DESIGN AND ANALYSIS OF ALGORITHMS
Homework 3

Aman Choudhary
MTech Coursework, CSA 2020
Sr No: 17920

October 30, 2020

1 Number of Shortest Paths

1.1 Pseudo Code

$INIT()$

1. $vis[src] = TRUE$
2. $s[src] = 1$
3. **for** each element x in $out[src]$
4. $\quad w = x->node$
5. $\quad$ **if** ($vis[w] == FALSE$)
6. $\quad\quad VISIT(w)$

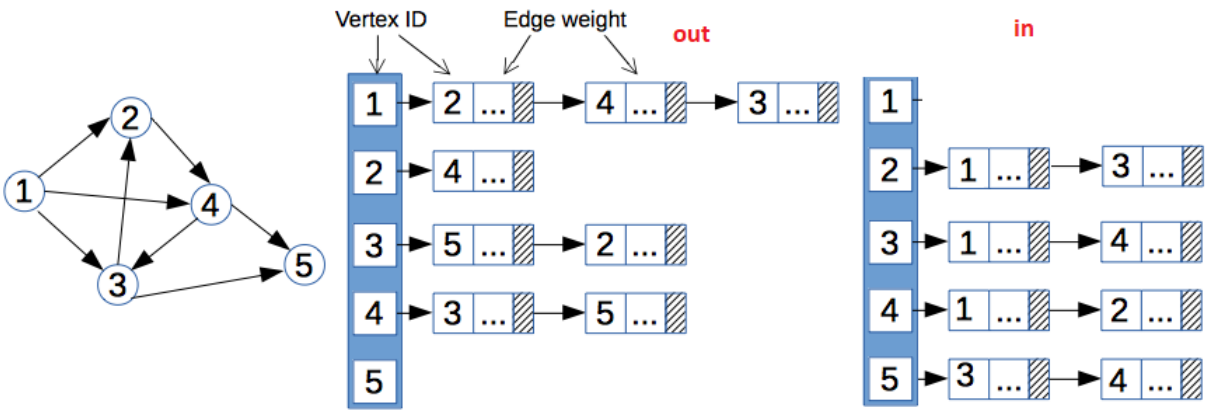
$VISIT(v)$

1. $vis[v] = TRUE$
2. **for** each element x in $in[v]$
3. $\quad u = x->node$ //Incoming edge u $->$ v
4. $\quad$ **if** ($d[u] + x->weight == d[v]$) //Shortest path from src to v contains u.
5. $\quad\quad$ **if** ($vis[u] == FALSE$)
6. $\quad\quad\quad VISIT(u)$
7. $\quad\quad s[v] \mathrel{+}= s[u]$
8. **for** each element x in $out[v]$
9. $\quad w = x->node$ //Outgoing edge v $->$ w
10. $\quad$ **if** ($vis[w] == FALSE$)
11. $\quad\quad VISIT(w)$

1.2 Explanation

out :
It is an adjacency list which stores information in terms of outgoing edges. $out[v]$ is thus, a single list which stores information about all outgoing edges of v.

in :
It is an adjacency list which stores information in terms of incoming edges. $in[v]$ is thus, a single list which stores information about all incoming edges to v.



in and out are just two different representations of $G(V, E)$. in is used to find all incoming edges efficiently. in can be computed in just a single pass through contents of out in $O(V + E)$ time. Computation of in is omitted for keeping the discussion brief.

$vis[1..V]$:
It is an array of V elements. $vis[i]$ denotes whether we have visited a particular node, and calculated the number of shortest paths for that node i or not. All elements of vis are initialized to $FALSE$.

$d[1..V]$:
It stores the length of the shortest path from src to a particular vertex. We get this as output after running Dijkstra's algorithm on our graph. For purposes of brevity, we view Dijkstra's algorithm as a black-box in our case. It takes our graph as input and spits out $d[1..V]$ for us. We thus leave out implementation details of Dijkstra's algorithm.

$s[1..V]$:
It stores the number of shortest path from src to each vertex. All elements of s are initialized to 0.

