Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Name: Apurba Koirala

Reg no: 22BCE3799

Subject Code: BCSE204P

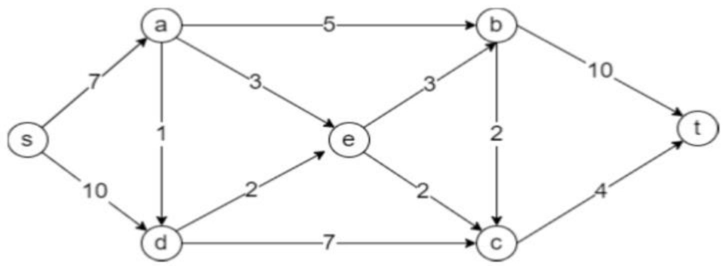
Course Title: Design and Analysis of Algorithms Lab

Lab Slot: L39 + L40

Guided by: Dr. IYAPPAN P

Lab Assessment 4

1. Determine the maximum flow possible in the given flow network using Ford Fulkerson method. Write the algorithm and analyse its time complexity.




Algorithm:

1. Function BFS(rGraph, source, sink, parent):
   - Initialize visited array as false.
   - Create a queue, enqueue source, mark it visited.
   - While queue is not empty:
     - Dequeue a node and explore its neighbors.
     - If an unvisited neighbor has positive capacity, enqueue it, mark visited, and store parent.
   - Return true if sink is reached, otherwise false.

2. Function FordFulkerson(graph, source, sink):
   - Create a residual graph rGraph initialized as graph.
   - Initialize max_flow = 0.
   - While there exists an augmenting path (using BFS):
     - Find the minimum residual capacity along the path.
     - Update residual capacities along the path.
     - Add path flow to max_flow.
   - Return max_flow.

3. Main function:
   - Define the capacity graph as a 7x7 matrix.
   - Call FordFulkerson(graph, source=0, sink=6).
   - Print the maximum possible flow.

Code

```
#include <iostream>
#include <climits>
#include <queue>
#include <string.h>
using namespace std;
bool bfs(int rGraph[][7], int s, int t, int parent[]) {
  bool visited[7];
  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 < 7; v++) {
            if (!visited[v] && rGraph[u][v] > 0) {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }
    return visited[t];
}
int fordFulkerson(int graph[7][7], int s, int t) {
    int u, v;
    int rGraph[7][7];

    for (u = 0; u < 7; u++)
        for (v = 0; v < 7; v++)
            rGraph[u][v] = graph[u][v];

    int parent[7];
    int max_flow = 0;

    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[7][7] = { {0, 7, 0, 0, 10, 0, 0},
                        {0, 0, 5, 0, 1, 3, 0},
                        {0, 0, 0, 2, 0, 0, 10},
                        {0, 0, 0, 0, 0, 0, 4},
                        {0, 0, 0, 7, 0, 2, 0},
```

```
                    {0, 0, 3, 2, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0} };

    cout << "The maximum possible flow is " << fordFulkerson(graph, 0, 6)<<endl;
    return 0;
}
```

Input:

```
int graph[7][7] = { {0, 7, 0, 0, 10, 0, 0},
                    {0, 0, 5, 0, 1, 3, 0},
                    {0, 0, 0, 2, 0, 0, 10},
                    {0, 0, 0, 0, 0, 0, 4},
                    {0, 0, 0, 7, 0, 2, 0},
                    {0, 0, 3, 2, 0, 0, 0},
                    {0, 0, 0, 0, 0, 0, 0} };
```

Output:

```
(base) apurbakoirala@Apurbas-MacBook-Pro ~ % /Users/apurbakoirala/Documents/VITw
ork/6th\ Sem/Design\ and\ Analysis\ of\ Algorithms\ Iyappan/Lab/Assesment4/Ford-
Fulkerson ; exit;
The maximum flow: 12
```

Time Complexity analysis

ford-fulkerson time complexity.

