Timur Garifullin
Due: 11:59pm Thursday, Feb 9

Homework 4

Spring 2023, Bhakta
CS 315, Algorithms

This assignment covers Breadth First Search, Dijkstra's algorithm, and Graph problem solving.

For these questions, you may need to submit graph drawings. You should either

- Learn how to use the TikZ markup to create graph drawings in LaTeX. This isn't too hard (there are examples in this homework), but the other options are easier.
- Draw a graph using other drawing software, and use the includegraphics command to add it to your pdf file. You could also draw a graph on paper, take a picture, and use includegraphics to add the picture to the final pdf document.

0. (0 points) (NOT OPTIONAL) Who did you work with on this homework? What sources did you consult? How long did this assignment take?

Solution:
Consulting: Artur Idrissov, Padmaja Karki
Source: Class Notes
Time Spent: 23 hours

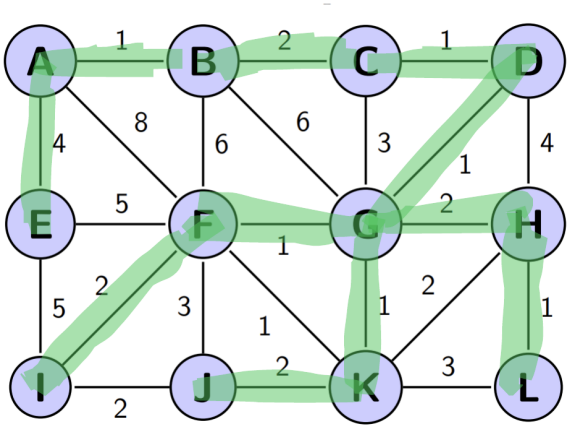
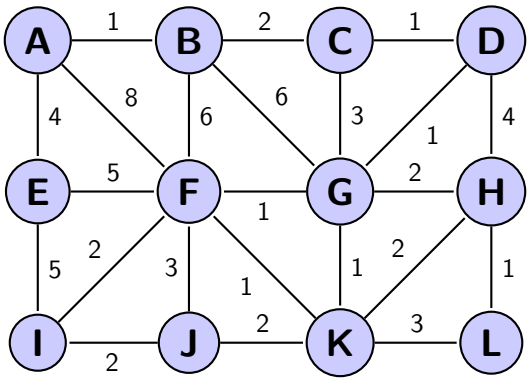


Figure 1: Graph with final shortest paths

1. (20 points) Run Dijkstra's algorithm on the following graph, starting at node A.

   I want you to show the full execution of Dijkstra's algorithm over time by showing the current estimated distance from the start vertex to every other vertex at each step of the algorithm (as was done in class on Thursday). You can do this in a separate table (as was done in class) or on the graph directly by showing multiple copies of the graph, one for each step.

   Draw the final shortest path tree that marks all the relevant edges used by Dijkstra's algorithm.



Solution: Solutions are shown on the Figure 1 (above) and Figure 2 (below)

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
A	0	0	0	0	0	0	0	0	0	0	0	0	0
B	∞	1	1	1	1	1	1	1	1	1	1	1	1
C	∞	∞	3	3	3	3	3	3	3	3	3	3	3
D	∞	∞	∞	4	4	4	4	4	4	4	4	4	4
E	∞	4	4	4	4	4	4	4	4	4	4	4	4
F	∞	8	7	7	7	7	6	6	6	6	6	6	6
G	∞	∞	7	6	5	5	5	5	5	5	5	5	5
H	∞	∞	∞	∞	8	8	7	7	7	7	7	7	7
I	∞	∞	∞	∞	∞	9	9	8	8	8	8	8	8
J	∞	∞	∞	∞	∞	∞	∞	9	8	8	8	8	8
K	∞	∞	∞	∞	∞	∞	6	6	6	6	6	6	6
L	∞	∞	∞	∞	∞	∞	∞	∞	9	8	8	8	8

Figure 2: Table showing steps of Dijkstra's algorithm from Exercise 1

2. (20 points) In Dijkstra's algorithm, all edge weights have to be ≥ 0. Dijkstra's algorithm doesn't work correctly when there are negative weighted edges in the graph.

   For example, in an extreme case, there could be a cycle (like a to b to c to d back to a) in the graph with total negative weight. In this case, the concept of shortest path isn't even well defined, because you could go around this cycle infinitely many times and reduce the total cost of any path to negative infinity.

   In a less extreme case, even without negative sum cycles, Dijkstra's algorithm may still fail. Construct a graph *without* negative sum cycles, only one negative edge, and choose a start vertex in that graph where Dijkstra's algorithm will fail to find the correct shortest path to all vertices. Explain this construction.

Solution: In the example graph (Figure 3) and a supporting table (Figure 4) on the On the next page I am showing a case when Dijkstra's algorithm fails.

At time 3, the algorithm has a choice between two paths that both have a length of 3. At that moment we are at vertex c and the algorithm will likely choose its neighbor e, which results in the path (a, b, c, e) of length 3 from vertex a to vertex e.

However, there is a shorter path between a and e vertices. Path a, b, d, e has a length of 2.

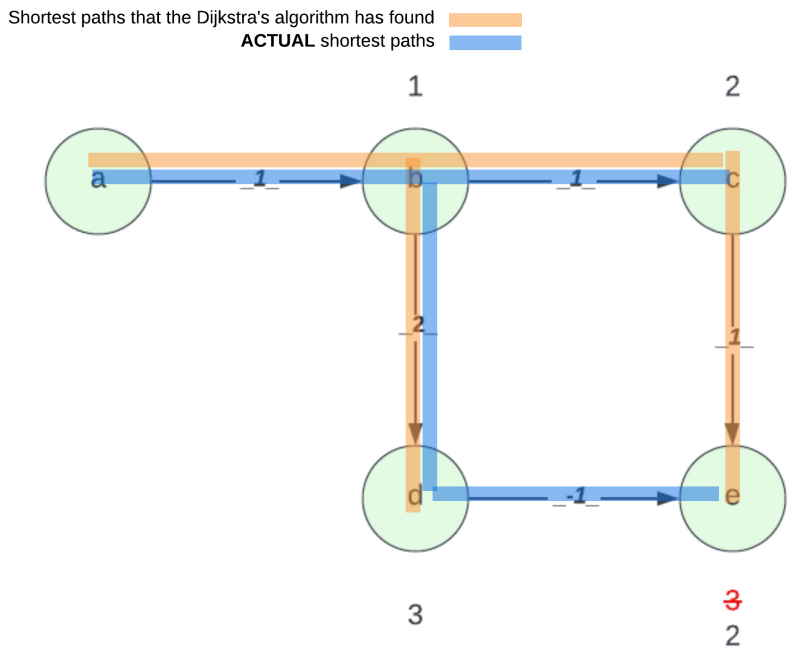


Figure 3: The actual shortest path vs the shortest path that the Dijkstra's algorithm would choose .

Time	0	1	2	3	4	5
A	0	0	0	0	0	0
B	∞	1	1	1	1	1
C	∞	∞	2	2	2	2
D	∞	∞	3	3	3	3
E	∞	∞	∞	3	3	3

Figure 4: Table showing every step of Dijkstra's algorithm

3. (EXAMPLE) You are given an unweighted, directed graph G with two types of edges - the red edges E_r and the blue edges E_b, and you are trying to find a path in the graph from vertex s to vertex t. The catch: your path must alternate using red edges and blue edges. The first edge in the path must be red, and the last edge in the path must be blue.

   Give an algorithm that can find the shortest such path, if one exists. You should give a clear description of your algorithm in words, and you should explain its running time. Faster (in O notation) and correct is worth more credit.

   Here is an example with 5 vertices, where vertex 1 is s and vertex 5 is t. The red edges E_r and the blue edges E_b are as follows:

   $E_r = 1 \to 2; 3 \to 4, 4 \to 3, 4 \to 5.$
   $E_b = 1 \to 3; 2 \to 3, 2 \to 4; 4 \to 3, 3 \to 5.$
   There is a red-blue alternating path of length 4 to go from 1 to 5: $1 \to 2 \to 4 \to 3 \to 5$.

   For this problem, you may repeat vertices in your shortest path.

   Hint: Construct a new graph G' from G and run a known algorithm on G'.

   Solution: One possible solution is to modify BFS so that it alternates red and blue edges, and keeps track of the shortest distance to each vertex that ends on a red edge and that ends on a blue edge. However, this is tedious to get the pseudocode right, and would require a proof of correctness.

   An easier to explain algorithm works as follows. Create a new graph G' from $G = (V, E)$ as follows:

   Create two sets of vertices V_L and V_R each of which are a copy of V. For each vertex v in V, we will use the notation v_l to represent the copy of v in V_l, and similarly v_r for the copy in V_R. For each red edge $(u, v) \in E_r$, create a directed edge in G' from u_l and v_r. For each blue edge $(u, v) \in E_b$, create a directed edge in G' from u_r and v_l

   By construction, the only paths possible in this graph alternate between V_L and V_R, since the only edges leaving V_L go to V_R and vice-versa. Also, by construction, the edges leaving V_L are the red edges of the original graph, and the edges leaving V_R are the blue edges of the original graph. Therefore, ordinary paths in the graph G' must correspond to alternating red-blue paths in G. So, there is a path from V_L's copy of s to V_L's copy of t in G' if and only if there is a corresponding path in the original graph from s to t that starts with a red edge, ends on a blue edge, and alternate in between, as desired.

   So our final algorithm is:

   - Construct G' as described above
   - Run BFS starting at s_l, recording the distance to all other vertices of G'.
   - If t_l is reached: Return the distance to t_l.
   - Else: return "Impossible"

   It takes $O(|V| + |E|)$ steps to create G'. Since G' has $2|V|$ vertices and $|E|$ edges, the BFS step will take $O(2|V| + |E|) = O(|V| + |E|)$ time. Therefore the total runtime is $O(|V| + |E|)$.

4. (20 points) Design an algorithm to solve the following problem.

   You are given a graph G with some of the edges marked *red*. These red edges are toll roads, and you only have enough money for k tolls. Given G, nodes s and t, and input parameter k, give an algorithm that can determine if there is a way to travel from s to t using at most k toll roads.

   Explain why your solution works, and analyze the total runtime of your algorithm in terms of $|V|, |E|$. Remember to include the costs of constructing the new graph and running your algorithm on the new graph. There are at least two different, reasonable ways to do this problem. Your final answer must have a runtime that is polynomial in $|V|, |E|, k$.

   Hint: First find a solution that will work when $k = 1$.

Solution:
1. Construct a new graph G', by iterating through the original graph's edges to identify and change the weights of the red edges (toll roads) to 1. In addition, assign the weights of the blue edges (free roads) to 0.
2. Run the Dijkstra's algorithm on the G' starting from vertex s. Implement a heap priority queue.
3. The goal is to travel from s to t using at most k toll roads.
- The algorithm will return True if the weight of the path from s to t is less or equal to k. Therefore, it is possible to find a path from vertex s to vertex t using at most k toll roads.
- The algorithm will return False if the weight of the path from s to t is greater than k. Therefore, it is impossible to find a path from vertex s to vertex t using at most k toll roads.

Runtime of creating new graph $G' : O(E)$
Runtime of the algorithm: $O((V + E) \log V)$

On the next page, you can find an example of a possible graph G and G'.

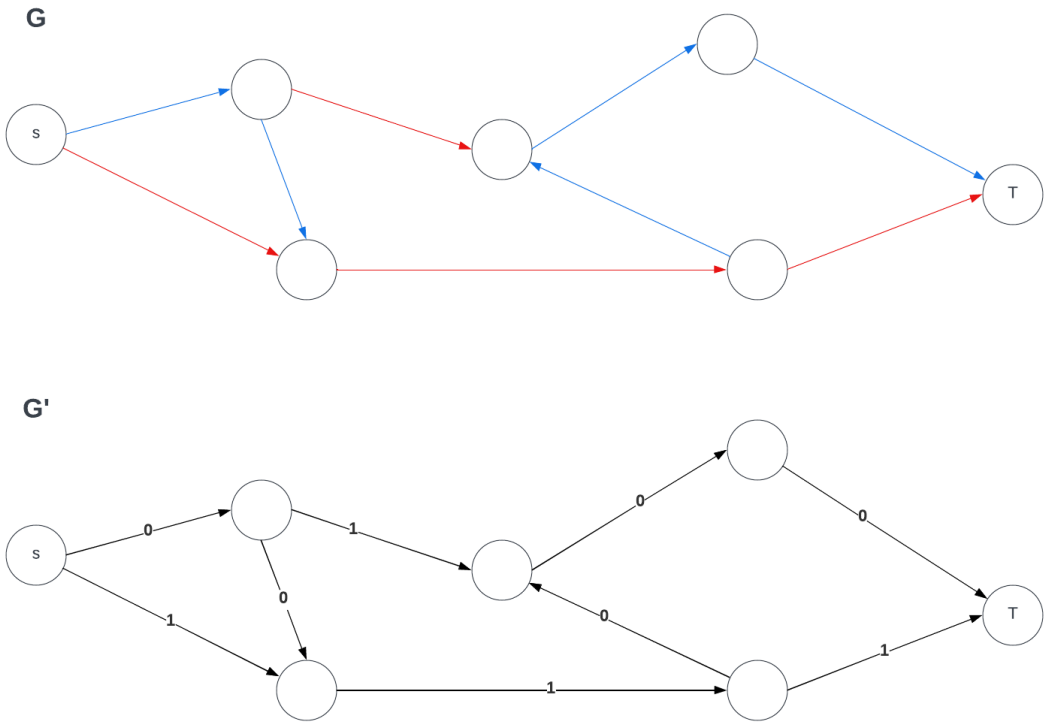


Figure 5: Original graph G and the new graph G'

5. (40 points) You are given an unweighted, directed graph G that represents a map, and you have an electric car. Some of the vertices, including the start vertex, are marked as Charging Stations. Your car can only go at most k edges between visits to a charging station. You want to find the shortest path from vertices *start* to *end* with the requirement that you can't go more than k edges without visiting one of the marked charging station vertices.

   (a) Find an algorithm to solve this problem that takes your initial vertex labeled graph, constructs a new graph from that information, and then runs a classic algorithm on that new graph. Your total runtime must be some polynomial in $|V|, |E|, k$. There are at least two, different, approaches that I know for this problem - either approach is fine. Explain why your solution works, and analyze the total runtime of your algorithm. Remember to include the time and space costs of constructing the new graph and running your algorithm on the new graph.

Solution:

1. Read the input data including the number of vertices n, number of connections m, number of charging stations c, and maximum reachable distance k.

2. Run a BFS traversal from each charging station to construct a new graph denoted as G'.

3. During the traversal, record the distance from the charging stations to the closest encountered charging station.

4. If, within a distance of k edges, a charging station is found, add the path into the new graph G'. Note we designate charging stations as vertices and the distances between them as edge weights.

5. Discard any paths surpassing k edges.

6. Execute a Dijkstra's algorithm on the newly formed graph G' starting from the initial vertex s and concluding at the terminal vertex t to identify the shortest path from s to t, ensuring that charging stations remain within the specified distance of k edges.

5. If a valid path is found within the maximum distance, return the shortest distance; otherwise, return "Impossible".

Runtime:
Creating new graph G': $O(E)$
Running the algorithm: $O(n(V+E) + |E|log|V|)$
Running BFS for several vertices: $O(n(V+E))$
Dijkstra: $O(|E|log|V|)$
Space complexity: $O(V + E + C^2)$

(b) Then, write code that will actually solve this problem.

Your code should read the input information from standard input (using input() in python, a Scanner(System.in) in Java, or cin in c++). Your code should write the information to standard out, (print in Python, System.out.println in Java, or cout in c++). If you wish to use a different language than these three, check with me first.

This is the format of the input:

The first line has four space separated integers: n m c k, where n is the number of cities (0 - (n-1)), m is the number of roads between cities, c is the number of charging stations, and k is the maximum number of edges that can be travelled between charging station visits. You may assume that city 0 is the start vertex and city $n-1$ is the destination vertex.

The next line contains c space separated integers, labelling the charging stations.

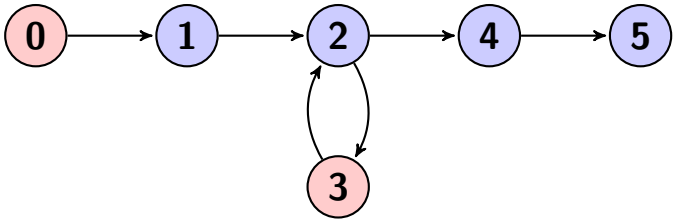
The next n lines $i = 0...n-2$ have a list of the neighbors of vertex i. You may assume that there is at least one outgoing edge for each of vertices $0...n-2$, and the destination vertex $n-1$ has no outgoing edges.

For example, you might see an input that looks like this:

```
Sample input:
6 6 2 3
0 3
1
2
3 4
4
5
Expected output:
6
```

This means that there are 6 vertices, 6 edges, 2 charging stations, and you can travel up to k = 3 edges before visiting a charging station. The charging stations are 0 and 3. The neighbors of vertex 0 are 1. The neighbors of vertex 1 are 2. The neighbors of vertex 2 are 3 and 4. The neighbors of vertex 3 are 4. The neighbors of vertex 4 are 5.

Therefore, this input is encoding the below graph (charging stations are marked red):



This input has a solution of length 6. Your program should simply print the number 6 which gives the fewest steps needed to reach the destination in this scenario. If it's impossible, print out the word "Impossible".

Submit your code along with your compiled PDF. Here, cite where you sourced your implementation of BFS/Dijkstra's/etc. (It might be from me).

Solution: BFS, Dijkstra's algorithms - Author: Prateek Bhakta
The code is included as a separate file

6. (Bonus) Kattis is a programming puzzle website, similar to popular websites like leetcode and topcoder, but with a focus on programming contest level problems.

   This problem is a harder version of the car transit problem in question 5.

   https://open.kattis.com/problems/fulltank

   Hints: You're trying to minimize the amount you spend. The notion of "what's the cheapest way to get to vertex x" is not well defined, because it depends on how much gas you have left. So instead, think about "what's the cheapest way to get to vertex x with k gallons left in the car.

   Because of the large size requirements, probably use a faster language like C++ rather than Python. To get faster runtimes, you may want to modify Dijkstra's algorithm instead of creating a new graph if this becomes an issue.

   As before, submit an explanation of your solution, working code, and a screenshot that you passed all test cases.

   Solution: