Ankit Murarka amurark 200111393

Introduction to Algorithms – Assignment 4

1.1.1.a Prove that in a breadth-first search of an undirected graph, the following properties hold:

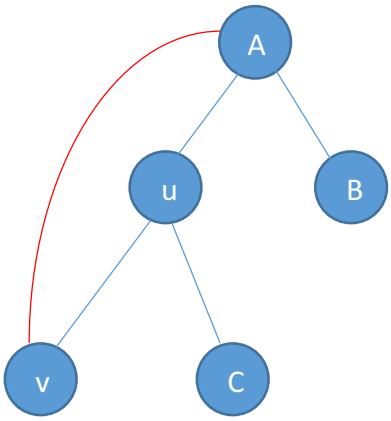
1. There are no back edges and no forward edges

⇨ Lets assume there is an edge (u,v) ∈ E in an undirected graph G(V,E). The edge can be one of the following 4 types of edges in a tree:

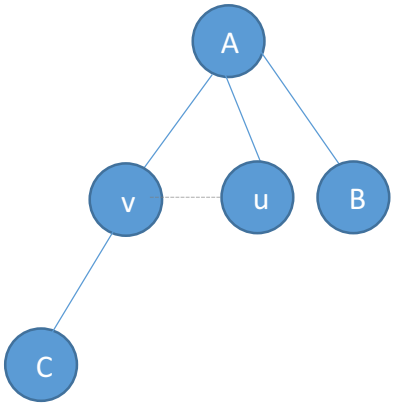
1. Tree Edge: An edge (u,v) is a tree edge if v was discovered first from u.
2. Back Edge: An edge (u,v) is a back edge connecting a vertex u to an ancestor u.
3. Forward Edge: An edge (u,v) is a forward edge which is a non-tree edge and connects a vertex u to descendant v.
4. Cross Edge: An edge (u,v) is a cross edge if these does not share an ancestor descendant relation.

Lets assume that edge (u,v) is a **forward edge** in a BFS search tree of an **undirected graph**. By the definition of a forward edge, we can conclude that u is an ancestor of v. In BFS search, we explore all edges of a vertex u before exploring edges of any other vertex. So, if v is a descendent of u it would have been explored while u was being explored and there would have been a direct **tree edge** between the two of them. Since, we proved that the two were connected by a tree edge, it is not possible to have a forward edge between the two.

Now lets assume that edge (u,v) is a **back edge** in a BFS search tree of an **undirected graph**. By the definition of a back edge, we can conclude that u is a descendent of v. Again if u and v are binded by an ancestor-descendant relationship, we can infer that there will be a tree edge between u and v because whenever the ancestor, v was being explored by BFS search, u would also have been discovered. So there exists a tree edge between them and hence the 2 cannot be connected by a back-edge.



A. Tree denoting a forward edge



B. Same tree with no forward edge

In the graph above, the red-line denotes a forward edge(or a back edge, since it's an undirected graph, we can assume either ways). Now, according to BFS search tree, on exploring A, the children of A are immediately explored. Looking at the graph, we can say that A has 3 children, u, v and B. Since v is a children of A, there will be a tree edge between the two. And since there is a tree edge between the 2, we can safely conclude that there can't be a forward or a back edge between them.

2. For each tree edge (u,v),we have v.d = u.d + 1.

⇨ In general terms, for search algorithms in Graphs, u.d is used to represent the distance of a vertex, u, from the source. Similarly, v.d is used to represent the distance of a vertex, v, from the source. For an edge connecting the vertices u and v, the property:

$$v.d = u.d + 1$$

hold true because of the way Breadth First Search traverses. We already know and have proved that Breadth First Search for undirected graphs **have only forward edges(no back edges and no cross edges)**. For each vertex(starting from the source) that BFS encounters, **BFS explores all its child vertices before moving on to the next vertex.** Hence, assuming that the distance of a parent(ancestor) vertex 'u' from the source is u.d, the distance of the child(descendent) vertex 'v' from the source will be v.d. Since BFS explores the child vertex after the parent vertex, we can safely say that v.d = u.d + 1.

3. For each cross edge (u,v), we have v.d = u.d or v.d = u.d + 1.

⇨ When we talk about cross edges in BFS, it is understood that the vertices sharing a cross edges does not share a ancestor-descendant relation.

So we can infer that vertices will be siblings because:
1. These vertices cannot be ancestor descendant.
2. Since there are not multiple trees in a BFS forest. We can say that they are part of the same tree.
3. Also, they can't be at different levels because then one would have been explored because of the other.
4. And, they can't have different parents because then one on the left would have explored the one on the right.

Which leaves us with just one possibility, that the 2 of them are siblings. Since the 2 are siblings, if the one is being explored(say **u**), the other would be there in the queue already. And according to **Lemma 22.3**, the vertices in a queue at any given time share the relation $v_i.d <= v_{i+1}.d$ for i = 1, 2, 3, …. r-1. And according to **Corollary 22.4**, if 2 vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d <= v_j.d$ at the time that v_j is enqueued.

1.1.1.b Prove that in a breadth-first search of a directed graph, the following properties hold:

1. There are no forward edges.

⇨ Lets assume that edge (u,v) is a **forward edge** in a BFS search tree of a **directed graph**. By the definition of a forward edge, we can conclude that u is an ancestor of v, assuming that the edge is directed from u to v. In BFS search, we explore all edges of a vertex u before exploring edges of any other vertex. So, if v is a descendent of u it would have been explored while u was being explored and there would have been a direct **tree edge** between the two of them. Since, we proved that the two were connected by a tree edge, it is not possible to have a forward edge between the two. Basically, the reason behind no forward edges in a directed tree and an undirected tree is the same.

2. For each tree edge (u, v), we have v.d = u.d + 1.

⇨ In general terms, for search algorithms in Graphs, u.d is used to represent the distance of a vertex, u, from the source. Similarly, v.d is used to represent the distance of a vertex, v, from the source. For an edge connecting the vertices u and v, the property:

$$v.d = u.d + 1$$

hold true because of the way Breadth First Search traverses. For each vertex(starting from the source) that BFS encounters, **BFS explores all its child vertices before moving on to the next vertex.** Hence, assuming that the distance of a parent(ancestor) vertex 'u' from the source is u.d, the distance of the child(descendent) vertex 'v' from the source will be v.d. Since BFS explores the child vertex after the parent vertex and tree edges in a BFS can only be between parents and children, we can safely say that **v.d = u.d + 1**.

3. For each cross edge (u, v), we have v.d <= u.d + 1.

⇨ Since, in BFS, we have already proved that cross edges between sibling nodes only, we can safely say that while one of the vertices(u or v) is being enqueued and explored the other edge will also be present in the queue. Since the 2 vertices will be in the same queue at the same time, according to **Lemma 22.3** and **Corollary 22.4**,

v.d <= u.d + 1

4. For each back edge (u, v), we have 0 <= v.d <= u.d.

⇨ Since the distance of any vertex v cannot be negative, we can clearly conclude that v.d is greater than or equal to 0. Now in this case the back edge (u, v) will be from u ~> v, so according to the definition of a back edge, u will be a descendant of v. So u.d > v.d. Therefore we can say that:

0 <= v.d <= u.d

1.2 An Euler tour of a strongly connected, directed graph G = (V, E) is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

a. Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex v ∈ V.

2. Purpose: Reinforce your understanding of minimum spanning trees and algorithm design (6 points). Please sole problem 23-4 on page 641. Give a convincing argument for the time bound of the implementation of each of the algorithms.

a. MAYBE-MST-A(G,w)

```
1. sort the edges into nonincreasing order of edge weights w.
2. T = E
3. for each edge e, taken in nonincreasing order by weight
4.   if T – {e} is a connected graph
5.    T = T – {e}
6. return T
```

- ⇨ The algorithm stated in this part **results in a minimum spanning tree**. This was concluded after considering the following points:
  1. The edges are being removed from the tree set **T** in a non-decreasing order if they satisy certain condition. This will result in a tree with minimum edge weight.
  2. The algorithm has a condition that makes sure that all the vertices are always connected and T remains a connected graph. This satisfies the condition for an MST.
  3. An MST algorithm can be greedy, and this algorithm is greedy as well. At each and every step, this algorithm is going for the best option out of a list of choices. That is to get rid of edges with higher weights.
  4. The definition of Minimum Spanning Trees states that the tree should have the minimum edge weight possible while still satisfying the conditions of a tree. In this algorithm, by removing the higher weight edges, this condition is satisfied.
  5. There should not be any loops in an MST. This algorithm also makes sure that if there is any loop with a high weight edge, it will remove that edge from the tree.

This algorithm follows a loop invariant that the edges in set T will always connect all the vertices in the tree. The loop invariant is used as follows:
Initialization: In line 1, T is initialized with E, that is all the edges in the tree., hence set T trivially satisfies the loop invariant.
Maintenance: The condition on line 4 ensures that the edges in the set, T are sufficient for a connected graph.
Termination: When the for loop terminates, the set, T will still have sufficient edges for a connected graph, last line returns this set T. Hence the set T returned in the last line must be a Minimum Spanning Tree.

