You are allowed to discuss with others but not allow to use references other than the course notes and reference books. Please list your collaborators for each questions. Write your own solutions and make sure you understand them.

There are 55 marks in total (including the bonus). The full mark of this homework is 50.

Enjoy :).

Please specify the following information before submission:

- Your Name: Zhenming Wang
- Your NetID: zw1806
- Collaborators:

Problem 1: An important edge [10 marks]

An important edge in a connected undirected graph $G = (V, E)$ is an edge that, if removed, would disconnect G. Give an $O(|V| + |E|)$ algorithm to determine whether G has an important edge.

Solution. **Please write down your solution to Problem 1 here.**

```
adj: adjacency list of graph
visited : list of visited node
timer: global variable time indicating current time
disc: list of discovery time of nodes
low: list of discovery time of nodes
    that is closest to the starting root for every vertex v

n: number of veritces

def importantEdge(v,parent=-1):
    visited[u] = True
    disc[u] = time
    low[u] = time
    timer += 1
    for v in graph[u]:
        if visited[u] == False:
            parent[v] = u
            importantEdge(v,u)
            low[u] = min(low[u],low[v])
            if low[v] > disc[u] and u != -1:
                return True //find a bridge

            children += 1
        elif visited[v] == True:
            low[u] = min(low[u], disc[v])

    // if current vertex is root and have more than one child
    if children > 1 and parent == -1:
        return True

def main():
    timer = 1
    visited = [false for v in range(n)]
    disc = [-1 for v in range(n)]
    low = [-1 for v in range(n)]
    s = an arbitrary starting node
    importantEdge(v,-1)
```

Problem 2: Unique Shortest Path [10 marks]

Given an undirected graph $G = (V, E)$ and two vertices $s, t \in V$, design a linear time algorithm to determine if there is a unique shortest path from s to t.

Solution. **Please write down your solution to Problem 2 here.**
By doing BFS, we could find the shortest distance from each vertex to the starting node. We make some modifiction to it so that in the meantime we maintain the shortest distance for each vertex and the number of shortest paths from one vertex to the starting point.

```
def shortestPath(Graph,|V|):
    visited  = [false for i in range(|V|)]
    distance = [infty for i in range(|V|)]
    num_of_path = [0 for i in range(|V|)]
    prev = [None for i in range(|V|)]

    queue Q //empty queue
    enqueue(Q,s)
    visited[s] = true
    distance[s] = 0
    num_of_path[s] = 1

    while Q is not empty:
        u = dequeue(Q)
        for each neighbour v of u:
            if visited[v] == false:
                enqueue(Q,v)
                visited[v] = True
                parent[v] = u
                distance[v] = distance[v] + 1
                num_of_path[v] = num_of_path[u]

            elif visited[v] == true:
                alt = distance[u] + 1

                //we have found a shorter path, update
                if alt < distance[v]:
                    distance[v] = alt
                    prev[v] = u
                    num_of_path[v] = num_of_path[u]

                // we have found an equal path, update num_of_path
                if alt == distance[v]:
                    num_of_path[v] += 1
```

Problem 3: Reachability [10 marks]

Given an undirected graph $G = (V, E)$ design a linear time algorithm to find a vertex $s \in V$ from which all other vertices are reachable (i.e. there is a directed path from s to v for all $v \in V$) or report that none exists.

Solution. **Please write down your solution to Problem 3 here.**
Since we are considering an undirected graph, so if we could reach all other vertices from one arbitrary vertex, that means we could start from any other nodes and reach an arbitrary node through this vertex. We simply need to perform BFS once, and traverse through the visited list and check if all the nodes have been visited.

```
def reachability(Graph,|V|):
    visited = [false for i in range(|V|)]
    queue Q //empty

    enqueue(Q,s)
    visited[s] = true

    while Q is not empty:
        u = dequeue(Q)
        for each neighbour v of u:
            if visited[v] == false:
                enqueue(Q,v)
                visited[v] = true
    for i in visited:
        if i == false:
            return false

    return true
```

Problem 4: Many stars [10 marks]

The following algorithm is performed on a connected undirected graph $G = (V, E)$, implemented by adjacency lists.

```
void mystery(v) {
    color v blue
    For each neighbor u of v
        if u is blue
            print *
        if u is red then
            Mystery(u)
}
int main() {
    color all vertices red
    Select an arbitrary vertex u
    mystery(u)
}
```

How many starts will be printed along the execution of `main`? Briefly explain your answer.

Solution. **Please write down your solution to Problem 4 here.**
The number of stars is the number of edges, $|E|$.
The algorithm is performing a DFS, and would output a star when a vertex on the other side of an edge is already visited. When we are at an arbitrary vertex at some point,denote by s_1 and are running the for loop of its neighbour, when a neighbour is already visited before, we would output a star, otherwise we would go to that node, denote by s_2,and at some point inside s_2's for loop, we would cover the edge (s_1,s_2), and a star would be outputted. An edge would not be visited twice during a DFS, and a star is printed only when the two vertices on both sides have been visited.

Problem 5: Odd Cycle [10+5 marks]

Given a strongly connected directed graph $G = (V, E)$, design a linear time algorithm to determine if there is a directed odd cycle in G or not.

Bonus: (5 marks) Prove the correctness of your algorithm.

Solution. **Please write down your solution to Problem 5 here.**

(a) If there does not exist an odd cycle, then the graph G is bypartite. For a bypartite graph, it's possible to color the whole graph using two colors, and the color of two nodes on two sides of an edge should be different. By using a BFS, we could get the answer.

```
color = [none for i in range(n)]
queue Q is empty
pick a source vertex s
enqueue(Q,s)
color[s] = 1

while Q is not empty:
    u = dequeue(Q)
    for each neighbor v of u:
        if color[v] = none:
            color[v] = 1 - color[u] //color with reversed color
            enqueue(Q,v)
        elif color[v] = color[u]:
            return True // found an odd cycle
return False
```

(b) *Proof.* Our G here is strongly connected, so BFS starting at an arbitrary vertex s is going to cover all the edges. It is well known that if the graph is bypartite, then we would be able to divide the vertices in two sets, and here in the algorithm, we use two opposing color to represent the two sets.

Now, I want to prove that if a graph is not bypartite, then there must exists at least one odd cycle.

If a graph is non-bypartite, there must be at least one edge connecting two nodes either of set A(odd distances from starting vertex) or of set B(even distanecs from starting vertex), which means there must be at least one edge connecting either two vertices of odd distance from starting vertex, or two vertices of even distance from starting vertex.

Let's say that one edge connects two vertices each having an even distance from starting vertex, then the length of the cycle including the starting vertex and the two vertices will be, Even + Even + 1, which is odd. Thus, we have found an odd length cycle.

So during BFS, if we have never seen an edge (u, v) that goes between vertices of the same parity then G is bipartite, and hence there does not exist an odd cycle. Otherwise, the graph is not bypartite and hence there exists an odd cycle.

□