Date of the Session:__/__/_____ **Time of the Session:_____to_____**

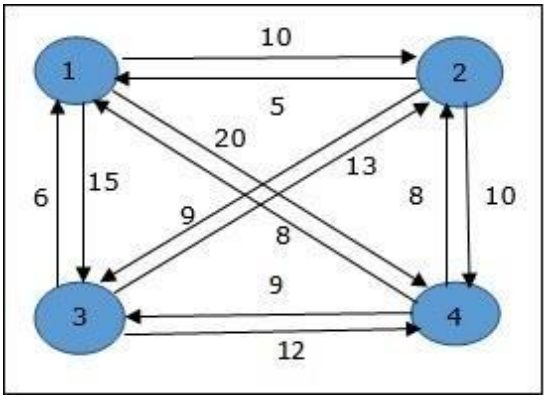
EX – 8 Working with TSP problem, Ford Fulkerson and Graph Traversal Techniques

Prerequisites:

- Basics of Data Structures and C Programming.
- Basic knowledge about Graphs.

Pre-lab:

1) Your father brought you a ticket to world tour. You have a choice to go to three places, your father knows the places you wanted to travel so he made a graph with the measuring distances from home. Now you start from your place (1: Source) to other places as shown in the graph below apply TSP to find shortest path to visit all places and return to home. (Ex: 2: London, 3: Paris, 4: Singapore)



[@KLWKS_BOT] THANOS

Step 1: Extract Given Distances

The graph is represented as follows:

- 1 → 2 = 10
- 1 → 3 = 15
- 1 → 4 = 20
- 2 → 1 = 5
- 2 → 3 = 9
- 2 → 4 = 10
- 3 → 1 = 6
- 3 → 2 = 13
- 3 → 4 = 12
- 4 → 1 = 8
- 4 → 2 = 8
- 4 → 3 = 9

Step 2: Evaluate All Possible Routes

We need to visit all cities (2, 3, 4) and return to 1 with minimal cost.

Possible Routes and Their Costs

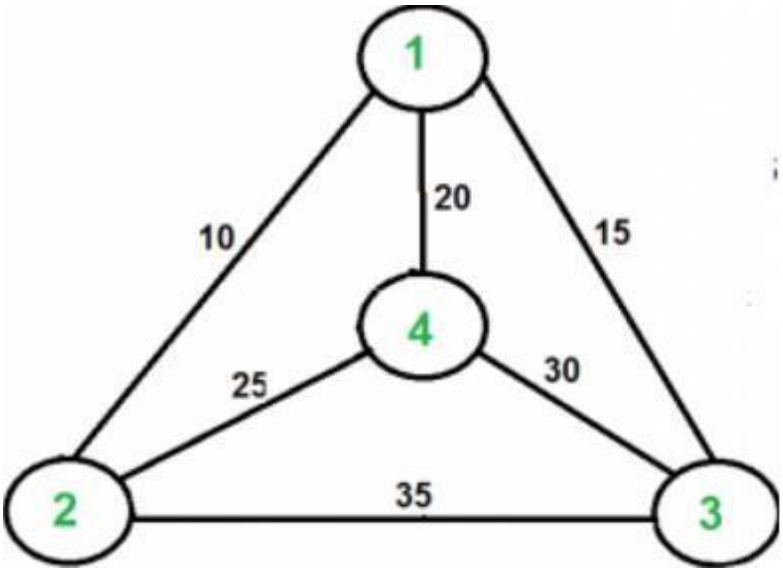
1. $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$
   - $10 + 9 + 12 + 8 = 39$
2. $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$
   - $10 + 10 + 9 + 6 = 35$ ✅
3. $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 1$
   - $15 + 13 + 10 + 8 = 46$
4. $1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 1$
   - $15 + 12 + 8 + 5 = 40$
5. $1 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1$
   - $20 + 8 + 9 + 6 = 43$
6. $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$
   - $20 + 9 + 13 + 5 = 47$

Step 3: Optimal Solution

The shortest path for TSP is:

$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$ with a total cost of 35.

2) You are given a map(graph) with the distances between the places. Here you will consider the 1st node as the starting point and ending point, Since the route is cyclic you can take any node as starting point and ending point. Find the minimum cost route and remember the Hamiltonian cycle.



Graph Representation

Vertices: `{1, 2, 3, 4}`

Edges with weights:

- (1,2) = 10
- (1,3) = 15
- (1,4) = 20
- (2,3) = 35
- (2,4) = 25
- (3,4) = 30

Solving for Minimum Cost Hamiltonian Cycle

A **Hamiltonian cycle** visits all nodes exactly once and returns to the starting node. We compute the total weight for all possible cycles and choose the minimum.

