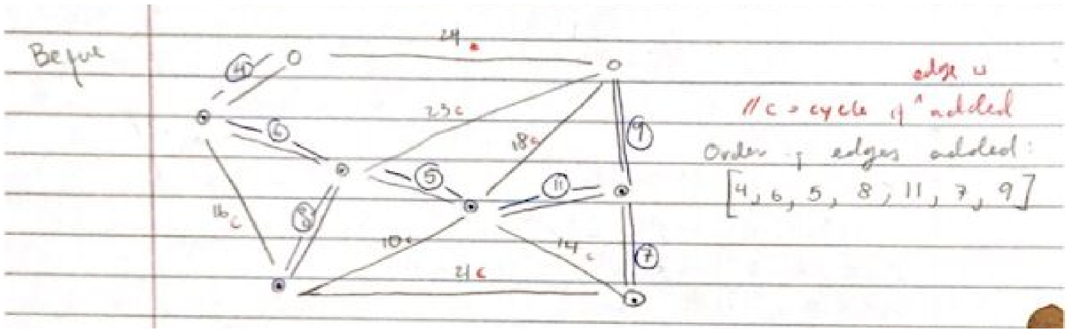
Catherine Kuntoro
CS 146 Section 7
Professor Aniket Chandak
6 May 2020

Homework 5 Questions

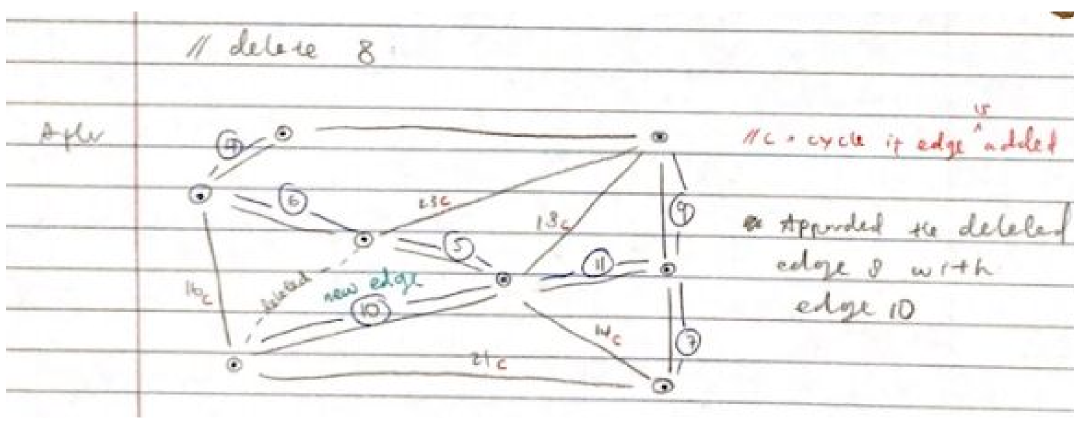
Problem 2: Given an MST for an edge-weighted graph G, suppose that an edge in G that does not disconnect G is deleted. Describe how to find an MST of the new graph in time proportional to E.

Assuming that the MST of the original edge-weighted graph G is a valid MST and the deleted edge does not disconnect G, there are two cases when updating the MST of G after one edge has been deleted. The first case is where the deleted edge is not part of the G's MST. Since the deleted edge is not part of the MST, the MST does not need any updates and stays the same.

The second case is that the deleted edge is part of the MST. In this case, the MST must change by replacing the deleted edge with the next minimum weight edge. Below are diagrams to illustrate this change:



The diagram above showed a graph G before any changes are made to it. Since one of the properties of MST is that it must be acyclic, some edges are marked with the letter "c" to symbolize that the addition of that edge will cause a cycle. The order of the edges added into MST is also listed in the hard brackets.


// delete 8
After
//c = cycle if edge 15 added
Appended the deleted edge 8 with edge 10
new edge
deleted
15c
13c
16c
14c
21c

In the diagram above, the edge with the weight 8 is deleted. The edge that replaces 8 is the edge with the weight 10.

Therefore, to replace an edge in the MST, select the next minimum weight edge that will reconnect the nodes that were disconnected because of the deletion of the edge. In the case of the diagram, it was edge with weight 10 since it is the next lightest edge after the deleted edge with weight 8.

Assuming that we have access to an Iterable data structure that stores all of the edges in graph G, to perform the replacement of the missing edge in the MST, iterate to the edge that is to be deleted. Then, take the edge adjacent to the edge to be deleted, and set it as the new edge of the MST. The new edge of the MST will be valid since no other nodes are deleted and the MST is already valid. Thus, it is safe to take the next minimum weight edge after the deleted edge as part of the MST.

Since we just need to iterate through the Iterable data structure that stores all of the edges of G once, therefore we could perform this replacement in time complexity of O(E), where E is the total number of edges.

Problem 3: Given an MST for an edge-weighted graph G and a new edge e, describe how to find an MST of the new graph in time proportional to V.

There are two cases when handling the addition of a new edge e into the edge-weighted graph G. The first case is the weight of the new edge e is larger than any of the edges in G's MST. In this case, there will be no changes to G's MST.

The second case is when e's weight is less than one of the edges in the MST. Assume that we have access to an Iterable data structure that stores all of the vertices in G. First, iterate to the vertex “v” that the new edge “e” is placed for. Then, if one of “v”’s edges is part of the MST, say its name is “x” and “x” is in the MST, compare “x” with “e”. If e’s weight is less than “x”’s weight, make “e” one of the edges in the MST and take “x” out from the list of edges of the MST. If “x”’s weight is less than “e”’s weight, then iterate to the next node that “e” connects to (since an edge connects two vertices). In that second vertex, perform the same check as we did for the first vertex that edge “e” connects to. If both cases result in no changes to the MST, then that means the weight of the new edge is larger than what the MST has for those two vertices, and thus should not be added to the MST. Otherwise, edge “e” will replace one of the edges that belong to one of “e”’s vertices.

Since we only need to iterate through an Iterable data structure that contains all of the vertices in graph G, the time complexity of updating the MST after a new edge has been added is O(V), where V is the total amount of vertices the graph G has.

Problem 4: Explain why the Dijkstra algorithm does not work with negative edges with examples

To calculate the distance to vertex “w” from vertex “v”, the following calculation to relax the edge between vertex v and w is performed:

distTo[w] <= distTo[v] + edge.weight()

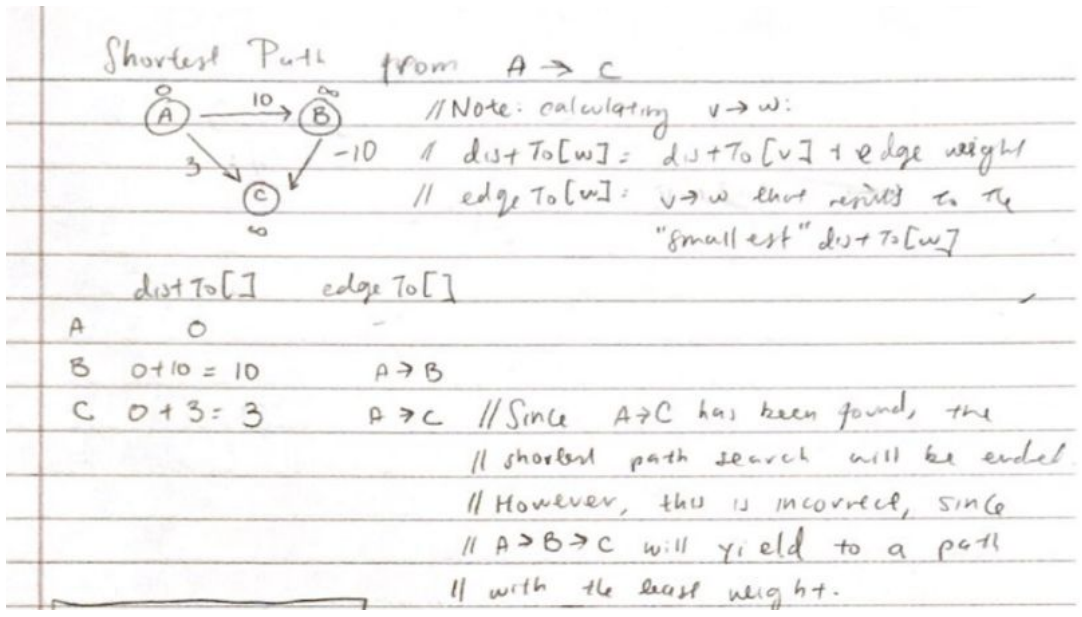
This inequality usually holds true since we expect that logically, the distance to vertex w should decrease the closer one is to vertex w. Therefore, the distTo[w] represents the distance of the shortest path from (v, w).

