Contents

1. Searching in Graphs

1.1. 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>;
}
```

1.2. BFS

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

    // Step 2: Loop
    while(queue ≠ ∅)
    {
        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>;
}
```

1.3. 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 count;
}
```

1.4. 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>;
}
```

1.5. 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;
}
```

1.6. 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();
    }
}
```

1.7. 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();
    }
}
```

2. Eulerian and Hamiltonian Graph

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

❖ 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>;                                                ∈
}
```

2.2. 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 $\in$ V s.t: $deg^+(u) – deg^-(u) = deg^-(v) – deg^+(v) = 1$
  - For $s \neq u, v$: $deg^+(s) = deg^-(s)$

❖ 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 $\in$ V s.t: $deg^+(u) – deg^-(u) = deg^-(v) – deg^+(v) = 1$
  - For $s \neq u, v$: $deg^+(s) = deg^-(s)$

❖ Notes: Algorithm for finding an Eulerian path is similar to the one for finding Eulerian circuit. However:

- Finding Eulerian circuit: starting from any u $\in$ V
- Finding Eulerian path:
  - Undirected graph: starting from an odd-degree
  - Directed graph: starting from u: $deg^+(u) – deg^-(u) = 1$

2.3. 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;
        }
}
```

3. Minimum Spanning Trees:

3.1. Kruskal:

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

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

    // Step 3: Loop
    while(|T| < n - 1 && E ≠ ∅)
    {
        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)};
}
```

3.2. Prim:

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

    // Step 2: Loop
    while(V ≠ ∅)
    {
        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)};
}
```

4. Shortest Path Problem

4.1. 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 ≠ Ø)
    {
        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;
            }
    }
}
```

4.2. 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;
                }
}
```

4.3. 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];
                }
}
```