a summary of the DIMACS format. For more details on it, refer to link.

```
c file: TestCase1—Sample test case for C++ Developer position
c source: Prateek Agarwal (prateeka@iisc.ac.in)
p sp 4 6
a 1 2 5
a 1 4 1
```

- Comments. Lines starting with "c" are comments, providing human-readable information and are ignored by programs.
- Problem line. The line starting with "p" specifies the problem type ("sp" for shortest path), the number of nodes "n", and the number of edges "m" in the graph.
- Edge Descriptors. Each edge is described using "a u v w", indicating an edge from node "u" to node "v" with weight "w".

Your task is to create a file named `question1.cpp` inside the `Runnables` folder. When compiled, `question1.cpp` runs Dijkstra's algorithm implemented in the repository for 200 randomly selected source-destination pairs on the Florida graph. Note that `question1.cpp` calls Dijkstra's algorithm already implemented in the repository. Do not code your own Dijkstra's algorithm. `question1.cpp` prints the total runtime in seconds in the following format:

```
Total runtime in seconds for 200 random Dijkstra:  x
```

Hints:

- Here are a few test cases for you to verify the Dijkstra's output:
  - Source: 264345, Target: 264327
    Shortest Path: 264345 - 269065 - 264331 - 264311 - 264324 - 264327
    Shortest path length: 10283
  - Source: 1234, Target: 1272
    Shortest Path: 1234 - 1235 - 1228 - 1226 - 1238 - 1247 - 1265 - 1272
    Shortest path length: 11158
  - Source: 0, Target: 1
    Shortest Path: 0 - 1
    Shortest path length: 14188
- A node labeled x in the DIMACS graph, will be labeled x-1 in the custom binary format. For example, node labeled 1 in the DIMACS graph will be represented as node 0 in the custom binary format. The sample cases provided above are w.r.t. custom binary format.
- You can spend weeks understanding all the algorithms. However, the question expects you to focus on identifying how and where Dijkstra's algorithm is being used in the codebase and replicate the process. The question can be completed in less than 25 lines of code!

Question 2 (60 Marks)

Dijkstra's algorithm, while effective, can be slow in real-world applications. Two popular speed-up techniques are often employed to address this are Bi-directional Search and A* Search.

- Bi-directional search: In this approach, we simultaneously run a forward search from the source node and a backward search from the destination node. Forward search involves running normal Dijkstra's algorithm starting from the source node. The backward search involves applying Dijkstra's algorithm on the reversed graph, i.e., a graph with the same set of nodes but with reversed edges. The idea is to reduce the search space. The forward and backward searches can either be adopted alternatively, based on the priority of nodes or any other method.

- A* Search: A* Search uses a *potential function* in addition to the distance label of nodes to order nodes in Dijkstra's priority queue. For a shortest path query, the potential function of a node is chosen to be a lower bound of the shortest path distance between the source and target. We can either use a modified Dijkstra in which nodes are ordered by the sum of their distance from the source to a node and the potential function value of that node or modify the edge weights in a preprocessing step by adjusting them based on the potential function values of the nodes they connect. A good potential function would thus lower the priority of nodes lying on the shortest path between the source and the target.

With the above knowledge, perform the following tasks:

- Copy the logic for Dijkstra's algorithm into a new file and modify it to implement either the Bi-directional Search or A* Search principle. You may choose either approach based on your convenience and understanding.

- Create a file named `question2.cpp` located inside the `Runnables` folder. When compiled, `question2.cpp` runs your modified Dijkstra's algorithm for 200 randomly selected source-destination pairs on the Florida graph. `question2.cpp` prints the total runtime in seconds in the following format:
  `Total runtime in seconds for 200 random modified Dijkstra:  x`

Hints: For more information on Bi-Directional and A* search, refer: MIT lecture, Reference 1, Reference 2. The question tests your research aptitude and critical thinking.