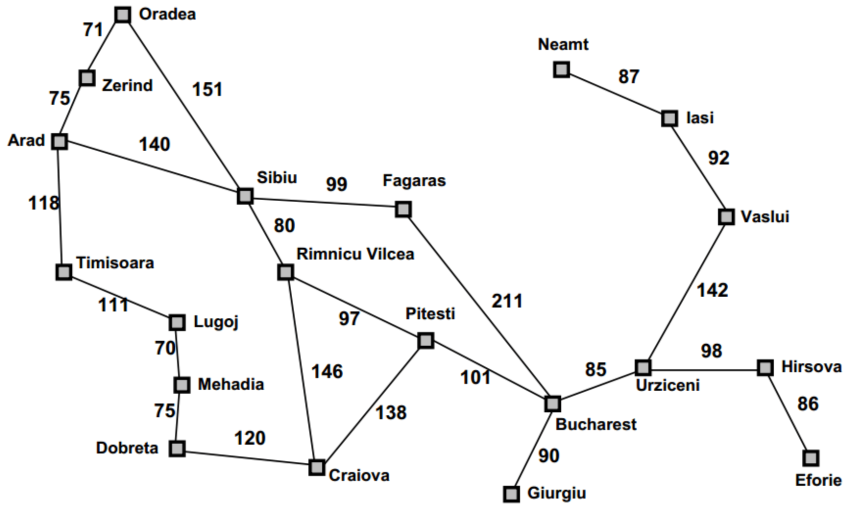
Introduction to Algorithms **F2021**

Problem Set 3

Handed Out: November 1st, 2021 *Due: November* 15th, 2021

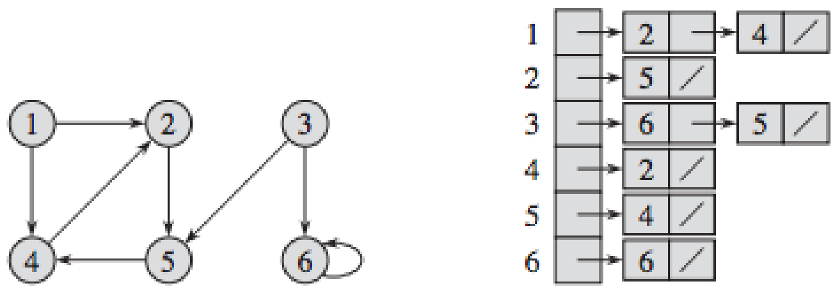


In this assignment will be focused on graphs! Well, one graph in particular: a map of cities in Romania! This is a relatively well known map in Computer Science circles because it is often used to introduce graph algorithms, and now it's our turn to play with it. In this assignment, you will be implementing 2 graph algorithms: Breadth First Search and Dijkstra's Algorithm. I have provided you with two text files that contain the information required to construct this graph: RomaniaVertices.txt and RomaniaEdges.txt. As implied by the name, RomaniaVertices.txt contains the name of every vertex in the graph, and RomaniaEdges.txt contains information about every edge in the graph. RomaniaVertices.txt is pretty self explanatory in terms of file structure (each vertex name is on its own line); however, RomaniaEdges.txt is a bit more complex. RomaniaEdges.txt is a comma separated text file in which each row represents a different edge. The first two entries on a row represent the two vertices connected by the edge, and the last element in the row is the weight associated with that edge. You will only use this weight for some parts of the assignment. I will indicate when this is necessary. This graph is UNDIRECTED. To represent this in an adjacency list, you can simply add duplicate edges that indicate that directionality goes two ways.

As always, you will submit a writeup in Word or PDF that summarizes your results and all code as a zip file. Submit the writeup (with attached source code) to the Canvas submission locker before 11:59pm on the due date.

Represent the Graph as an Adjacency List (20 Points)

The first part of the assignment involves simply reading in the vertex and edge data for the graph and representing it as an adjacency list. Recall, an adjacency list is, essentially, a hash table in which each element in the hash table refers to one of the vertices in the graph and points to list containing all adjacent vertices. An image of an example adjacency list can be seen below.



You are free to use any data structure to implement this adjacency list, as long as it's not something like std::adjacencyList (which I'm not sure even exists). As long as the data structure you use was not specifically designed for graph computation, then it should be fine to use. Some examples of reasonable data structures include: Hash Tables/Dictionaries, Lists of Lists/Vectors of Vectors, Vector of Arrays, etc.

You should also implement a method to print out your graph. It should print output in the following format with each vertex getting its own line: *vertexName -¿ edge1,edge2,edge3.* This output should be included in the writeup

- For this assignment, you are allowed to use some high level libraries to implement your adjacency matrix, as long as they are not specifically designed with graph computation in mind. If you're unsure if a library is allowed, just ask and I'll let you know.
- As mentioned at the start of class, all code should be written in Python 3.8 OR C++. If you'd like to use another language, just ask and I will let you know if it's okay. There have been some unexpected errors that have occurred involving compiling code on the TA's machine. We've narrowed Python issues down to newer (Python 3.9+) distributions of Python. As such, please do not use Python 3.9+ to implement this assignment. For C++, be aware that we are compiling and running these assignments on Windows machines. Please ensure that your code runs on a Windows machine to ensure that things go smoothly. I'm looking into getting access to a Mac purely for grading purposes, but it's not clear that this will materialize in a timely manner.
- Please include instructions for how to compile and run your code in your writeup.
- Explain any implementation choices you had to make in the final report (such as underlying data structure), and also include the text representation of your graph (described above)!

Breadth First Search (30 points)

In this part of the assignment, we're going to implement a class graph search algorithm: Breadth First Search! Using the adjacency list you created in Part 1, implement breadth first search. For this part of the assignment, you WILL NOT be using edge weights. The breadth first search algorithm that we discuss in class makes use of several auxiliary data structures, including a queue to keep track of the search process. You are allowed to use any data structure you'd like for these auxiliary structures, within reason (nothing that can just solve the problem with a single method call, but I don't think that will be an issue). This means that you can use pre-existing queue implementations (which both Python and C++ have) if you'd like!

Along with the BFS algorithm, you will need to implement the PRINT-PATH method outlined in the book. This will allow you to print out the shortest path from some starting vertex, s, to any other vertex, v. Using this method combined with your BFS implementation, print out the shortest path from Arad to Sibiu, from Arad to Craiova, and finally from Arad to Bucharest.

Details:

- Feel free to use pre-existing data structures for your auxiliary data structures. As always, don't use methods that can, essentially, answer the question in one method call.
- As mentioned at the start of class, all code should be written in Python 3 (not Python 3.9+) OR C++. If you'd like to use another language, just ask and I will let you know if it's okay.
- Please include instructions for how to compile and run your code in your writeup.
- Explain any implementation choices you had to make in the final report (such as how ties were broken).
- Be sure to include your shortest paths in the writeup!

Dijkstra's Algorithm! (30 points)

Dijkstra's Algorithm! This is probably one of the most iconic graph algorithms out there. It calculates the single-source shortest path on a weighted graph. As you can probably tell, we're going to be using edge weights for this part of the assignment. In this part of the assignment, implement Dijkstra's algorithm. As with the previous part, you are allowed to use any pre-existing data structure (within reason) to implement the auxiliary data needed by this algorithm. A notable example of this for Dijkstra's algorithm is the min priority queue. Dijkstra's algorithm makes use of a min priority queue for sorting path lengths. For this assignment, you can use pre-existing implementations of min priority queues (which both C++ and Python have).

Once you've implemented Dijkstra's, use it to find the shortest path from Arad to Bucharest. You should be able to print out this path using a method similar to (possibly identical to, depending on your implementation) the PRINT-PATH method used in part 2. Is this path different from the path from Arad to Bucharest you found in part 2? Why or why not?

Details:

- Feel free to use pre-existing data structures for your auxiliary data structures. As always, don't use methods that can, essentially, answer the question in one method call.
- As mentioned at the start of class, all code should be written in Python 3 (not 3.9+) OR C++. If you'd like to use another language, just ask and I will let you know if it's okay.
- Please include instructions for how to compile and run your code in your writeup.
- Explain any implementation choices you had to make in the final report (such as how ties were broken).
- Remember to include the shortest path and answers to questions in the writeup!

Writeup (20 points)

You will include a written report with your submission detailing important details about your implementation, as well as the results of any analyses requested in the assignment. The report must be complete and clear. A few key points to remember:

- Complete: the report does not need to be long, but should include everything that was requested.
- Clear: your grammar should be correct, your graphics should be clearly labeled and easy to read.
- Concise: I sometimes print out reports to ease grading, don't make figures larger than they need to be. Graphics and text should be large enough to get the point across, but not much larger.
- Credit (partial): if you are not able to get something working, or unable to generate a particular figure, explain why in your report. If you don't explain, I can't give partial credit.

Bonus: Prim’s Algorithm (10 points)

Prim’s algorithm is commonly used to construct minimum spanning trees. For extra credit, implement Prim’s algorithm on the Romania graph. All rules from previous parts apply. Feel free to use pre-existing data structures to implement auxiliary structures required by the algorithm. Once you’ve implemented this algorithm, print the spanning tree by printing the full predecessor structure.