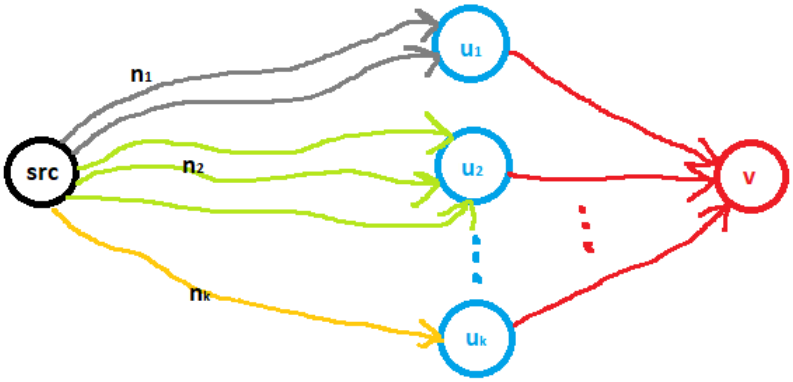
1. Our algorithm is a basic graph traversal. $INIT()$ and $VISIT(v)$ are similar. The difference is that in $INIT()$, we do work related to src only, and in $VISIT(v)$, we do it for other vertices. It is a three step job for each vertex:
   (i) Mark it as visited. (Line 1 in $INIT()$, Line 1 in $VISIT(v)$)
   (ii) Compute the number of shortest paths frm src to that vertex v. (Line 2 in $INIT()$, Line 2-7 in $VISIT(v)$)
   (iii) Explore its unvisited neighbors. (Line 3-6 in $INIT()$, Line 8-11 in $VISIT(v)$)
2. In $INIT()$, first we mark src as visited. Then, we initialize $s[src] = 1$, which conveys, that there is only one shortest path to src, that is the empty path. Then we proceed to visit each of its unexplored neighbours.
3. On entering $VISIT$, we first mark the vertex as visited (Line 1).
4. Then, we check each of its incoming edges. For each such edge (u, v), we verify whether the origin vertex (u) of that edge lies on a shortest path to v (Line 4). If it does, then we check whether we have computed the number of shortest paths to u or not.
5. If we have already calculated the number of shortest paths for the vertex u, we simply add them to our current counter $s[v]$ (Line 7). Otherwise, we first visit u (Line 6). Note that this is a recursive process, and it ensures that all such vertices which lie on the shortest path to v are first visited. In other words, all dependencies for calculation of $s[v]$ are first resolved.
6. After the work for this vertex v is completed, we proceed to visit its unexplored neighbours (Lines 8-11).

1.3 Proof

1. The algorithm produces correct results.

In order to reach vertex v, we must first reach a vertex u such that $(u, v) \in E$. Let there be k incoming edges to v. Also, let the number of shortest paths to each of these vertices u_1 , u_2 , ..., u_k be n_1, n_2, ..., n_k respectively. If upon appending edge (u_1, v) to a shortest path from src to u_1 gives a path of length $d[v]$, then we have a shortest path for v. In this way, for each one of the n_1 shortest paths from src to u_1, we get a corresponding shortest path to v upon appending edge (u_1, v) to it. We can argue similarly for other vertices u_2,...,u_k. Thus we generate shortest paths to v, using shortest paths to vertices from which there are incoming edges to v. Hence,

The total number of shortest paths to v is precisely, the sum of number of shortest paths to all such vertices u such that $(u, v) \in E$ and $d[u] + w(u, v) = d[v]$.



2. All vertices are visited.

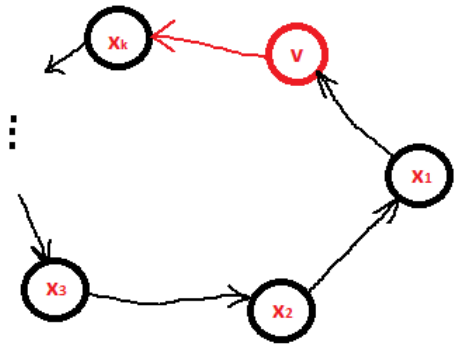
Our algorithm traverses the graph in a similar manner to DFS. The only difference is that in (Lines 2-7), we make sure that all our required values have been computed for calculating $s[v]$. This somewhat changes the order of visiting the vertices. But, nevertheless all the nodes are visited.
Also, each node is visited exactly once, because before every call to $VISIT(v)$ we check, if it has been already visited or not by verifying $vis[v] == FALSE$.

