Contents

1. Notations
   - Directed graph G =<V, E> is said to be strongly connected if there is a path between every pair of vertices.
   - Directed graph G =<V, E> is said to be weakly connected if its corresponding undirected graph is connected.
2. Searching in Graphs
   2.1. DFS using Stack

```
DFS(u)
{
```

```
    // Step 1: Initialize
    stack = Ø; push(stack, u);
    <visit u>; visited[u] = True;

    // Step 2: Loop
    while(stack ≠ Ø)
    {
        s = pop(stack);
        for(t ∈ Adj(s))
        {
            if(!visited[t])
            {
                <visit t>; visited[t] = True;
                push(stack, s); push(stack, t);
                break;
            }
        }
    }

    // Step 3: Return results
    return <set of traversed vertices>;
}
```

2.2.BFS

```
BFS(u)
{
    // Step 1: Initialize
    queue = Ø; push(queue, u);
    visited[u] = True;

    // Step 2: Loop
    while(queue ≠ Ø)
    {
        s = pop(queue);
        <visit s>;
        for(t ∈ Adj(s))
            if(!visited[t])
            {
                visited[t] = True;
                push(queue, t);
            }
    }

    // Step 3: Return results
```

```
    return <set of traversed vertices>;
}
```

2.3. Determine the number of connected components

```
CountComp()
{
    // Step 1: Initialize
    count = 0;

    // Step 2: Loop
    for(u ∈ V){
        if(!visited[u]){
            count = count + 1;
            BFS(u); // DFS(u)
            <store vertices of the connected component>;
        }
    }

    // Step 3:
    return <connected components>;
}
```

2.4. Finding paths between vertices

To store the path, we use the array previous[].When push t ∈ Adj(u) to the stack or queue, we set previous[t] = u

❖ For DFS:

```
DFS(u)
{
    // Step 1: Initialize
    stack = ∅; push(stack, u);
    visited[u] = True;

    // Step 2: Loop
    while(stack ≠ ∅)
    {
        s = pop(stack);
        for(t ∈ Adj(s))
        {
            if(!visited[t])
            {
                visited[t] = True;
                push(stack, s); push(stack, t);
                previous[t] = s;
                break;
```

```
            }
        }
    }

    // Step 3: Return results
    return <set of traversed vertices>;
}
```

❖ For BFS:

```
BFS(u)
{
    // Step 1: Initialize
    queue = ∅; push(queue, u);
    visited[u] = True;

    // Step 2: Loop
    while(queue ≠ ∅)
    {
        s = pop(queue);
        for(t ∈ Adj(s))
            if(!visited[t])
            {
                visited[t] = True;
                previous[t] = s;
                push(queue, t);
            }
    }

    // Step 3: Return results
    return <set of traversed vertices>;
}
```

2.5. Strongly Connected Property of Directed Graph

```
bool Strongly_Connected(G = <V, E>)
{
    ReInit(); // for u ∈ V: visited[u] = False
    for(u ∈ V)
        if(BFS(u) ≠ V) // DFS(u) ≠ V
            return false;
        else ReInit();
    return true;
}
```

2.6. Finding Cut Vertices

```
Finding_Cut_Vertices(G = <V, E>)
{
    ReInit(); // for u ∈ V: visited[u] = False
    for(u ∈ V)
```

```
    {
        visited[u] = True;
        if(BFS(v) ≠ V\{u}) // DFS(v) ≠ V\{u}
            <u is a cut vertex>;
        ReInit();
    }
}
```

2.7. Finding Bridges

```
Finding_Bridges(G = <V, E>)
{
    ReInit(); // for u ∈ V: visited[u] = False
    for(e ∈ E)
    {
        E = E \ {e}
        if(BFS(1) ≠ V) // DFS(1) ≠ V
            <e is a bridge>;
        E = E U {e};
        ReInit();
    }
}
```

3. Eulerian and Hamiltonian Graph

3.1. Eulerian Graph:

❖ Necessary and Sufficient Conditions for Eulerian Graph:

- Undirected graph: Connected undirected graph G = <V, E> is Eulerian graph if and only if every vertex of G has even degree.
- Directed graph: Weakly-connected directed graph G = <V, E> is Eulerian graph if and only if in-degree of each vertex equals to its out-degree.

❖ Eulerian Graph Proof:

- Undirected graph: $\begin{cases} \mathrm{DFS}(u) = \mathrm{BFS}(u) = V \\ \deg(u): \text{even?}\ (\forall u \in V) \end{cases}$
- Directed graph: $\begin{cases} \exists u \in V: \mathrm{DFS}(u) = \mathrm{DFS}(u) = V \\ \deg^{+}(u) = \deg^{-}(u)\ (\forall u \in V) \end{cases}$

❖ Present algorithm:

```
Euler_Cycle(u)
{
    // Step 1: Initialize
    stack = ∅;
    CE = ∅;
    push(stack, u);

    // Step 2: Loop
    while(stack ≠ ∅)
    {
        s = get(stack);
```

```
        if(Adj(s) ≠ ∅)
        {
            t = <the first vertex in Adj(s)>;
            push(stack, t);
            E = E \ {(s, t)};
        }
        else
        {
            s = pop(stack);
            s ⇒ CE;
        }
    }

    // Step 3: Result
    <overturning vertices in CE>;                                        ∈
}
```

