**SFWRENG 2C03**

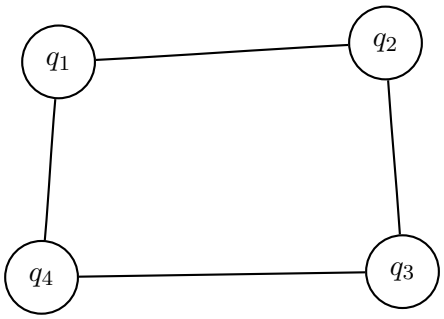
# Assignment 4

**Shazil Arif**

Revised: March 23rd, 2020

1. 4.1.31
   If we have V edges in an undirected graph, there are $\binom{V}{2}$ ways to pick a pair (x,y) of vertices. (Note the order of the pair does not matter since the graph is undirected). Then for these $\binom{V}{2}$ pairs there are E ways to choose the vertices to connect. Then we have :
   $\binom{\binom{V}{2}}{E} = \frac{\binom{V}{2}!}{E!(\binom{V}{2}-E)!}$

   The above is only true if V > E or V < E. Notice that this will not work if $\binom{V}{2}$ = E because the denominator in the above fraction will become 0. This occurs when V = E, so when the number of vertices equals number of edges. In that case there is only one undirected graph. Consider this example with V = E = 4:




   We can see there is only one undirected graph

2. 4.1.32
   Assuming the graph is undirected, every edge shows up exactly twice in the adjacency list. If we allow parallel edges then a given node will show up twice or more in the adjacency list of a particular node. For example if our adjacency list looks like:
   Node : Neighbours
   1 : |2|3|3|
   2 : |3|1|
   3 : |3|2|1|1

   There is a parallel edge between nodes 1 and 3 because they show up twice in the adjacency lists.. To count parallel edges we iterate over the

adjacency list of every node and check if a neighbor is visited/occurs more than once. We can use an additional boolean array to mark visited nodes which will take up space proportional to O(V) but the running time will remain linear proportional to O(V+E). Since every edge shows up twice (because v is connected w AND w is connected to v) when counting the parallel edges we will count twice the actual value, so the number of actual parallel edges will be the total count divided by 2. Pseudocode:

```
//Let G represent the graph
parallel_edges = 0;
for node in G:
    //boolean array to mark visited nodes
    Boolean[] visited = new Boolean[G.num_nodes()];
    for neighbor in node.adjaceny_list():
        if(visited[neighbor]):
            parallel_edges++;
        else:
            visited[neighbor]=True;

return parallel_edges/2; //since edges will show up twice in undirected
```

If the graph is undirected we can just return the total instead of dividing it by 2. This is because v connects to w but w does not necessarily connect to v.

We will iterate over all nodes and edges. Total running time will be O(V+E)

3. 4.2.31
   To find the strong components that contain a given node/vertex s we do the following:

   a. Find the set of vertices reachable from s. To do this simply run a DFS starting from s. When the DFS finishes the regular visited array will contain all the vertices reachable from s. Since this uses DFS, this is done in linear time.

   b. Find the set of vertices that can reach s. To do this take the visited array from step 1 (it contains all nodes reachable from s) then check if each of these nodes can reach s. Run DFS starting from each node, at any point in the DFS if we reach s then s is reachable from the given node. add it to a set. This uses DFS so it will complete in Linear time.

c. Take the intersection of the above two sets since s connects to these nodes AND these nodes connect back to s thus the component is strongly connected.

For a quadratic algorithm that finds the strongly connected components:

a. For every vertex in the graph, find the set of vertices that it can reach
b. Then find the set of vertices that can reach each of the above vertices
c. Take the intersection of the sets from step 1 and 2 which gives strong components

The above does a DFS for every node, thus it takes time V(E+V) and is quadratic

4. 4.2.41
If the digraph G has an odd-length directed cycle, then this cycle will be contained in one of the strong components. If we treat the strong component as an undirected graph, the odd-length directed cycle becomes a regular odd-length cycle. A undirected graph is bipartite if and only if it has no odd-length cycle.

Suppose a strong component of G is nonbipartite (when treated as an undirected graph). This means that there is an odd-length cycle C in the strong component, ignoring direction. If C is a directed cycle, then we are done. Otherwise, if an edge v $\implies$ $w$ is pointing in the "wrong" direction, we can replace it with an odd-length path that is pointing in the opposite direction. To see how, note that there exists a directed path P from w to v because v and w are in the same strong component. If P has odd length, then we replace edge v $\implies$ w by P; if P has even length, then this path P combined with v $\implies$ w is an odd-length cycle.

Ideas borrowed from:
Source: https://algs4.cs.princeton.edu/42digraph/?fbclid=IwAR2c6VSnEo7AVxJKvE-1Mo4MMwHB8WP9CVtjsC8dqbtpmeQeoFl-ONSXvE0

5. 4.2.43
If a vertex in a DAG is reachable from every other vertex it is the "sink" in the topological ordering of the vertices. Then our problems boils down to checking if there is only one sink in the DAG. In this case we have a digraph, not a DAG so instead we have to compute the kernel DAG based on the strong components of the digraph. The same idea of finding one sink applies to the kernel DAG. We can do this by

first computing the strong components using the Kosaraju-Sharir algorithm, get the Kernel DAG and iterate over all nodes in the kernel DAG and checking which nodes have an outdegree of 0. Pseudocode:

```
//get strong components
KosarajuSharirSCC kosarajuSharirSCC = new KosarajuSharirSCC(digraph);

//get kernel dag
Digraph kernel = kosarajuSharirSCC.getKernelDAG();

int sinks = 0;
for node in kernel:
    if node.outdegree()==0:
        sinks++; //this is a sink since its outdegree is 0

return sinks==1;
```

Computing out degree is constant time O(1) since we just check if the adjacency list of the node is empty. Kosaraju-Sharir algorithm is computed in linear time O(V+E). We iterate over V vertices so our total running time is O(V) and is linear.

6. 4.3.20
   True. Since Kruskal's algorithms adds edges in order of their weight (smallest to largest) at any given point the smallest weight edge is added. But, the newly added edge has a weight larger than the previous edge added. This means vertices in the tree connected by the edges added are closer to each other than they are to other vertices in the graph since they are connected by edges whose weight is smaller than the weight of a newly added edge.

7. 4.3.32
   We can use a modified version of Kruskal's algorithm. Let V be the number of vertices in Graph G. We run the same algorithm except when we choose a new minimum weight edge to add to the MST, we first check if the edge is in in the given set S:

   a. If it is in S and it does not generate a cycle with the edges already in the MST, add it to the MST
   b. if it is not in S or adding it generates a cycle with the edges already in the MST then do nothing (continue execution)
   c. Repeat until all edges have been processed or the MST contains V - 1 edges

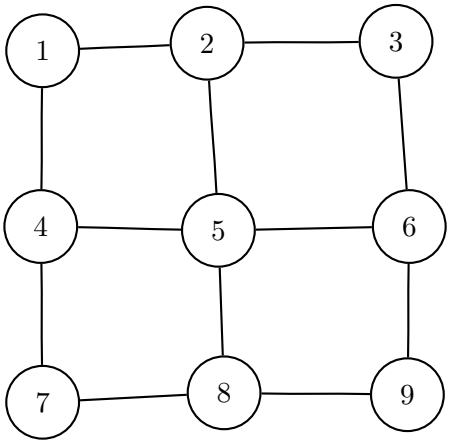
8. 4.4.25
   We can iterate over all the vertices in the subset S, and run Dijkstra's Algorithm using each vertex as the source. When we relax an edge we check if it is in the subset T:

   a. Iterate over all vertices in set S and add them to a priority queue
   b. Run Dijkstra's Algorithm while the priority queue is not empty
   c. Relax the vertex with the minimum weight edge, remove from priority queue
   d. Check if the relaxed vertex is in the subset T, if it is in T, break otherwise continue the algorithm while the queue is not empty

9. 4.4.33
   The main idea is to construct a graph from the matrix cells. The graph can be constructed so that the neighbors of a node are the cells adjacent to a cell in the matrix. This includes the nodes to the left, right, up, down and diagonally top left, top right, bottom left, bottom right, for a total of at most 8 neighbors for each node. Once the graph is constructed we can simply use Dijkstra's algorithm to find the shortest path. If we are restricted to move right and down only then when we construct the graph we simply only take the cells adjacent to a given matrix that are horizontally to its right or vertically down. Once the graph is constructed we can again use Dijkstra's algorithm to find the shortest path. For example if we have:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

   The graph will look like (Assuming we can only move left,right and up, down):



Now we have to come up with a unique index for each node so that it can be represented as a adjacency list. Consider the 2D array as a 1D array row-wise:
1 2 3 | 4 5 6 | 7 8 9
0 1 2 3 4 5 6 7 8

Labelled on the second line is the index/position if the matrix was shown as a 1d array row wise. We can use these indices/position as the node numbers in the adjacency list. We can produce this index using i*n+j where i,j is the row and column number of a given cell and n is the size of matrix. Finally to create this graph from the adjacency list we iterate over all rows and column and take the neighbors of a cell. Pseudocode:

```
for (int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){

     new_node = i*n + j;
     Graph.addNode(new_node)


        x = {1,-1,0,0};
        y = {0,0,1,-1};
        for(int k = 0; k < 4; k++){

         //this will access everything horizontally and vertically adjacent

         x_pos = i + x[k];
         y_pos = j + x[k];

         //check if in bounds of matrix
         if(x_pos >= 0 && x_pos < n && y_pos >=0 && y_pos < n){

         neighbor = x_pos*n + y_pos; //get unique index

                Graph[new_node].adjList.add(neighbor);
         }

        }
    }
}
```

Now run Dijkstra's algorithm on the Graph constructed from matrix to find shortest path.

Source is 0. Destination is: n*n - 1

10. 4.4.47

In Proposition X in the book it states any vertex t that is reachable from s, consider a specific shortest path from s to t: v0 $\Longrightarrow$ v1 $\Longrightarrow$ ... $\Longrightarrow$ vk, where v0 is s and vk is t. If there are no negative cycles, a path exists and k cannot be larger than V - 1. By induction on i it can be shown that after the ith pass the algorithm computes a shortest path from s to vi.

The base case (i = 0) is trivial. Assume the claim to be true for i, v0 $\Longrightarrow$ v1 $\Longrightarrow$ ... $\Longrightarrow$ vi is a shortest path from s to vi, and distTo[vi] is its length. Now, we relax every edge in the ith pass, including vi $\Longrightarrow$ vi+1, so distTo[vi+1] is no greater than distTo[vi] plus the weight of vi $\Longrightarrow$ vi+1. Now, after the ith pass, distTo[vi+1] must be equal to distTo[vi] plus the weight of vi $\Longrightarrow$ vi+1. It cannot be greater because we relax every edge in the ith pass, in particular vi $\Longrightarrow$ vi+1, and it cannot be less because that is the length of v0 $\Longrightarrow$ v1 $\Longrightarrow$ ... $\Longrightarrow$ vi+1, a shortest path. Thus the algorithm computes a shortest path from s to vi+1 after the (i+1)st pass when there are no negative cycles reachable from s.

The highest possible number of edges in a shortest path is V - 1, which is a path that passes through every vertex in the graph. So, if there are no negative cycles reachable from s, all shortest paths will be computed after the V - 1st pass of the algorithm, and no edges will be relaxed in any future passes.
Thus if any edge is relaxed during the V'th pass of the algorithm it means that a new shortest path has been found, using V edges. This would be possible in the presence of a negative cycle. Since the edgeTo[] array stores the edges on the shortest paths found, it will have a directed cycle and this would have to be a negative cycle.

Ideas borrowed from page 671 of the course textbook/Algorithms 4th Edition