Unity id : amanur
Name - Anusha Gururaja Manur

# HOMEWORK - 4

1. **Purpose: Learn about articulation points, bridges, biconnected components, and Euler tours.**

   **Solution:**

**a. Prove that the root of $G_\pi$ is an articulation point of G if and only if it has at least two children in $G_\pi$ .**

Let us assume that the root of $G_\pi$ be an articulation point.
Case 1: No children

If the root has no children, then the root is not connected to any other vertex. If the root is not connected to any vertex, it cannot be an articulation point. This violates the assumption

Case 2: 1 child

$G_\pi$ is a depth search tree. If it has a single child, the other elements can be reached through that vertex. Therefore, removing that vertex will not disconnect the graph. This again violates the assumption that the root is an articulation point.

Therefore, if the root of $G_\pi$ has to be an articulation point, it should have at least 2 children

Let us consider the converse, if the root of $G_\pi$ has 2 children, then it is an articulation point.

Since the root has 2 children, then it has a left and right subtree. Since $G_\pi$ is a depth first tree of an undirected graph, we know that there are no edges between nodes of left subtree and nodes of right subtree. This means that there is a unique path between nodes of the subtrees which is through the root of $G_\pi$. Therefore, if we remove the root, there will be disconnected components. Therefore, if root of $G_\pi$ has at least two children, then it is an articulation point.

**b. Let v be a non root vertex of G. Prove that is an articulation point of G if and only if has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.**

Let us assume that v is an articulation point and a non root vertex. If we remove v, then it will lead to disconnected components. All ancestors of v will be in different components whereas its descendents will be in different components. If a backedge existed from v or any descendent of v, this would not lead to multiple components. This violates the assumption that v is an articulation point. Therefore, if v a non-root vertex of $G_\pi$ is an articulation point then it cannot contain children or descendants having back edge.

Let us consider the converse. If there do not exist backedges from v nor its descendents to ancestor of v, then v is an articulation point. If there exists no backedge from v or its descendents, this implies that there is a unique path between v, its ancestors and descendents. If v is removed, then there will be disconnected components. This implies that v is an articulation point.

**C. Show how to compute v.low for all vertices in O(E) time.**
To compute v.low, we use the following equation -
low[v] = min( d[v], min ( low[u] ), min $_{(v,w)}$ (d[w]) ),
Where,
u - descendent of v
(v,w) - backedge from v to v’s ancestor - (v,w)

We compute low in 2 stages -

1. We consider the leaves first and compute their low
2. We use the values of low of the leaves and compute the values of the non-leaf nodes.

We initially start from the leaves and compute low as we move towards the root. The leaves have no roots, and hence we consider only d[v] or d[w], if v has back edges.
$low_{leaf} = min( d[v] , min_{(v,w)} (d[w]))$

Then, we move up the tree to non-leaf nodes. To compute low[v] for these nodes, we assume that low has been computed correctly. If low[v]= d[w], this indicates that there is backedge (v,w) from v or (u,w) from u, which is a descendent of v. If u is a descendent of v, then we know that d[u] > d[v] as u is visited after v and hence the discovery time of u is greater than that of v. Therefore, if d[w] < d[v], we also have d[w] < d[u], implying that low[u] = d[w].

Running time : To compute low[v], we look at,

- Discovery time of v
- Discovery time of ancestor w of v, over all backedges from v to w.
- Discovery time of ancestor w from u, where u is a descendent of v.

The total time needed for the above computation depends on traversal of edges and is hence proportional to O(E) and we compute this for each node V. Since in a connected graph, V=O(E), we get the total bound as O(E).

**d. Show how to compute all articulation points in O(E) time.**

To find the articulation points, we run depth first search algorithm and the algorithm described in part c. In (c), we have shown that the running time is O(E).
According to part (a), we can check if the root is an articulation point in O(1).
Now consider part (b) which says that if v is a non-root vertex Gπ, the v is an articulation point of G iff v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v. Since s and other descendents of v are discovered after v,  d[v] < low[s]. This indicates that there is no backedge from s to a proper ancestor of v, because if there existed such a backedge, this condition would not hold. So, if low[s] > d[v], then s has no back edge to a proper ancestor of v and thus v is an articulation point.

Running time : The time taken is calculated by considering the number of children of each non root vertex v. This will be O(V).
Since in the connected graph V=O(E), all parts (DFS, computing low and testing for backedge can be done in O(E) time.

**e. Prove that an edge of G is a bridge if and only if it does not lie on any simple cycle of G.**

Let <u,v> be a bridge. If <u,v> is removed from G, this would lead to disconnected components C1 which has vertex ‘u’ and C2 which has vertex ‘v’.
Let us assume that <u,v> is part of a simple cycle. This implies that there exists a set of vertices $v_!, v_2, ..., v_n$ such that a path $u, v, v_1, v_2, ..., v_n$ exist. In case (u,v) is removed, there still exists a path from u to v through $u, v_n, ..., v_2, v_1, v$. This indicates that we can still reach v from u even if the edge <u,v> is removed. Hence, they are not disconnected and this implies that <u,v> is not a bridge if it lies on a simple cycle.

Let us now consider the converse. Let us assume that <u,v> is not part of a simple cycle nor is it a bridge. This implies that since <u,v> is not a bridge, there should still be a path connecting u and v, such that $u, v_n, ..., v_1, v$. This also indicates that since such a path exists between u and v, even on removal of edge <u,v>, it must lie on a simple cycle $u, v, v_1, v_2, ..., v_n$.This contradicts our assumption that <u,v> does not lie on a simple cycle. Hence, (u,v) must be a bridge.

**B. Euler tour**

Presence of an euler tour means that all edges of G can be covered by visiting each edge just once. For every vertex v, to choose one outgoing edge, the euler tour must go through v. Therefore, it visits v as many times as the number of outgoing edges from v, which is the outdegree. Since each edge can be visited only once, to enter a vertex, the euler tour must have visited each of the incoming edges. Therefore, since we enter through a different edge each time, the number of times we visit v will be equal to the incoming edges of v. Therefore for an euler tour, outdegree of v = indegree of v.

If the number of incoming edges is greater than the number of outgoing edges, then there cannot be a unique path out for each path in. Similarly, if there are more outgoing edges than incoming edges, then there cannot be a unique path in, for each path out.

**2. *Purpose: Reinforce your understanding of minimum spanning trees* (8 points). Please solve problem 23-4 on page 641. You do NOT have to describe the most efficient implementation of each algorithm.**

a) The spanning tree T generated is a minimum spanning tree.

   **Proof of correctness -**
   We prove this by induction. We need to show that on removing e from T on line 6, T still contains a MST.

   We remove e in such a way that it does not disconnect T. Since removing e does not disconnect T, we can say that e was part of a cycle and hence its removal did not disconnect T. This implies that there is a T'⊆ T that is an MST of G.

   **Case 1:** e is not part of T'.
   T' does not contain e and is a MST.

   **Case 2**: e is a part of T'

   We know that every edge of a tree is a cut edge. If e is still part of T', then removing e will create 2 trees $T_a$ *and* $T_b$ . But to make sure it's still connected, we need another edge e' that is part of the same cycle as e and hence removal of one of these vertices will still ensure that T' is still connected. If weight(e) >= weight(e'), then e is removed and not e', and T' remains connected. So $T'' = T_a \cup T_b \cup e'$ cannot be smaller than T' as T' is an MST. But e' <= e so weight of T' equals weight of T''. Therefore T'' is also an MST even when e is removed.

b) MST is not generated.
   Let us take an example.
   Example:

   Consider undirected edges:
   (ab, 5)
   (bc, 10)
   (ca, 18)

Since on line 2, the edges are picked arbitrarily, the edges are picked in any order and this does not always lead to a MST. For example, the algorithm might end up picking ab, ca or bc, ca as the edges of the minimum spanning tree which is incorrect.

c) The spanning tree T generated is a minimum spanning tree.

Let E be the set of edges considered uptil now by the algorithm. Initially, we have E=Null and add edges to E = E ∪ {e}, as we progress.

**Loop Invariant:**
Every connected component of T is an MST of the corresponding connected component of E. In other words, T is a minimum spanning forest of E.

**Initialisation:**
E = Φ, T also equals Φ. Loop invariant holds trivially.

**Maintenance:**
Considering the loop invariant, E has the edges we've considered till now and T will be the minimum spanning forest of E.