3.2. Semi-Eulerian Graph

- Necessary and Sufficient Conditions for Semi-Eulerian Graph:
  - Undirected graph: Connected undirected graph G = < V, E > is semi-Eulerian if and only if G has 2 vertices with odd degree.
  - Directed graph: Weakly-connected directed graph G = <V, E> is semi-Eulerian if and only if:
    - There exits exactly 2 vertices u, v ∈ V s.t: $deg^+(u) – deg^-(u) = deg^-(v) – deg^+(v) = 1$
    - For s ≠ u, v: $deg^+(s) = deg^-(s)$
- Semi-Eulerian Graph Proof:
  - Undirected graph:
    - DFS(u) = BFS(u) = V
    - Has 2 vertices with odd degree
  - Directed graph:
    - DFS(u) = BFS(u) = V
    - u, v ∈ V s.t: $deg^+(u) – deg^-(u) = deg^-(v) – deg^+(v) = 1$
    - For s ≠ u, v: $deg^+(s) = deg^-(s)$
- Notes: Algorithm for finding an Eulerian path is similar to the one for finding Eulerian circuit. However:
  - Finding Eulerian circuit: starting from any u ∈ V
  - Finding Eulerian path:
    - Undirected graph: starting from an odd-degree
    - Directed graph: starting from u: $deg^+(u) – deg^-(u) = 1$

3.3. Hamiltonian Graph:

```
Hamilton(k)
{
    for(y ∈ Adj(X[k - 1]))
```

```
        if(k == n + 1 && y == v)
            Out(X[1], X[2], ..., X[n], v);
        else if(!visisted[y])
        {
            X[k] = y;
            visited[y] = True;
            Hamilton(k + 1);
            visited[y] = False;
        }
}
```

4. Minimum Spanning Trees:
   4.1. Spanning Tree:

```
Tree_DFS(u)
{
    visited[u] = True;
    for(v ∈ Adj(u))
        if(!visited[v])
        {
            T = T U {(u, v)};
            Tree_DFS(v);
        }
}
```

```
Tree_BFS(u)
{
    // Step 1: Initialize
    T = ∅;
    queue = ∅;
    push(queue, u);
    visited[u] = True;

    // Step 2: Loop
    while(queue ≠ ∅)
    {
        s = pop(queue);
        for(t ∈ Adj(s))
            if(!visited[t])
            {
                T = T U {(s, t)};
                visited[t] = True;
                push(queue, t);
            }
    }

    // Step 3: Return results
    if(|T| < n - 1)
```

```
        <graph is not connected>;
    else
        return <set of edges of spanning tree T>;
}
```

4.2. Kruskal:

```
Kruskal()
{
    // Step 1: Initialize
    T = Ø;
    d(H) = 0;

    // Step 2: Sort
    <Sort edges in the ascending order of length>;

    // Step 3: Loop
    while(|T| < n - 1 && E ≠ Ø)
    {
        e = <minimum length edge>;
        E = E \ {e};
        if(T U {e} does not produce a circuit)
        {
            T = T U {e};
            d(H) = d(H) + d(e);
        }
    }
    if(|T| < n - 1) <not connected>;
    else return {T, d(H)};
}
```

4.3. Prim:

```
Prim(s)
{
    // Step 1: Initialize
    Vh = {s};
    V = V \ {s};
    T = Ø;
    d(H) = 0;

    // Step 2: Loop
    while(V ≠ Ø)
    {
        e = {u, v}; // u ∈ V, v ∈ Vh
        if(e does not exist)
            return <not connected>;
        T = T U {e};
```

```
        d(H) = d(H) + d(e);
        Vh = Vh U {u};
        V = V \ {u};
    }

    // Step 3: Result
    return {T, d(h)};
}
```

5. Shortest Path Problem

5.1. Dijkstra:

```
Dijkstra(s)
{
    // Step 1: Initialize
    d[s] = 0;
    T = V \ {s};
    for(v ∈ V)
    {
        d[v] = a(s, v);
        pre[v] = s;
    }

    // Step 2: Loop
    while(T ≠ ∅)
    {
        d[u] = min(d[z]| z ∈ T)
        T = T \ {u};
        for(v ∈ T)
            if(d[v] > d[u] + a(u, v))
            {
                d[v] = d[u] + a(u, v);
                pre[v] = u;
            }
    }
}
```

5.2. Bellman-Ford:

```
Bellman-Ford(s)
{
    // Step 1: Initialize
    d[s] = 0;
    for(v ∈ V)
    {
        d[v] = a(s, v);
        pre[v] = s;
    }

```

```
    // Step 2: Loop
    for(k = 1; k <= n - 1; k++)
        for(v ∈ V\{s})
            for(u ∈ V)
                if(d[v] > d[u] + a(u, v))
                {
                    d[v] = d[u] + a(u, v);
                    pre[v] = u;
                }
}
```

5.3.Floyd:

```
Floyd()
{
    // Step 1: Initialize
    for(i = 1; i <= n; i++)
        for(j = 1; j <= n; j++)
        {
            d[i, j] = a(i, j);
            if(a(i, j) != ∞) next[i, j] = i;
            else next[i, j] = null;
        }

    // Step 2: Loop
    for(k = 1; k <= n; k++)
        for(i = 1; i <= n; i++)
            for(j = 1; j <= n; j++)
                if(d[i, j] > d[i, k] + d[k, j])
                {
                    d[i, j] = d[i, k] + d[k, j];
                    next[i, j] = next[i, k];
                }
}
```