Possible Hamiltonian Cycles:

1. **1 → 2 → 3 → 4 → 1**
   Cost = `10 + 35 + 30 + 20 = 95`
2. **1 → 2 → 4 → 3 → 1**
   Cost = `10 + 25 + 30 + 15 = 80`
3. **1 → 3 → 2 → 4 → 1**
   Cost = `15 + 35 + 25 + 20 = 95`
4. **1 → 3 → 4 → 2 → 1**
   Cost = `15 + 30 + 25 + 10 = 80`
5. **1 → 4 → 2 → 3 → 1**
   Cost = `20 + 25 + 35 + 15 = 95`
6. **1 → 4 → 3 → 2 → 1**
   Cost = `20 + 30 + 35 + 10 = 95`

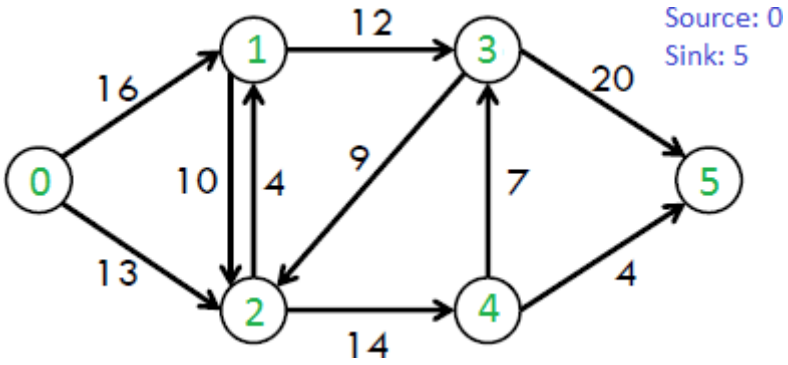
Minimum Cost Hamiltonian Cycle

The **optimal route** with minimum cost is:

- **1 → 2 → 4 → 3 → 1** OR **1 → 3 → 4 → 2 → 1**
- **Minimum cost = 80**

In-Lab:

1) Emma has a graph which represents a flow network where every edge has a capacity. Also given two vertices source s and sink t in the graph, fin the maximum possible flow from s to t with following constraints:
   a. Flow on an edge does not exceed the given capacity of the edge.
   b. Incoming flow is equal to outgoing flow for every vertex except s and t.



Find the Maximum flow using the graph and by implementing the code.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

struct Edge {
   int flow, capacity, u, v;
};

struct Vertex {
   int h, e_flow;
};

struct Graph {
   int V;
   struct Vertex *ver;
   struct Edge *edge;
   int edge_count;
};

struct Graph* createGraph(int V) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->V = V;
   graph->ver = (struct Vertex*)malloc(V * sizeof(struct Vertex));
   graph->edge = (struct Edge*)malloc(100 * sizeof(struct Edge)); // Adjust size as needed
```

```
    graph->edge_count = 0;
    for (int i = 0; i < V; i++)
        graph->ver[i] = (struct Vertex){0, 0};
    return graph;
}

void addEdge(struct Graph* graph, int u, int v, int capacity) {
    graph->edge[graph->edge_count++] = (struct Edge){0, capacity, u, v};
}

void preflow(struct Graph* graph, int s) {
    graph->ver[s].h = graph->V;
    for (int i = 0; i < graph->edge_count; i++) {
        if (graph->edge[i].u == s) {
            graph->edge[i].flow = graph->edge[i].capacity;
            graph->ver[graph->edge[i].v].e_flow += graph->edge[i].flow;
            graph->edge[graph->edge_count++] = (struct Edge){-graph->edge[i].flow, 0, graph-
>edge[i].v, s};
        }
    }
}

int overFlowVertex(struct Graph* graph) {
    for (int i = 1; i < graph->V - 1; i++)
        if (graph->ver[i].e_flow > 0)
            return i;
    return -1;
}

void updateReverseEdgeFlow(struct Graph* graph, int i, int flow) {
    int u = graph->edge[i].v, v = graph->edge[i].u;
    for (int j = 0; j < graph->edge_count; j++) {
        if (graph->edge[j].v == v && graph->edge[j].u == u) {
            graph->edge[j].flow -= flow;
            return;
        }
    }
    graph->edge[graph->edge_count++] = (struct Edge){0, flow, u, v};
}

int push(struct Graph* graph, int u) {
    for (int i = 0; i < graph->edge_count; i++) {
        if (graph->edge[i].u == u) {
```

