MakeGraph

```
#Make an Adjacency List
def makeGraph(nodes,edges):
    G = {}

    for u in nodes:
        G[u] = []

    for x,y,w in edges:
        G[x].append((y,w))

    return G
```

Priority Queue For Shortest Path Algorithm

```
def enqueue(Q,label,p):
    for i in range(len(Q)):
        if Q[i][0] == label:
            del Q[i]
            break
    for i in range(len(Q)):
        if Q[i][1] > p:
            Q.insert(i,(label,p))
            return
    Q.append((label,p))
    return

def dequeue(Q):
    x = Q[0]
    del Q[0]
    return x[0]
```

Breadth-First Traversal

Let G = (V, E) is a graph which is represented by an adjacency matrix Adj. Assume that nodes in a graph record visited/unvisited information.

procedure BREADTH-FIRST (G)
1. Initialize all vertices as "unvisited".
2. Let Q be a **queue**.
3. Enqueue the root on Q.
4. **While** Q not empty, **do**
5. **begin**
6. n <- Dequeue Q.
7. **If** n is marked as "unvisited", **then**
8. **begin**
9. Mark n as "visited", and output n to the terminal.
10. **For** each vertex v in Adj[n], **do**
11. **If** v is marked "unvisited", **then**
12. enqueue v on Q.
13. **End**
14. **end**

Depth-First Traversal

Let G = (V, E) is a graph which is represented by an adjacency matrix Adj. Assume that nodes in a graph record visited/unvisited information.

procedure DEPTH-FIRST (G)
1. Initialize all vertices as "unvisited".
2. Let S be a stack.
3. Push the root on S.
4. **While** S not empty, **do**
5. **begin**
6. Let n <- Pop S.
7 **If** n is marked as "unvisited", **then**
8. **begin**
9. Mark n as "visited", and output n to the terminal.
10. **For** each vertex v in Adj[n], **do**
11. **If** v is marked as "unvisited", **then** // this test is actually redundant
12. push v on S.
13. **end**
14. **End**

Dijkstra’s Algorithm

Let G = (V, E) which is represented by an adjacency list Adj. Some support data structures:

- d is an array of size |V|. Each d[i] contains the current shortest distance from s to vertex i
- Q is a priority queue of UNKNOWN vertices.
- p is an array of size |V|. Each p[i] contains the parent of vertex i.
- s is the source vertex.

```
procedure DIJKSTRA (G, s)      //  s is the source vertex
1. For every v != s initialize d[v] and p[v] with positive infinity and 0;
   Initialize d[s] = 0, p[s] = 0
2. Let Q be a priority queue
3. Q <- V // initialize Q with all vertices as UNKNOWN
4. While Q not empty, do
5.   begin
6.       u <- ExtractMin(Q)       // Q is modified
7        Mark u as KNOWN       // Dequeuing u is the same as marking it as KNOWN
8        for each vertex v in Adj[u] do
8.         begin
9.             if v is UNKNOWN and d[v] > d[u] + weight(u, v), then do
10.              begin
11.                  d[v] = d[u] + weight(u, v) // update with shorter path
12.                  p[v] = u // update v's parent as u
13.              end
14.        end
15.  end
```