***NAME : Himanshu Dixit***

***ENROLL NO. : 21103262***

***BATCH : B11***

***Algorithms and Problem Solving [15B17CI471]***

***Lab - Week 8***

**Solution 1)**

#include <iostream>

#include <limits.h>

#include <queue>

#include <string.h>

using namespace std;

#define V 6

bool bfs(int rGraph[V][V], int s, int t, int parent[])

{

    bool visited[V];

    memset(visited, 0, sizeof(visited));

    queue<int> q;

    q.push(s);

    visited[s] = true;

    parent[s] = -1;

    while (!q.empty())

    {

        int u = q.front();

        q.pop();

        for (int v = 0; v < V; v++)

        {

            if (visited[v] == false && rGraph[u][v] > 0)

            {

                if (v == t)

                {

                    parent[v] = u;

                    return true;

                }

                q.push(v);

                parent[v] = u;

                visited[v] = true;

            }

        }

    }

    return false;

}

int fordFulkerson(int graph[V][V], int s, int t)

{

    int u, v;

    int rGraph[V]

              [V]; // Residual graph where rGraph[i][j]

    for (u = 0; u < V; u++)

        for (v = 0; v < V; v++)

            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to

    // store path

    int max\_flow = 0; // There is no flow initially

    while (bfs(rGraph, s, t, parent))

    {

        int path\_flow = INT\_MAX;

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

        {

            u = parent[v];

            path\_flow = min(path\_flow, rGraph[u][v]);

        }

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

        {

            u = parent[v];

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

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

        }

        max\_flow += path\_flow;

    }

    return max\_flow;

}

int main()

{

    int graph[V][V] = {{0, 16, 13, 0, 0, 0}, {0, 0, 10, 12, 0, 0}, {0, 4, 0, 0, 14, 0}, {0, 0, 9, 0, 0, 20}, {0, 0, 0, 7, 0, 4}, {0, 0, 0, 0, 0, 0}};

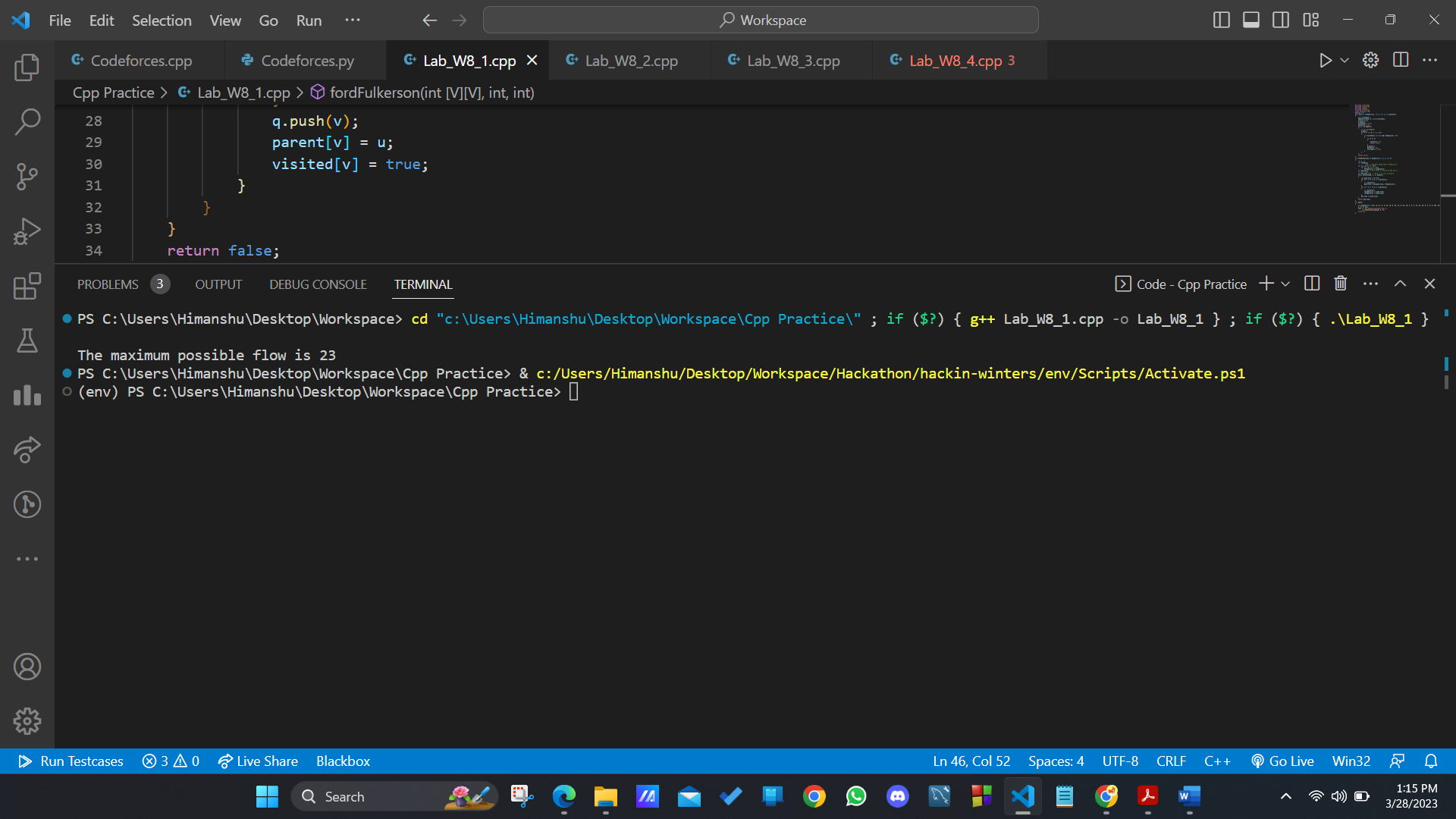
    cout << "The maximum possible flow is "

         << fordFulkerson(graph, 0, 5);

    return 0;

}

**Output :**



**Solution 2)**

#include <cstdio>

#include <queue>

#include <cstring>

#include <vector>

#include <iostream>

using namespace std;

int c[10][10];

int flowPassed[10][10];

vector<int> g[10];

int parList[10];

int currentPathC[10];

int bfs(int sNode, int eNode) // breadth first search

{

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

    memset(currentPathC, 0, sizeof(currentPathC));

    queue<int> q; // declare queue vector

    q.push(sNode);

    parList[sNode] = -1;       // initialize parlist’s source node

    currentPathC[sNode] = 999; // initialize currentpath’s source node

    while (!q.empty())         // if q is not empty

    {

        int currNode = q.front();

        q.pop();

        for (int i = 0; i < g[currNode].size(); i++)

        {

            int to = g[currNode][i];

            if (parList[to] == -1)

            {

                if (c[currNode][to] - flowPassed[currNode][to] > 0)

                {

                    parList[to] = currNode;

                    currentPathC[to] = min(currentPathC[currNode],

                                           c[currNode][to] - flowPassed[currNode][to]);

                    if (to == eNode)

                    {

                        return currentPathC[eNode];

                    }

                    q.push(to);

                }

            }

        }

    }

    return 0;

}

int edmondsKarp(int sNode, int eNode)

{

    int maxFlow = 0;

    while (true)

    {

        int flow = bfs(sNode, eNode);

        if (flow == 0)

        {

            break;

        }

        maxFlow += flow;

        int currNode = eNode;

        while (currNode != sNode)

        {

            int prevNode = parList[currNode];

            flowPassed[prevNode][currNode] += flow;

            flowPassed[currNode][prevNode] -= flow;

            currNode = prevNode;

        }

    }

    return maxFlow;

}

int main()

{

    int nodCount, edCount;

    cout << "enter the number of nodes and edges\n";

    cin >> nodCount >> edCount;

    int source, sink;

    cout << "enter the source and sink\n";

    cin >> source >> sink;

    for (int ed = 0; ed < edCount; ed++)

    {

        cout << "enter the start and end vertex along with capacity\n";

        int from, to, cap;

        cin >> from >> to >> cap;

        c[from][to] = cap;

        g[from].push\_back(to);

        g[to].push\_back(from);

    }

    int maxFlow =

        edmondsKarp(source, sink);

    cout << endl

         << endl

         << "Max Flow is:" << maxFlow << endl;

}

**Solution 3)**

#include <iostream>

#include <string.h>

using namespace std;

#define M 6

#define N 6

bool bpm(bool bpGraph[M][N], int u,

         bool seen[], int matchR[])

{

    for (int v = 0; v < N; v++)

    {

        if (bpGraph[u][v] && !seen[v])

        {

            seen[v] = true;

            if (matchR[v] < 0 || bpm(bpGraph, matchR[v],

                                     seen, matchR))

            {

                matchR[v] = u;

                return true;

            }

        }

    }

    return false;

}

int maxBPM(bool bpGraph[M][N])

{

    int matchR[N];

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

    int result = 0;

    for (int u = 0; u < M; u++)

    {

        bool seen[N];

        memset(seen, 0, sizeof(seen));

        if (bpm(bpGraph, u, seen, matchR))

            result++;

    }

    return result;

}

// Driver Code

int main()

{

    bool bpGraph[M][N] = {{0, 1, 1, 0, 0, 0},

                          {1, 0, 0, 1, 0, 0},

                          {0, 0, 1, 0, 0, 0},

                          {0, 0, 1, 1, 0, 0},

                          {0, 0, 0, 0, 0, 0},

                          {0, 0, 0, 0, 0, 1}};

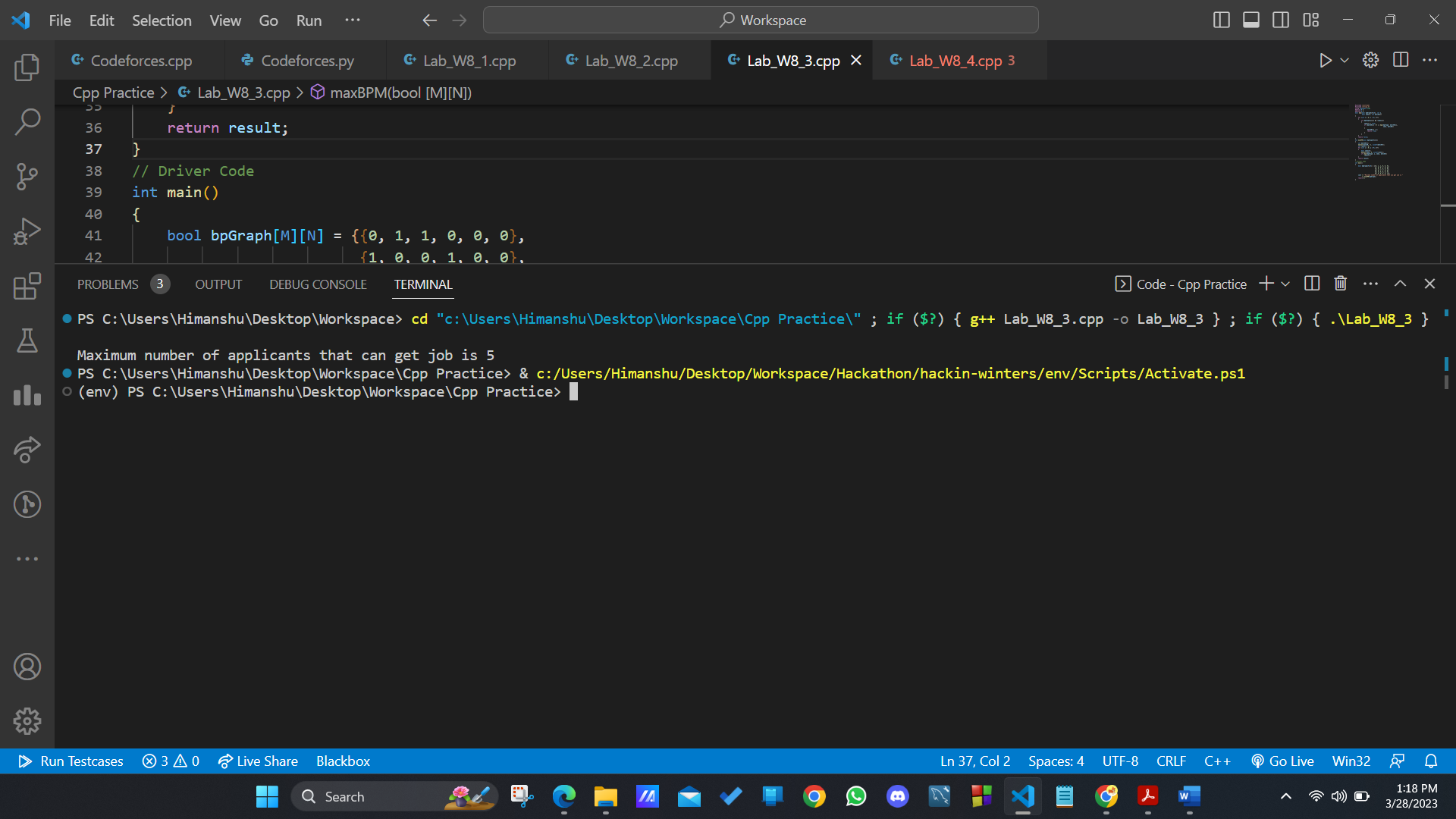
    cout << "Maximum number of applicants that can get job is "

         << maxBPM(bpGraph);

    return 0;

}

**Output :**



**Solution 4)**

The problem you described is known as the edge-disjoint paths problem, which can be solved using the Ford-Fulkerson algorithm for maximum flow. Here's one way to apply it to your problem:

1. Initialize the flow f(e) = 0 for each edge e in the graph.

2. While there exists an augmenting path P from s to t in the residual graphGf:

a. Find the minimum residual capacity c\_f(P) of the edges on path P.

b. Update the flow f(e) = f(e) + c\_f(P) for each edge e on path P.

c. Update the residual graph Gf by subtracting c\_f(P) from the residual capacity of each edge on P and adding c\_f(P) to the residual capacity of the corresponding reverse edges.

3. The maximum number of edge-disjoint paths from s to t is equal to the maximum flow value found by the algorithm.

Note that in your case, the capacity of each edge is 1 (since you can only send one file at a time on each link). Also, the residual graph Gf can be constructed using the residual capacities of the edges as follows:

1. If f(e) < c(e), add the edge (u,v) to Gf with residual capacity c\_f(u,v) = c(u,v) - f(u,v).

2. If f(e) > 0, add the reverse edge (v,u) to Gf with residual capacity c\_f(v,u) = f(u,v).

3. Remove any edge with zero residual capacity from Gf.

Once you have the maximum flow value, you can find the corresponding edge-disjoint paths by performing a depth-first search on the residual graph from s to t, following edges with positive residual capacity. Each path found in this way corresponds to a file transfer that uses distinct network links.