3. The algorithm terminates.

The only case when our algorithm does not terminate is when the sequence of calls to $VISIT()$ falls in a loop. We will show that this can never happen. The only thing we are doing different from DFS is checking for dependencies when calculating shortest paths, which alters the sequence of nodes visited.

Let us suppose for calculating the number of shortest paths to a vertex v, we traverse back via an incoming edge to x_1. Then for calculating the number of shortest paths for x_1, we go back again via an incoming edge to x_2. Then we visit x_3, x_4,...,x_k in similar manner. We keep doing this till we come back at v from x_k.

Now, this is an interesting situation. We came back from where we started. In other words, we are saying that the shortest path to v is of the form ..v, x_k, x_{k-1},..., x_3, x_2, x_1, v. However, this can never be a shortest path to v because it has a **redundant loop of edges all of which have positive weights**. Hence, by contradiction, we have shown that we can never end up visiting vertices in a loop.



1.4 Time Complexity Analysis

We break our algorithm into several steps and do the analysis:

(i) Run Dijkstra's algorithm on our graph to compute the shortest paths $d[1..V]$. We have given run time analysis of Dijkstra's in brief as an appendix. The time taken for this step is $O(V + E) \log V$.

(ii) Creating a different representation in from out takes a single pass through the contents of out. This step takes $O(V + E)$ time.

(iii) Our main algorithm $VISIT$, is a basic graph traversal. We visit each node exactly once and do constant amount of work at each node which includes marking it as visited and updating its $s[v]$. Also since our graph traversal uses adjacency list representation and not an adjacency matrix, the traversal takes $O(V + E)$ time.

Total Time Complexity : $O((V + E) \log V) + O(V + E) = O((V + E)(\log V + 1))$. The constant term 1 is negligible when we consider asymptotic run time analysis. Hence the total complexity is again $O((V+E) \log V)$.

