**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.

1. 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.

1. 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.

1. 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 vi.d <= vi+1.d for i = 1, 2, 3, …. r-1. And according to **Corollary 22.4**, if 2 vertices vi and vj are enqueued during the execution of BFS, and that vi is enqueued before vj. Then vi.d <= vj.d at the time that vj 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.

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

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)

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

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 = ∞ #this is to indicate levels

s.d = 0

s.l = 0

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

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

v.usp = true

s.d = 0

Comment the program and explain.

//Its an undirected graph, so a vertex can be encountered more than once.

//The condition if v not-belongs to S actually makes it efficient and faster

//If more than one path to the same vertex from the source are same in weight, then its not a UNIQUE shortest path