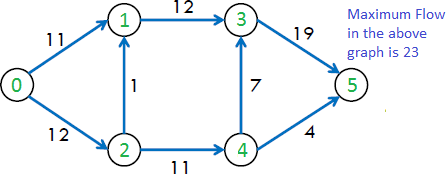
**AlgorithmsandProblem-SolvingLab(15B17CI471) EVEN 2024**

**Week-4(19thFeb–25thFeb2024) Assignment-5**

**Topic:NetworkFlow**



**Q1. Given a graph which represents a flow network where everyedge has a capacity. Also given two vertices *source* ‘s’ and *sink* ‘t’ in the graph, write a program to find the maximum possible flow from s to tusing Ford Fulkerson algorithm with following constraints:**

1. **Flowonanedgedoesn’texceedthegivencapacityofthe edge.**
2. **Incomingflowisequaltooutgoingflowforeveryvertexexcepts and t.**

#include <climits>

using namespace std;

#define N 100

int graph[N][N], residualGraph[N][N];

int parent[N], visited[N];

int min(int a, int b)

{

return (a < b) ? a : b;

}

int dfs(int u, int t, int flow, int n)

{

if (u == t)

return flow;

visited[u] = 1;

for (int v = 0; v < n; v++)

if (!visited[v] && residualGraph[u][v] > 0)

int min\_flow = min(flow, residualGraph[u][v]);

int path\_flow = dfs(v, t, min\_flow, n);

if (path\_flow > 0)

{

residualGraph[u][v] -= path\_flow;

residualGraph[v][u] += path\_flow;

return path\_flow;

}

return 0;

}

int fordFulkerson(int s, int t, int n)

{

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

residualGraph[i][j] = graph[i][j];

int maxFlow = 0;

while (true)

{

for (int i = 0; i < n; i++)

visited[i] = 0;

int path\_flow = dfs(s, t, INT\_MAX, n);

if (path\_flow == 0)

break;

maxFlow += path\_flow;

}

return maxFlow;

}

int main()

{

int n, m, u, v, capacity, s, t;

cout<<"Input the no. of vertices : ";

cin >> n;

cout<<"Input the no. of edges : ";

cin >> m;

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

graph[i][j] = 0;

cout<<"Input the vertices and their capacities :\n";

for (int i = 0; i < m; i++)

{

cin >> u >> v >> capacity;

graph[u][v] = capacity;

}

cout<<"Input the source vertice 's' : ";

cin >> s;

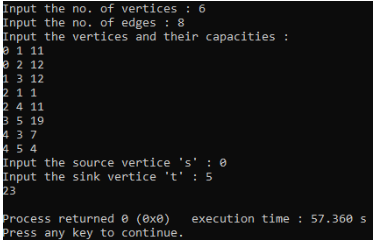
cout<<"Input the sink vertice 't' : ";

cin >> t;

cout << fordFulkerson(s, t, n) << endl;

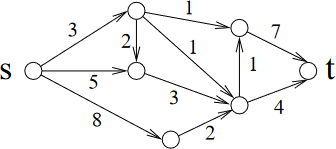
}

**Output :**



**Q2. Given a graph which represents a flow network where everyedge has a capacity. Also given two vertices *source* ‘s’ and *sink* ‘t’ in the graph, write a program to find the maximum possible flow from s to tusing Edmond’s Karp algorithm with following constraints:**

1. **Flowonanedgedoesn’texceedthegivencapacityofthe edge.**
2. **Incomingflowisequaltooutgoingflowforeveryvertexexceptsand t.**



#include <iostream>

#include <queue>

#include <climits>

using namespace std;

#define N 100

int graph[N][N], residualGraph[N][N];

int parent[N], visited[N];

int bfs(int s, int t, int n)

{

for (int i = 0; i < n; i++)

{

visited[i] = 0;

parent[i] = -1;

}

queue<int> q;

q.push(s);

visited[s] = 1;

while (!q.empty())

{

int u = q.front();

q.pop();

for (int v = 0; v < n; v++)

if (!visited[v] && residualGraph[u][v] > 0)

{

q.push(v);

parent[v] = u;

visited[v] = 1;

if (v == t)

return 1;

}

}

return 0;

}

int edmondsKarp(int s, int t, int n)

{

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

residualGraph[i][j] = graph[i][j];

int maxFlow = 0;

while (bfs(s, t, n))

{

int path\_flow = INT\_MAX;

for (int v = t; v != s; v = parent[v])

{

int u = parent[v];

if (residualGraph[u][v] < path\_flow)

path\_flow = residualGraph[u][v];

}

for (int v = t; v != s; v = parent[v])

{

int u = parent[v];

residualGraph[u][v] -= path\_flow;

residualGraph[v][u] += path\_flow;

}

maxFlow += path\_flow;

}

return maxFlow;

}

int main()

{

int n, m, u, v, capacity, s, t;

cout<<"Input the no. of vertices : ";

cin >> n;

cout<<"Input the no. of edges : ";

cin >> m;

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

graph[i][j] = 0;

cout<<"Input the vertices and their capacities :\n";

for (int i = 0; i < m; i++)

{

cin >> u >> v >> capacity;

graph[u][v] = capacity;

}

cout<<"Input the source vertice 's' : ";

cin >> s;