However, this inequality will not be true if there is a distTo[w] that is larger than distTo[v] + edge.weight(), as in:

distTo[w] > distTo[v] + edge.weight() // Contradiction of the inequality shown above

This contradiction implies that there exists a path “distTo[v]+edge.weight()” that is shorter than disTo[w], which is supposed to hold the shortest path.

An example is provided in the diagram below:



Shortest Path from A → C

$$\begin{tikzcd}
\overset{0}{A} \arrow[r, "10"] \arrow[dr, "3"'] & \overset{\infty}{B} \arrow[d, "-10"] \\
& \underset{\infty}{C}
\end{tikzcd}$$

//Note: calculating v → w:
// distTo[w] = distTo[v] + edge weight
// edgeTo[w]: v→w that results to the "smallest" distTo[w]

	distTo[]	edgeTo[]
A	0	-
B	0+10 = 10	A→B
C	0+3 = 3	A→C

//Since A→C has been found, the
// shortest path search will be ended
// However, this is incorrect, since
// A→B→C will yield to a path
// with the least weight.

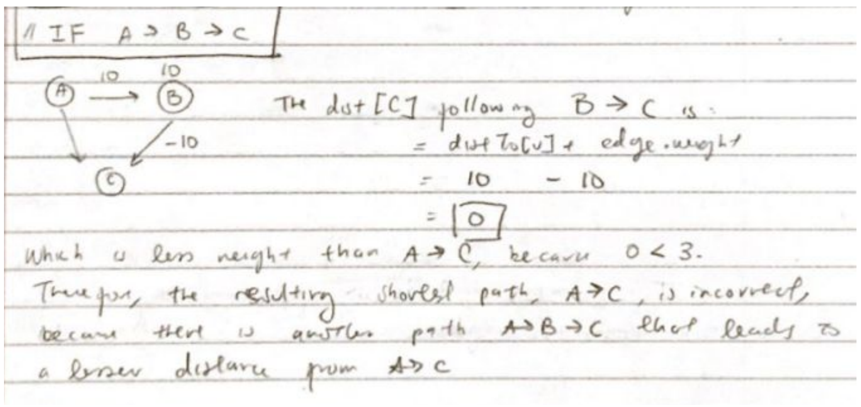
Pictured above is a directed graph where vertex A connects to B and C, and vertex B connects to C. Note that the edge at (B, C) has a negative weight of -10.

To find the shortest path from vertex A to C, first, set vertex A's distance to zero while the other vertices have a distance of infinity. Then, calculate the new distances of vertices B and C with the formula:

distTo[w] = distTo[v] + edge.weight()

Where v is the starting vertex that is trying to reach w: (v, w).

In the diagram above, after calculating the new distances, distance (A, C) is less than (A, B). Therefore, the Dijkstra algorithm will proceed to vertex C to evaluate for the shortest path from (A, C). However, since the destination C has been reached, the algorithm will end. However, this is incorrect, since the path going from (A, B) and (B, C) will have a shorter distance than (A, C).



// IF A → B → C

$$\begin{tikzcd} A \arrow[r, "10"] \arrow[dr] & B \arrow[d, "-10"] \\ & C \end{tikzcd}$$

10

The dist[C] following B → C is:
= distTo[v] + edge.weight
= 10 − 10
= $\boxed{0}$

Which is less weight than A → C, because 0 < 3.
Therefore, the resulting shortest path, A → C, is incorrect, because there is another path A → B → C that leads to a lesser distance from A → C

Since the algorithm will end once it goes through the path of (A, C), it will not evaluate the paths from (A, B) and (B, C). The diagram above shows the resulting distance to C if the paths (A, B) and (B, C) were taken. The final distance of (A, B) and (B, C) is 0, which is less than (A, C)'s distance of 3. Therefore, the Dijkstra algorithm results in an incorrect shortest path.

The diagrams above is an example of the contradiction that fails the Dijkstra algorithm:

distTo[w] > distTo[v] + edge.weight() // Contradiction of the inequality shown above

The final distTo[C] of the diagram, 3, is greater than (distTo[B] + edge.weight) that results to a 0. The Dijkstra algorithm fails because it did not result in the shortest path because there exists a path (that was not calculated) that was shorter than the final answer.

Therefore, the explanation and example prove that Dijkstra's algorithm may produce a path that is not the shortest from vertex v and w if there exist some negative edges.