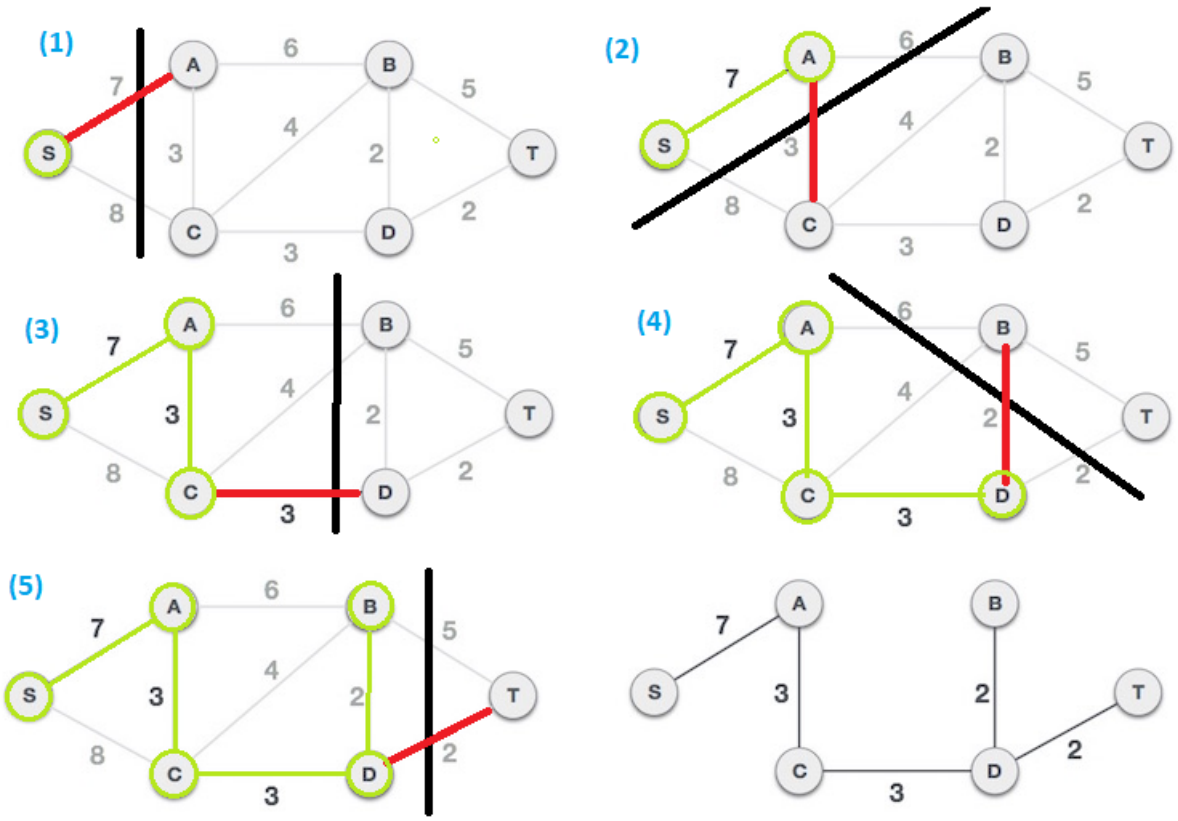
2 Minimum Spanning Tree

We assume that we compute the MST using Prim's Algorithm. In Prim's Algorithm, at any point in time, we have two sets of vertices. Let X denote the set of vertices already included in the partial MST constructed up till now, and the other remaining vertices lie in $V - X$.

At each step of the algorithm, there is a well defined **cut** in the graph, which separates X and $V - X$. The algorithm selects the edge e which has the minimum weight among all the edges present in the cut. This directly follows from the Cut Property.

Cut Property: Consider an edge e of G. Suppose there is a cut (X,$V - X$) such that e is the cheapest edge of G that crosses it. Then e belongs to the MST of G.

The above step is repeated for $V - 1$ times, till our MST has $V - 1$ edges. Then the algorithm terminates.



The above figure illustrates a run of Prim's Algorithm. At each step, the set X is represented by green vertices. The cut between X and $V - X$ is represented by the black line. The cheapest edge in the cut which is selected by Prim's Algorithm is represented in red line.

Note that Prim's algorithm can be thought of as a series of tournaments. Each tournament selects the cheapest edge in the cut at that point in the graph. Every edge in MST is a winner of one such tournament. Thus, a series of $V-1$ such tournaments give us the MST.

In the problem, we have been given a graph G and a minimum spanning tree T. Then we decrease the weight of one of the edges in T. We are required to show that T is still a minimum spanning tree for G.

Let us consider an edge e in T. It was selected by the Prim's Algorithm because at some point in the algorithm, we came across a cut (A,B) where e was the cheapest edge for that cut. That is why, the algorithm selected the edge to be present in MST. That is the edge won that tournament, as there was no better choice other than e.

Now, let us decrease the weight of this edge e. So, when we run Prim's algorithm again, we will again come across that cut (A,B). Since, no other weights have been altered, the cut in which e was present is identical to last time, except for the weight of e. The environment in which e was selected the previous time, it has not changed. This time e would be even more cheaper, and hence a stronger candidate for selection. It will win the tournament again and be selected in MST. Hence we get the same minimum spanning tree T.

Dijkstra's Algorithm

Given a graph and a source vertex in the graph, we want to find the shortest path from the source to all vertices in the graph.

This algorithm is very similar to Prim's algorithm for MST. We generate a shortest path tree with given source as root. We maintain two sets, one set contains vertices included in the shortest path tree, and the other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we pick a vertex which is in the other set (not yet included set) and has a minimum distance from the source. We add this vertex to first set.

Algorithm

1. Create a min heap of size V. Every node contains vertex number and distance from the source.
2. Initialise the heap by setting distance of source as 0, and all other vertices as INF.
3. While min heap is not empty:
   a) Extract vertex 'u' with the minimum distance value.
   b) For every adjacent vertex of 'u', that is still in heap update d[v] if,

$$d[u] + w(u,v) < d[v]$$

Time Complexity = $O((V+E)\log V)$

Build and Initialize heap: $O(V)$

Extract Min: $V * O(\log V) = O(V \log V)$

Decrease Key: $E * O(\log V) = O(E \log V)$

$$O((V+E)\log V)$$