A more efficient implementation would be:
For sorting, we use Merge-Sort which is O(E.logE) where E is the number of edges.
For Breadth First Search to check connectivity for each edge: O(V+E) * total number of edges E, which gives O(E(V+E)).

So, the **running time complexity** of the algorithm **MAYBE-MST-A** comes out to be O(ElogE + E(V+E)).

b. MAYBE-MST-B(G,w)

```
1. T = Ø
2. for each edge e, taken in arbitrary order
3.   if T U {e} has no cycles
4.    T = T U {e}
5. return T
```

- ⇨ The algorithm stated in this part **won't necessarily result in a minimum spanning tree**. This was concluded after considering the following points:

1. The edges are being added to the tree set T in an arbitrary fashion. This might result on a connected spanning tree, but it wont necessarily be a tree with minimum weighted edges.
2. Although the algorithm has a condition that there are no cycles while adding new edges, there is no condition to check if the edges being added are minimum weight edges.

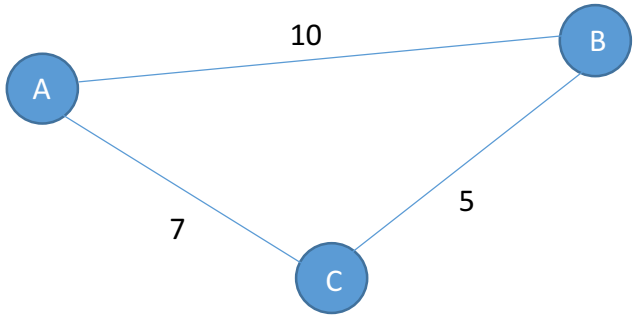
For ruuning time complexity:
We have V MAKE-SET operations + E FIND-SET Operations + V-1 Unions.
So, the **running time** of the algorithm **MAYBE-MST-B** comes out to be O(VlgV + E).

c. MAYBE-MST-C(G,w)

```
1. T = Ø
2. for each edge e, taken in arbitrary order
3.   T = T U {e}
4.   if T has a cycle c
5.     let e' be a maximum-weight edge on c
6.     T = T – {e'}
7. return T
```

⇨ The algorithm stated in this part **results in a minimum spanning tree**. This was concluded after considering the following points:

1. Although the edges are being added to the tree set T in an arbitrary fashion, each time it is ensured that if the added edge creates a loop(cycle), the maximum weight edge from the cycle is removed. Lets look at this cycle which is assumed to be a part of a graph. In this case, vertices A, B and C are connected by 3 edges weighing 10, 7 and 5. So according to the algorithm **e' will be equal to 10**. Hence even if the edge (A, B) weighing 10 is removed, the 3 vertices A,B and C are still connected, hence satisfying the conditions for a spanning tree or a connected graph.



2. The algorithm has a condition which makes sure that there are no cycles in the tree. Hence satisfying a condition for an MST.
3. Since the algorithm is ensuring that whenever there is a possibility that a maximum weighted edge can be removed while ensuring that all the vertices are connected, it removes the maximum edge hence maintaining the conditions of an MST.

A better implementation with time complexity would be:
We first need to create an adjacency list for T, we require:

1. To add edges to T which will take O(V)
2. To check for any existing cycles which will require O(V).
3. And if there are any cycles remove them which will require O(V).
4. And doing this for E edges will result in a running time of O(VE).

The **running time** of the algorithm **MAYBE-MST-C** comes out to be O(VE).

3. Purpose: Reinforce your understanding of Dijkstra's shortest path algorithm, learn about multiple solutions, and practice algorithm design (4 points). In the usual formulation of Dijkstra's algorithm, the number of edges in the shortest (=lightest) path is not a consideration. Here, we assume that there might be multiple shortest paths. Design an algorithm that takes as input a graph G=(V,E), directed or undirected a nonnegative cost function on E and vertices s and t; your algorithm should output a path with the fewest edges amongst all shortest paths from s to t.

⇨ The modified Dijksta's algorithm is as follows:

```
DIJKSTRA'S(G,W,S)
  INITIALIZE-SINGLE-SOURCE(G,s)
  S = ϕ
  Q = G.V
  while Q ≠ ϕ
    u = EXTRACT-MIN(Q)
    S = S U {u}
    for each vertex v ∈ G.adj(u)
# The if statement makes the algorithm faster and more efficient by making sure that the vertices are not already a part of the shortest path before relaxing them.
      if v ∉ S then
        RELAX(u,v,w)


RELAX(u,v,w)
  if v.d > u.d + w(u,v)
    v.d = u.d + w(u, v)
    v.Π = u
```

```
# If there are two paths from the source to the vertex 'v' which are equal in length
then check if the current path is having a lower edge count, if it is having a lower edge
count then assign it as the shortest path to that vertex.
  else if v.d = u.d + w(u, v)
    if v.l > u.l + 1
      v.d = u.d + w(u, v)
      v.Π = u
      v.l = u.l + 1


INITIALIZE-SINGLE-SOURCE(G, s)
  for each vertex v ∈ G.V
    v.d = ∞
    v.Π = NIL
# v.l indicates the edge count in the shortest path to v.
    v.l = ∞
  s.d = 0
# Initialize the edge count in the shortest path to source vertex to 0.
  s.l = 0
```

DESCRIPTION OF THE ALGORITHM

The algorithm works on a non-negative weight, directed graph and results in a shortest path from a single source to all other vertices.

According to Dijkstra's Algorithm the Set of edges(**S**) which conclude in the shortest path is initialized to NULL SET and the source is initialized. The distance, d indicated the weight of the path leading to that vertex. For the source, the label '**d**' is initialized to **0** while for all other vertices **v**, it is initialized to **∞**. The label pi(**Π**) is used to indicate predecessor vertex. It is initialized to **nil** for all the vertices.

A priority queue **Q** is initialized with all the vertices in the graph.

A **while** loop which terminates when the queue is empty is initiated. Each time, the vertex(u) with the minimum value of 'd' is extracted from the queue and all its neighbouring vertices(v)(from the adjacency list) are explored and RELAXED.

THE RELAX FUNCTION

The relax function checks if the sum of the weight of the path to the vertex u and the weight of the edge between u and v is less than the current weight of the path to the vertex v. If this condition is true, it means that a new shortest path is explored from the source to the vertex v.

To ensure that the shortest path to each vertex from the source has the fewest edges, if more than one shortest path is explored for the same vertex, then the edge count comparison is done and the path with the fewer number of edges is chosen.

PROOF OF CORRECTNESS

We use the following loop invariant:

At the start of each iteration of **while** loop, for each vertex v.d = ∂(s, v) for each vertex v ∈ S. In other words, the distance calculated between the source to a any vertex v will be the shortest distance before each iteration of for loop.

INITIALIZATION

Initially S = ϕ, this means that the loop invariant is trivially true.

MAINTENANCE

To prove that u.d = ∂(s, u) for each vertex, 'u' added to the set 'S' in.
By contradiction, we can say that u.d not equal to ∂(s, u). Lets say that y appears before u on a shortest path from s to u and all edge weights are non-negative, we have ∂(s, y) <= ∂(s, u) and thus, we have

y.d = ∂(s, u)
<= ∂(s, u)
<= u.d

But because u and y were in V – S, we have u.d <= y.d. Thus we see the contradiction fails and hence:

y.d = ∂(s, y) = ∂(s, u) = u.d

TERMINATION

At termination, Q = ϕ, so it the invariant hold true trivially.

TIME BOUND OF THE ALGORITHM

To maintain the minimum priority queue being used in dijkstra's algorithm. There are 3 basic priority queue operations:- INSERT(when we are initializing the Queue) , EXTRACT-MIN and DECREASE-KEY(when we are changing the d value of a vertex in RELAX function). The algorithm calls the INSERT and EXTRACT-MIN function V times while the DECREASE-KEY function is called for the adj list of each vertex. Since the total number of elements in an adjacency list for a **directed graph** equals E(no. of edges). The maximum number of DECREASE-KEY operations equals E.
Using a Binary Heap for minimum priority queue, the total time to build the binary heap is O(V) and each DECREASE-KEY operation takes O(lg V) time. For E such operations the total time

taken would be O((V+E)lg V). The operation to select the shortest path with fewer edges, does not effect the time complexity of the algorithm.

---

4. Purpose: Reinforce your understanding of Dijkstra's shortest path algorithm, and practice algorithm design (6 points). Suppose you have a weighted, undirected graph G with positive edge weights and a start vertex s. Describe a modification of Dijkstra's algorithm that runs (asymptotically) as fast as the original algorithm, and assigns a label usp[u] to every vertex u in G, so that usp[u] is true if and only if there is a unique shortest path from s to u. By definition usp[s] is true. In addition to your modification, be sure to provide arguments for both the correctness and time bound of your algorithm.

⇨ The modified Dijkstra's algorithm is as follows:

```
DIJKSTRA'S(G,W,S)
  INITIALIZE-SINGLE-SOURCE(G,s)
  S = ϕ
  Q = G.V
  while Q ≠ ϕ
    u = EXTRACT-MIN(Q)
    S = S U {u}
    for each vertex v ∈ G.adj(u)
      if v ∉ S then
        RELAX(u,v,w)


RELAX(u,v,w)
  if v.d > u.d + w(u,v)
    v.d = u.d + w(u, v)
    v.Π = u
# If there are 2 paths which are equal in weight originating from the source to the vertex 'v', then set the flag 'usp' for that vertex to FALSE.
  else if v.d = u.d + w(u, v)
    v.usp = false


INITIALIZE-SINGLE-SOURCE(G, s)
  for each vertex v ∈ G.V
    v.d = ∞
```

```
    v. Π = NIL
# Initialize a flag for unique shortest path to each vertex in the graph. It is TRUE if there exists a UNIQUE SHORTEST PATH to that vertex, else its false.
    v.usp = true
  s.d = 0
  v.usp = true
```

DESCRIPTION OF THE ALGORITHM

The algorithm works on a non-negative weight, directed graph and results in a shortest path from a single source to all other vertices.

According to Dijkstra's Algorithm the Set of edges(**S**) which conclude in the shortest path is initialized to NULL SET and the source is initialized. The distance, d indicated the weight of the path leading to that vertex. For the source, the label '**d**' is initialized to **0** while for all other vertices **v**, it is initialized to **∞**. The label pi(**Π**) is used to indicate predecessor vertex. It is initialized to **nil** for all the vertices.

A priority queue **Q** is initialized with all the vertices in the graph.

A **while** loop which terminates when the queue is empty is initiated. Each time, the vertex(u) with the minimum value of 'd' is extracted from the queue and all its neighbouring vertices(v)(from the adjacency list) are explored and RELAXED.

THE RELAX FUNCTION

The relax function checks if the sum of the weight of the path to the vertex u and the weight of the edge between u and v is less than the current weight of the path to the vertex v. If this condition is true, it means that a new shortest path is explored from the source to the vertex v.

To ensure that a vertex has a unique shortest path from the source. Each vertex other than the source are assigned a flag 'usp' which is true if and only if there is a UNIQUE SHORTEST PATH from the source to that vertex. If there are more than 1 shortest paths from the source to that vertex, the flag is set to false.

PROOF OF CORRECTNESS

We use the following loop invariant:

At the start of each iteration of **while** loop, for each vertex v.d = ∂(s, v) for each vertex v ∈ S. In other words, the distance calculated between the source to a any vertex v will be the shortest distance before each iteration of for loop.

INITIALIZATION

Initially S = ϕ, this means that the loop invariant is trivially true.

MAINTENANCE

To prove that u.d = ∂(s, u) for each vertex, 'u' added to the set 'S' in.
By contradiction, we can say that u.d not equal to ∂(s, u). Lets say that y appears before u on a shortest path from s to u and all edge weights are non-negative, we have ∂(s, y) <= ∂(s, u) and thus, we have

y.d = ∂(s, u)
<= ∂(s, u)
<= u.d

But because u and y were in V – S, we have u.d <= y.d. Thus we see the contradiction fails and hence:

y.d = ∂(s, y) = ∂(s, u) = u.d

TERMINATION

At termination, Q = ϕ, so it the invariant hold true trivially.

TIME BOUND OF THE ALGORITHM

To maintain the minimum priority queue being used in dijkstra's algorithm. There are 3 basic priority queue operations:- INSERT(when we are initializing the Queue) , EXTRACT-MIN and DECREASE-KEY(when we are changing the d value of a vertex in RELAX function). The algorithm calls the INSERT and EXTRACT-MIN function V times while the DECREASE-KEY function is called for the adj list of each vertex. Since the total number of elements in an adjacency list for a **directed graph** equals E(no. of edges). The maximum number of DECREASE-KEY operations equals E.
Using a Binary Heap for minimum priority queue, the total time to build the binary heap is O(V) and each DECREASE-KEY operation takes O(lg V) time. For E such operations the total time taken would be O((V+E)lg V). The operation to change the USP flag takes a constant time per cycle, so it doesn't effect the time complexity of the algorithm.