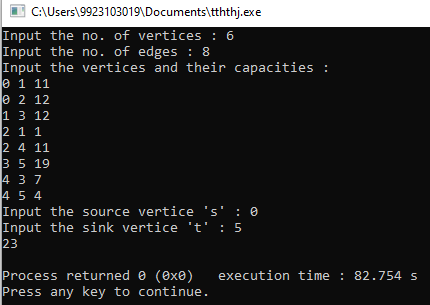
cout<<"Input the sink vertice 't' : ";

cin >> t;

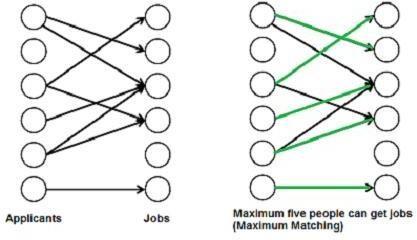
cout << edmondsKarp(s, t, n) << endl;

}

**Output :**

****

**Q*3.* There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Implement an efficient algorithm to find an assignment of jobs to applicants in such that as many applicants as possible get jobs.**



#include <iostream>

using namespace std;

#define M 100

#define N 100

int applicants, jobs;

int graph[M][N], jobAssigned[N], visited[M];

int bpm(int u)

{

for (int v = 0; v < jobs; v++)

{

if (graph[u][v] && !visited[v])

{

visited[v] = 1;

if (jobAssigned[v] == -1 || bpm(jobAssigned[v]))

{

jobAssigned[v] = u;

return 1;

}

}

}

return 0;

}

int maxBipartiteMatching()

{

for (int i = 0; i < jobs; i++)

jobAssigned[i] = -1;

int maxMatches = 0;

for (int i = 0; i < applicants; i++)

{

for (int j = 0; j < jobs; j++)

visited[j] = 0;

if (bpm(i))

maxMatches++;

}

return maxMatches;

}

int main()

{

cout<<"Input the number of applicants : ";

cin >> applicants;

cout<<"Input the number of jobs : ";

cin >> jobs;

for (int i = 0; i < applicants; i++)

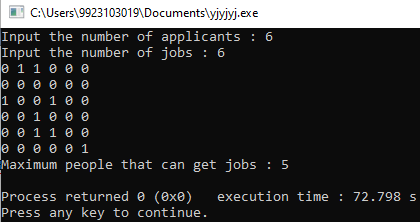
for (int j = 0; j < jobs; j++)

cin >> graph[i][j];

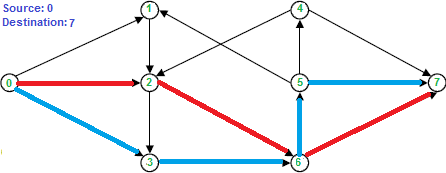
cout << maxBipartiteMatching() << endl;

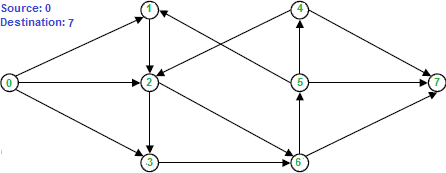
}

**Output :**



**Q4. Given a directed graph (representing a network) and two vertices in it, source ‘s’ and destination‘t’.Suppose you wantto sendsomelargefiles from stot butneverhavetwo filesuse the same network link (to avoid congestion on the links). Implement an algorithm to find out the maximum number of files that can be sent from s to t.**

**For example, for the given network, maximum two files can be sent via paths shown in blue and red.**

#include <iostream>

#include <cstring>

using namespace std;

#define N 100

int graph[N][N], residualGraph[N][N], parent[N];

bool visited[N];

bool dfs(int u, int t, int n)

{

if (u == t) return true;

visited[u] = true;

for (int v = 0; v < n; v++)

{

if (!visited[v] && residualGraph[u][v] > 0)

{

parent[v] = u;

if (dfs(v, t, n)) return true;

}

}

return false;

}

int fordFulkerson(int s, int t, int n)

{

memcpy(residualGraph, graph, sizeof(graph));

int maxFlow = 0;

while (true)

{

memset(visited, false, sizeof(visited));

memset(parent, -1, sizeof(parent));

if (!dfs(s, t, n)) break;

int pathFlow = 1;

for (int v = t; v != s; v = parent[v])

{

int u = parent[v];

residualGraph[u][v] -= pathFlow;

residualGraph[v][u] += pathFlow;

}

maxFlow++;

}

return maxFlow;

}

int main()

{

int n, m, u, v, s, t;

cout<<"Input the number of vertices : ";

cin >> n;

cout<<"Input the number of directed edges : ";

cin >> m;

cout<<"Input the source : ";

cin >> s;

cout<<"Input the destination : ";

cin >> t;

memset(graph, 0, sizeof(graph));

cout<<"Input the vertices connected : ";

for (int i = 0; i < m; i++)

{

cin >> u >> v;

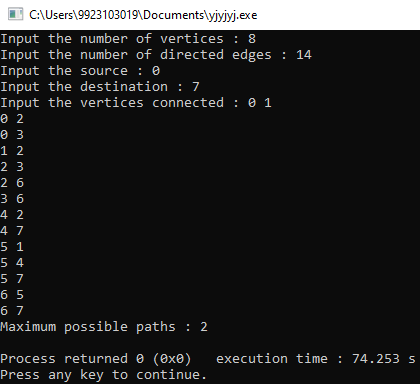
graph[u][v] = 1;

}

cout << "Maximum possible paths : " << fordFulkerson(s, t, n) << endl;

}

**Output :**

****