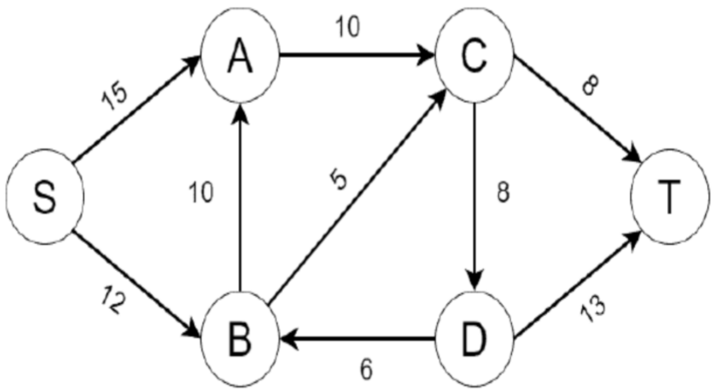
number of augmenting paths is at most $O(f)$

aug-path found using BFS, $O(E)$

Overall time complexity:

$O(fE)$

2. Determine the maximum flow possible in the given flow network using Edmond karp method. Write the algorithm and analyse its time complexity




Algorithm:
Function EdmondsKarp(graph, source, sink):
    maxFlow = 0
    While there exists a path from source to sink in residualGraph using BFS:
        pathFlow = min(residual capacity along the path)
        For each edge (u, v) in the found path:
            residualGraph[u][v] -= pathFlow
            residualGraph[v][u] += pathFlow
        maxFlow += pathFlow
    Return maxflow

Code:

```
#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)
{
  memset(parList, -1, sizeof(parList));
```

```
    memset(currentPathC, 0, sizeof(currentPathC));
    queue<int> q;
    q.push(sNode);
    parList[sNode] = -1;
    currentPathC[sNode] = 999;
    while(!q.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;
}
```

Input

```
enter the number of nodes and edges
6 9
enter the source and sink
0 5
enter the start and end vertex along with capacity
0 1 15
enter the start and end vertex along with capacity
0 2 12
enter the start and end vertex along with capacity
1 3 10
enter the start and end vertex along with capacity
2 1 10
enter the start and end vertex along with capacity
2 3 5
enter the start and end vertex along with capacity
3 4 8
enter the start and end vertex along with capacity
3 5 8
enter the start and end vertex along with capacity
4 2 6
enter the start and end vertex along with capacity
4 5 13
```

Output

```
Max Flow is:15

Saving session...
```

Time complexity analysis

Edmond Karp time complexity analysis.

Using adjacency Matrix, BFS takes $O(V+E)$

each edge's capacity is reduced at most $O(V)$ times.

BFS is executed at most $O(VE)$ times.

$$O(VE) \times O(V+E) = O(VE^2)$$

as $V \leq E$ is a connected graph.

3. **Design an algorithm for** computing the convex hull of a set of points given in a two dimensional plane using the following algorithms and analyse its time complexity.
   a. Graham Scan Algorithm
   b. Jarvis March Algorithm or Gift Wrapping

Graham Scan

Algorithm

```
FUNCTION convexHullGS(points[], n):
    Find the point with the smallest y-coordinate (leftmost if tie), set as p0
    Sort points by polar angle with p0 (using orientation and distance)

    Initialize an empty stack S
    Push first three points onto S

    FOR i FROM 3 TO n-1:
        WHILE orientation(nextToTop(S), S.top(), points[i]) != counterclockwise:
            Pop from S
        Push points[i] onto S
```

WHILE S is not empty:
    Print and pop points from S

Code:

```
#include <iostream>
#include <stack>
#include <stdlib.h>
using namespace std;

struct Point {
    int x;
    int y;
};

Point p0;

Point nextToTop(stack<Point> &S) {
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    return res;
}

void swap(Point &p1, Point &p2) {
    Point temp = p1;
    p1 = p2;
    p2 = temp;
}

int dist(Point p1, Point p2) {
    return (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
}

int orientation(Point p, Point q, Point r) {
    int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    if (val == 0) return 0;
    return (val > 0) ? 1 : 2;
}

int compare(const void *vp1, const void *vp2) {
    Point *p1 = (Point *) vp1;
    Point *p2 = (Point *) vp2;
    int o = orientation(p0, *p1, *p2);
    if (o == 0)
        return (dist(p0, *p2) >= dist(p0, *p1)) ? -1 : 1;
```

```
    return (o == 2) ? -1 : 1;
}

void convexHull(Point points[], int n) {
    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++) {
        int y = points[i].y;
        if ((y < ymin) || (ymin == y && points[i].x < points[min].x))
            ymin = points[i].y, min = i;
    }
    swap(points[0], points[min]);
    p0 = points[0];
    qsort(&points[1], n - 1, sizeof(Point), compare);

    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);

    for (int i = 3; i < n; i++) {
        while (orientation(nextToTop(S), S.top(), points[i]) != 2)
            S.pop();
        S.push(points[i]);
    }

    while (!S.empty()) {
        Point p = S.top();
        cout << "(" << p.x << ", " << p.y << ")" << endl;
        S.pop();
    }
}

int main() {
    Point points[] = { { 0, 3 }, { 1, 1 }, { 2, 2 }, { 4, 4 }, { 0, 0 },
                       { 1, 2 }, { 3, 1 }, { 3, 3 } };
    int n = sizeof(points) / sizeof(points[0]);
    cout << "The points in the convex hull are: \n";
    convexHull(points, n);
    return 0;
}
```

Input:

```
Point points[] = { { 0, 3 }, { 1, 1 }, { 2, 2 }, { 4, 4 }, { 0, 0 },
                   { 1, 2 }, { 3, 1 }, { 3, 3 } };
```

Output:

```
ork/6th\ Sem/Design\ and\ Analysis\ of\ Algorithms\ Iyappan/Lab/Assesment4/GS ;
exit;
The points in the convex hull are:
(0, 3)
(4, 4)
(3, 1)
(0, 0)

Saving session...
```

Time complexity Analysis:

Time complexity Analysis of Graham's Scan.

Lowest Y coordinate finding : check each point's y-coordinate
$O(n)$

Sorting Points by Polar angle, sorting T.C = $O(n \log n)$.

Constructing Convex Hull : stack used, points pushed & poped at most once.

$O(n)$

Overall T.C = $O(n) + O(n \log n) + O(n) =$

$O(n \log n)$ //

b. Jarvis March

Algorithm
FUNCTION orientation(p, q, r):
    val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y)
    IF val == 0:
        RETURN 0
    ELSE IF val > 0:

```
        RETURN 1
    ELSE:
        RETURN 2

FUNCTION convexHull(points[], n):
    IF n < 3:
        RETURN

    INITIALIZE hull
    FIND leftmost point l
    p = l

    REPEAT:
        ADD points[p] to hull
        q = (p + 1) % n

        FOR i = 0 TO n-1:
            IF orientation(points[p], points[i], points[q]) == 2:
                q = i

        p = q

    UNTIL p == l

    FOR each point in hull:
        PRINT point
```

Code

```
#include <iostream>
#include <vector>
using namespace std;

struct Point {
    int x, y;
};

int orientation(Point p, Point q, Point r) {
    int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    if (val == 0) return 0;
    return (val > 0) ? 1 : 2;
}

void convexHull(Point points[], int n) {
    if (n < 3) return;
    vector<Point> hull;
    int l = 0;
    for (int i = 1; i < n; i++)
```

```
        if (points[i].x < points[l].x)
            l = i;
    int p = l, q;
    do {
        hull.push_back(points[p]);
        q = (p + 1) % n;
        for (int i = 0; i < n; i++)
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;
        p = q;
    } while (p != l);
    for (int i = 0; i < hull.size(); i++)
        cout << "(" << hull[i].x << ", " << hull[i].y << ")\n";
}

int main() {
    Point points[] = { { 0, 3 }, { 1, 1 }, { 2, 2 }, { 4, 4 }, { 0, 0 },
                      { 1, 2 }, { 3, 1 }, { 3, 3 } };
    int n = sizeof(points) / sizeof(points[0]);
    convexHull(points, n);
    return 0;
}
```

Input

```
    Point points[] = { { 0, 3 }, { 1, 1 }, { 2, 2 }, { 4, 4 }, { 0, 0 },
                      { 1, 2 }, { 3, 1 }, { 3, 3 } };
```

Output

```
exit;
(0, 3)
(0, 0)
(3, 1)
(4, 4)
```

Time complexity analysis

Jarvis March.

Finding Leftmost Point.

Time complexity: $O(n)$, like Graham Scan

Constructing Convex Hull

Algorithm iterates h time for h points in convex hull,

per iteration complexity $O(n)$

Total complexity: $O(nh)$

Overall Time complexity: $O(n) + O(nh) = O(nh)$