Case 1: $T = T \cup e$ does not form a cycle
If the addition of e to T does not form a cycle, this means that e connects 2 components of T. This implies that T = T ∪ {e} must be an MST of E = E ∪ {e}

Case 2: $T = T \cup e$ forms a cycle
Let's remove e' , which is the maximum weight edge, from T which is part of cycle formed by T ∪ {e}. Removal of this leads to 2 components $T_a\ and\ T_b$. To make sure it remains connected, we add e which reconnects $T_a\ and\ T_b$, which was disconnected on removal of e'. T before removing e' was a minimum spanning forest of E. Since e' has maximum weight, e<e', new $T = T_a \cup T_b \cup e$ and E gets updated to E = E ∪ {e} as the cost of this forest is lesser than the one containing e'. At each point T is the minimum spanning forest of the set of edges E considered then.

**Termination:**
At the end of all iterations, we will have E which will contain all the edges of G and T will have only edges which have minimum weight and also not contain cycles. Hence T would be the minimum spanning tree of G.

**3.** ***Purpose: Reinforce your understanding of Dijkstra's shortest path algorithm, and practice algorithm design (16 points).*** **Suppose you have a weighted, undirected graph *G* with positive edge weights and a start vertex *s*.**

**a) (6 points) Describe a modification of Dijkstra's algorithm that runs (asymptotically) as fast as the original algorithm, and assigns a binary value usp[*u*] to every vertex *u* in *G*, so that usp[*u*]=1 if and only if there is a unique shortest path from *s* to *u*. We set usp[*s*]=1. In addition to your modification, be sure to provide arguments for both correctness and time bound of your algorithm, and an example.**

Let us modify the RELAX (u, v) in the regular Dijkstra's algorithm to check if there exists a unique shortest path from s to u for every vertex u in G.

There is a possibility of 3 cases in the relax function.

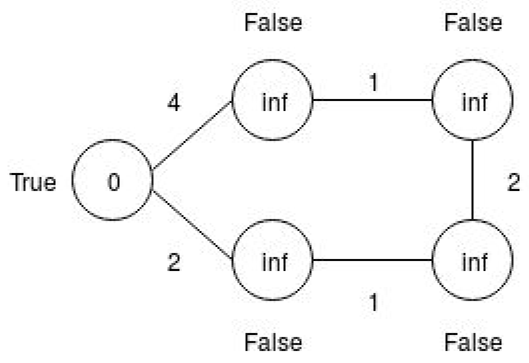
Case 1: d[u] + weight (u, v) < d[v]
A shorter path was discovered. This path can be considered unique only if the shortest path to u is also unique.

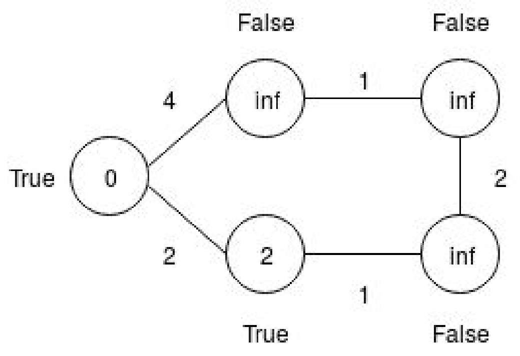
Case 2: d[u] + weight (u, v) = = d[v]
Another path to v was discovered through u, but is the same length as the current path. Since we have 2 such paths, it can't be unique.

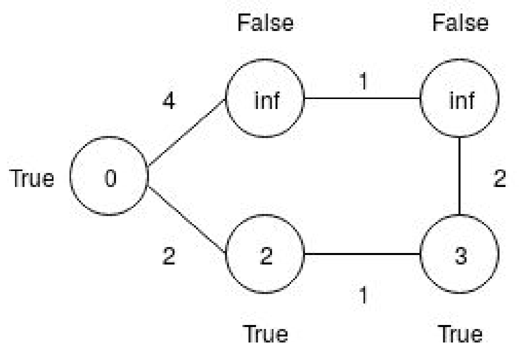
Case 3: d[u] + weight (u, v) > d[v]
Since we did not find a shorter path, this does not have an affect on the uniqueness of path from source to v.