```
            if (graph->edge[i].flow == graph->edge[i].capacity)
                continue;
            if (graph->ver[u].h > graph->ver[graph->edge[i].v].h) {
                int flow = (graph->edge[i].capacity - graph->edge[i].flow < graph->ver[u].e_flow) ?
                        graph->edge[i].capacity - graph->edge[i].flow : graph->ver[u].e_flow;
                graph->ver[u].e_flow -= flow;
                graph->ver[graph->edge[i].v].e_flow += flow;
                graph->edge[i].flow += flow;
                updateReverseEdgeFlow(graph, i, flow);
                return 1;
            }
        }
    }
    return 0;
}

void relabel(struct Graph* graph, int u) {
    int mh = INT_MAX;
    for (int i = 0; i < graph->edge_count; i++) {
        if (graph->edge[i].u == u) {
            if (graph->edge[i].flow == graph->edge[i].capacity)
                continue;
            if (graph->ver[graph->edge[i].v].h < mh) {
                mh = graph->ver[graph->edge[i].v].h;
                graph->ver[u].h = mh + 1;
            }
        }
    }
}

int getMaxFlow(struct Graph* graph, int s, int t) {
    preflow(graph, s);
    while (overFlowVertex(graph) != -1) {
        int u = overFlowVertex(graph);
        if (!push(graph, u))
            relabel(graph, u);
    }
    return graph->ver[t].e_flow;
}

int main() {
    int V = 6;
    struct Graph* g = createGraph(V);
```

```
    addEdge(g, 0, 1, 16);
    addEdge(g, 0, 2, 13);
    addEdge(g, 1, 2, 10);
    addEdge(g, 2, 1, 4);
    addEdge(g, 1, 3, 12);
    addEdge(g, 2, 4, 14);
    addEdge(g, 3, 2, 9);
    addEdge(g, 3, 5, 20);
    addEdge(g, 4, 3, 7);
    addEdge(g, 4, 5, 4);
    int s = 0, t = 5;
    printf("Maximum flow is %d\n", getMaxFlow(g, s, t));
    free(g->ver);
    free(g->edge);
    free(g);
    return 0;
}
```

2) For the above given graph Emma wants an s-t to cut that requires the source s and the sink t to be in different subsets, and it consists of edges going from the source's side to the sink's side. So, she wants you to find the minimum capacity s-t cut of the given network. Expected output is all the edges of minimum cut.

Source code:

```
# include <stdio.h>
# include <stdlib.h>
# include <limits.h>

#define N 6
#define MIN(a, b) ((a) < (b) ? (a) : (b))

typedef struct {
   int u, v, cap, flow;
} Edge;

Edge edges[20];
int height[N], excess[N], edgeCount, visited[N];

void addEdge(int u, int v, int cap) {
   edges[edgeCount++] = (Edge){u, v, cap, 0};
   edges[edgeCount++] = (Edge){v, u, 0, 0};
}

void push(int i) {
   int flow = MIN(excess[edges[i].u], edges[i].cap - edges[i].flow);
   edges[i].flow += flow;
   edges[i^1].flow -= flow;
   excess[edges[i].u] -= flow;
   excess[edges[i].v] += flow;
}

void relabel(int u) {
   int minH = INT_MAX;
   for (int i = 0; i < edgeCount; i++)
      if (edges[i].u == u && edges[i].flow < edges[i].cap)
         minH = MIN(minH, height[edges[i].v]);
   height[u] = minH + 1;
}

void dfs(int u) {
```

```
    visited[u] = 1;
    for (int i = 0; i < edgeCount; i++)
        if (edges[i].u == u && edges[i].flow < edges[i].cap && !visited[edges[i].v])
            dfs(edges[i].v);
}

void minCut(int s, int t) {
    for (int i = 0; i < N; i++) visited[i] = 0;
    dfs(s);
    printf("Minimum s-t cut edges:\n");
    for (int i = 0; i < edgeCount; i++)
        if (visited[edges[i].u] && !visited[edges[i].v])
            printf("%d -> %d\n", edges[i].u, edges[i].v);
}

int maxFlow(int s, int t) {
    height[s] = N;
    for (int i = 0; i < edgeCount; i++)
        if (edges[i].u == s) {
            edges[i].flow = edges[i].cap;
            excess[edges[i].v] += edges[i].cap;
            excess[s] -= edges[i].cap;
        }
    while (1) {
        int pushed = 0;
        for (int i = 0; i < edgeCount; i++)
            if (excess[edges[i].u] > 0 && height[edges[i].u] > height[edges[i].v]) {
                push(i);
                pushed = 1;
            }
        if (!pushed) break;
        for (int u = 1; u < N - 1; u++)
            if (excess[u] > 0) relabel(u);
    }
    return excess[t];
}
```

```
int main() {
   addEdge(0, 1, 16); addEdge(0, 2, 13);
   addEdge(1, 2, 10); addEdge(2, 1, 4);
   addEdge(1, 3, 12); addEdge(2, 4, 14);
   addEdge(3, 2, 9);  addEdge(3, 5, 20);
   addEdge(4, 3, 7);  addEdge(4, 5, 4);

   printf("Maximum flow is %d\n", maxFlow(0, 5));
   minCut(0, 5);
   return 0;
}
```

Post-Lab:

1) Hogwarts has yet again declared The Triwizard tournament. Harry must pass this round to get to the next one. Each participant will be given a graph with vertices as shown below. Each vertex is a dungeon, and a golden egg is placed in the root dungeon i.e., the root vertex. Help harry find the dungeon with the golden egg using traversing or searching tree or graph data structure. (P.S: To pass this round Harry must write a code).

Source code:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_NODES 100

typedef struct {
   int adj[MAX_NODES][MAX_NODES];
   int vertices;
} Graph;

void init_graph(Graph *g, int vertices) {
   g->vertices = vertices;
   for (int i = 0; i < vertices; i++) {
      for (int j = 0; j < vertices; j++) {
         g->adj[i][j] = 0;
      }
   }
}

void add_edge(Graph *g, int u, int v) {
   g->adj[u][v] = 1;
}

void bfs(Graph *g, int start) {
   int queue[MAX_NODES], front = 0, rear = 0;
   int visited[MAX_NODES] = {0};

   queue[rear++] = start;
   visited[start] = 1;

   while (front < rear) {
      int node = queue[front++];
      printf("Dungeon %d checked\n", node);
      for (int i = 0; i < g->vertices; i++) {
```

```
            if (g->adj[node][i] && !visited[i]) {
                queue[rear++] = i;
                visited[i] = 1;
            }
        }
    }
    printf("Golden egg found at dungeon: %d\n", start);
}

int main() {
    Graph g;
    init_graph(&g, 8);
    add_edge(&g, 1, 2);
    add_edge(&g, 1, 3);
    add_edge(&g, 2, 4);
    add_edge(&g, 2, 5);
    add_edge(&g, 3, 6);
    add_edge(&g, 3, 7);

    int root_dungeon = 1;
    bfs(&g, root_dungeon);
    return 0;
}
```

2) KL University is starting next week. There are S subjects in total, and you need to choose K of them to attend each day, you required number of credits to pass the semester. There are N+1 buildings. Your hostel is in building number 0. Subject j is taught in building Bj. After each subject, you have a break, during which you go back to your hostel. There are M bidirectional paths of length 1 which connects building b1 to building b2. Find the minimum total distance you need to travel each day if you choose your subjects correctly.

Input

2 3 2 2
0 1
1 2
2 0
1 2

Output

4

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAX_BUILDINGS 100

int graph[MAX_BUILDINGS][MAX_BUILDINGS];
int distances[MAX_BUILDINGS], visited[MAX_BUILDINGS];

void bfs(int start, int n) {
  for (int i = 0; i <= n; i++) distances[i] = INT_MAX, visited[i] = 0;
  distances[start] = 0;

  for (int i = 0; i <= n; i++) {
    int minDist = INT_MAX, u = -1;
    for (int j = 0; j <= n; j++) {
      if (!visited[j] && distances[j] < minDist) {
        minDist = distances[j];
        u = j;
      }
    }
    if (u == -1) break;
    visited[u] = 1;
    for (int v = 0; v <= n; v++) {
      if (graph[u][v] && distances[u] + 1 < distances[v]) {
        distances[v] = distances[u] + 1;
      }
    }
  }
```

```
}

int compare(const void* a, const void* b) {
   return (*(int*)a - *(int*)b);
}

int main() {
   int N = 2, S = 3, K = 2, M = 2;
   int edges[2][2] = {{0, 1}, {1, 2}};
   int subjects[] = {1, 2};

   for (int i = 0; i <= N; i++)
      for (int j = 0; j <= N; j++)
         graph[i][j] = 0;

   for (int i = 0; i < M; i++) {
      int u = edges[i][0], v = edges[i][1];
      graph[u][v] = graph[v][u] = 1;
   }

   bfs(0, N);

   int subjectDistances[S];
   for (int i = 0; i < S; i++) subjectDistances[i] = distances[subjects[i]];
   qsort(subjectDistances, S, sizeof(int), compare);

   int totalDistance = 0;
   for (int i = 0; i < K; i++) totalDistance += 2 * subjectDistances[i];

   printf("%d\n", totalDistance);
   return 0;
}
```

Comments of the Evaluators (if Any)	Evaluator's Observation Marks Secured:_____out of [50]. Signature of the Evaluator Date of Evaluation: