Algorithms and Data Structures: Homework #12

Due on May 4, 2020 at 23:00

Henri Sota

Problem 12.1

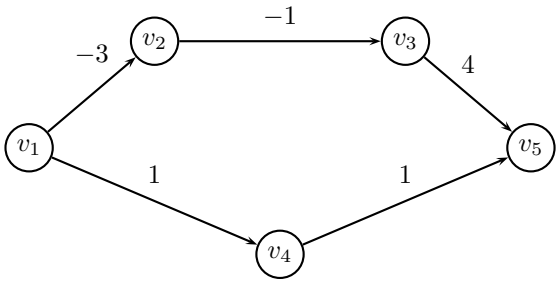
Your friend (who has not taken an "Algorithms and Data Structures" course) asks you for help on implementing an algorithm for finding the shortest path between two nodes u and v in a directed graph (possibly containing negative edge weights). The friend proposes the following algorithm:

1) Add a large constant to each edge weight such that all weights become positive.

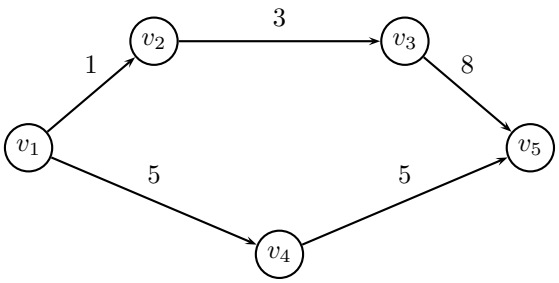
2) Run Dijkstra's algorithm for the shortest path from u to v.

Prove or disprove the correctness of the above algorithm to find the shortest path (note that in order to disprove, you only need to give a counterexample).

In order to prove the incorrectness of the aforementioned algorithm, an example where the algorithm fails is provided:
Let G be a graph made up of a set of directed edges between the vertices in the vertex set, $V = (v_1, v_2, v_3, v_4, v_5)$, accompanied with their weights, $E = ((v_1, v_2, -3), (v_2, v_3, -1), (v_3, v_5, 4), (v_1, v_4, 1), (v_4, v_5, 1)$, where the direction is pointing in the sense from the first vertex to the second vertex of the edge and the weight is the third element of the edge.



Directed graph before adding constant to weights



Directed graph after adding constant to weights

In the example above, v_1 is the start of the path, u, and v_5 is the end of the path, v. The shortest path that starts from node v_1 and ends at node v_5 is the path with weight 0, that goes through node v_2 and node v_3. Adding a constant, c, to each path in order to make all of the path weights positive, such that $c \in \mathbb{N}, c > 3$, the shortest path now becomes the one that goes through node v_4, with path weight 10, smaller than the weight 12 of the previous shortest path after adding the constant. Therefore, the algorithm mentioned above is not correct.

Problem 12.2

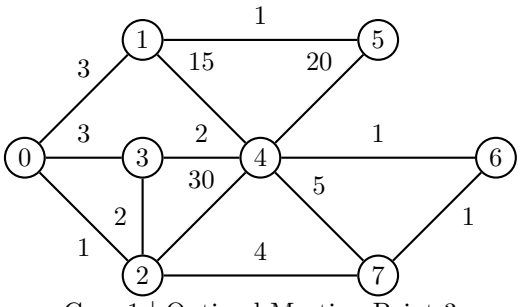
You are trying to meet your friend who lives in a different city than you and you want to meet in a city that is between where either of you live. Time is limited and you are trying to minimize the time spent traveling. So, where exactly should you meet?

You are given a graph G with nodes $V = \{0, \ldots, n-1\}$ that represent the cities and edges E that represent streets connecting the cities directly. The edges are associated with the distance $d(e)$, which is the time needed to travel between two cities. You are given your own city p and your friend's city q with $p, q \in \{0, \ldots, n-1\}$. Implement an algorithm that finds the target city m in which you and your friend should meet in order to minimize travel time for both of you (you drive towards your meeting city simultaneously, so if you travel for x minutes and your friend travels for y minutes, then you will want to minimize $\max(x, y)$). The graph is given to you with an adjacency matrix, where each entry x_{ij} represents the time (in minutes) that it takes to travel from city i to city j. Naturally, the indices are the nodes. The algorithm should return the target city $m \in \{0, \ldots, n-1\}$.

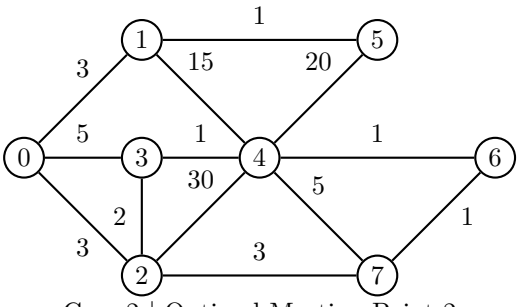
The prototype of the corresponding function should be similar to:

```
int find_meetup_city(int[][] adj_matrix, int your_city, int friend_city);
```

My implementation of `find_meetup_city` function is able to find the optimal meeting point by combining the results of running Dijkstra's algorithm from both cities, your_city and friend_city. During the first run of Dijkstra, all the cities are discovered and the distance to be minimized by the algorithm is equal to the shortest path to your_city as Dijkstra first is run starting from your_city. During the second run of Dijkstra, because every city has been discovered, the distance to be minimized is set to the maximum shortest path coming from both shortest paths, from your_city and friend_city, to the current city. After both runs of Dijkstra's algorithm, all the cities hold their maximum shortest distance from both cities. The final step is to find the minimum of this array holding these distances.



Case 1 | Optimal Meeting Point 3



Case 2 | Optimal Meeting Point 2

Implementation can be found in the file "optimalMeetingPoint.cpp", inside the "optimalMeetingPoint" directory. Test cases for both cases given above are found in the file "testcases.txt", inside the "optimalMeetingPoint" directory.

Problem 12.3

Consider a puzzle that consists of a $n \times n$ grid where each field contains a value $x_{ij} \in \mathbb{N}$. Our player starts in the top left corner of the grid. The goal of the game is to reach the bottom right corner with the player.
Rules of the game: On each turn, you may move your player up, down, left or right. The distance by which the player moves in a chosen direction is given by the number of its current cell. You must stay within the board (you cannot go off the edge of the board).
Example: If your player is on a square with value 3, then you may move either three steps up, down, left or right (as long as you do not leave the board).

a) Represent the problem as a graph problem. Formalize it by determining what is represented as the nodes and as the edges.

   In order to represent the problem as a graph, definitions for the vertices and edges of the graph must be given. In this case, the vertices can be represented by the cells of the puzzle board (Fields in the implementation) and the edges can be represented by the possible steps taken from one vertex to another, such that the step is valid, within the bounds. The edges can therefore represent the steps between $Field_{i,j}$ and the Fields, $Field_{i-step,j}$, $Field_{i,j+step}$, $Field_{i+step,j}$ and $Field_{i,j-step}$, if the step is valid in that particular direction. The puzzle board would contain the whole set of vertices such that for each cell, there is an equivalent node on the graph. Each cell would also hold a value, steps in the implementation, that contains the minimum number of steps taken to arrive at that cell starting from the upper left cell. The upper left cell would be the initial cell with steps set to 0. Therefore the problem can be modelled to ask for the shortest path, the least number of steps, in this graph that starts from the upper left cell and ends at the bottom right cell of the puzzle board.

b) Implement the class `PuzzleBoard` similar to class declaration shown below.

```
class PuzzleBoard {
  private:
    // your choice
  public:
    // Subpoint (b)
    PuzzleBoard(int boardSize, int[][] fields = null);
    /
    * constructor should create the
    graph (as you defined it in subpoint (a) with the values from fields.
    If fields is null, then initialize the fields of the board with
    random values between 1 and boardSize-1. */
    bool makeMove(int direction);
    /
    * makes the move (if valid), direction is 0 for up, 1
    for right, 2 for down, 3 for left. Returns true if the move was
    valid, false otherwise. */
    bool getResult();
    /
    * Returns false if game is not over yet, true if puzzle was solved */
    std::ostream &operator<<(std::ostream &os, PuzzleBoard const &m);
    /
    * in Python, this is the __str__ method. */
    // Subpoint (c)
    int solve();
    /
    * returns the minimum number of moves needed to solve the puzzle,
    and -1 if it is not solvable. */
}
```

Implementation can be found in the file "PuzzleBoard.cpp", inside the "numberMaze" directory. One test case has been provided as input and is found in the file "testcases.txt", inside the "numberMaze" directory.