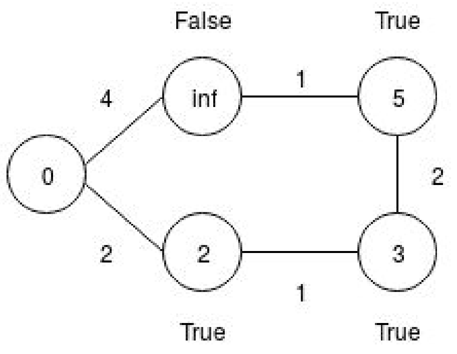
```
PseudoCode:
usp[u] = False //Initialize for all nodes in graph G
usp[s] = True // only source has a usp of true and is initialized likewise.

RELAX (u, v) :
        if d[u] + weight (u, v) < d[v] :
                d[v] = d[u] + weight (u, v)
                usp[v] = usp[u]
        else if d[u] + weight (u, v) = = d[v] :
                usp[v] = False
```

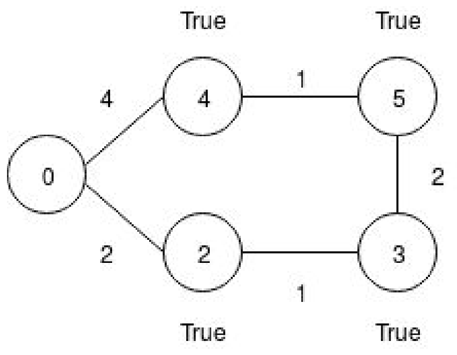
**Example:**

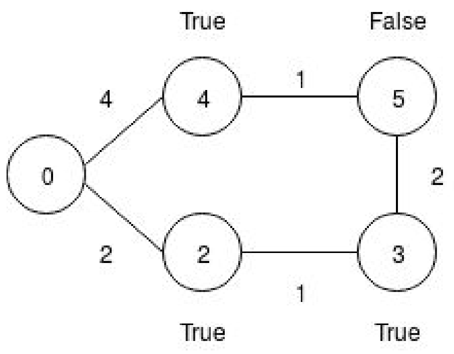

False
False
4
inf
1
inf
True
0
2
2
inf
inf
1
False
False


False
False
4
inf
1
inf
True
0
2
2
2
inf
1
True
False


False
False
4
inf
1
inf
True
0
2
2
2
3
1
True
True


False
True
4
inf
1
5
0
2
2
2
3
1
True
True


True
True
4
4
1
5
0
2
2
2
3
1
True
True


True
False
4
4
1
5
0
2
2
2
3
1
True
True

**Proof of correctness :**

Usp[u] will be set to true only if there is a shortest path from source 0, to u anytime during execution. Let V is the set of vertices to which we have already found the shortest path. u is added to V, only when  d[u] = δ(s, u). usp[u] will be true only once i.e, when there is a shortest path from u to s. When there are multiple shortest paths, then when the duplicate shortest path is found, our algorithm will set usp[u] to false.

**Time Complexity:**
We have made modifications to the Relax function with some assignment operations which are constant time operations which will not increase the time complexity of the algorithm and does not affect the overall time complexity of  Dijkstra's algorithm.
Therefore, using binary heaps, we get the overall time complexity as ((V+E)logV)and using Fibonacci heaps, we get (VlogV+E) as the overall time complexity.

**b) (10 points) Implement the algorithm described in a). As you can see in the code framework provided (DijkstraFramework), Dijkstra.py/java file has a method: Dijkstra_alg. The method has an input parameter list (n, e, mat, s), where n = number of vertices of G, e = number of *undirected* edges of G, mat = an e x 3 matrix that defines the edges of G, and s = the source vertex of Dijkstra's algorithm. Assume the vertices of G are numbered 1…n. In mat each row consists of three integers <u, v, weight>. Here, u and v define the edge (u,v), and weight is the corresponding edge weight. The method returns an n x 2 matrix, where the i*th* row contains the path length and the usp value for the shortest path between source and vertex *i* for *i* {1, …n}. More detailed examples (that refers to the first two test cases) are given below. You can create additional methods if required but do not change the name of existing methods and any existing code - points will be cut if you do. To avoid loss of marks, you should use basic arrays to implement the priority queue in Dijkstra's algorithm, and not use any packages. You can code this in either Java or Python. Another file DijkstraTest.py/java has been provided which tests your code for various inputs. You need to test your code on VCL using this file (instructions for VCL setup and test are can be referred from readme file provided in last homework). Please only submit the file Dijkstra.py/java and not the test file. Your code will be checked on VCL by us automatically using the provided cases, plus some (unknown) inputs. To avoid loss of marks please make sure that all provided test cases pass on VCL by using the test file.**