LAB MANUAL

Applied Graph Theory -Lab

[For 7th Semester, B.Tech MCE]

Paper Code: MC-404 Title:Applied Graph Theory

Department of Applied Mathematics
Delhi Technological University
Bawana Road, Delhi - 42

APPLIED GRAPH THEORY LAB (MC – 404)

L	T	P
0	0	2

Lab based on the paper MC – 404

Evaluation		Total Marks	Credit Type
Sess.	End		
30	70	100	2C

Applied Graph Theory Laboratory Manual Contents

Applied Graph Theory Lab based on C++ programming

[For 7th Semester, B.Tech (MCE)]

Paper Code: MC-404 **Subject: Applied Graph Theory Lab**

Objective: Graph Theory Lab introduces the students learn some fundamental concepts and Various graphs algorithms and implement them in the lab using C++ so that they will be able to apply in real life problems.

List of Experiments

A
1. **Program to find the number of vertices, even vertices, odd vertices and number of edges in a graph.**
2. **Program to find union, intersection and ring-sum of two graphs.**

B
3. **Program to find minimal spanning tree of a graph using Prim's Algorithm.**
4. **Program to find minimal spanning tree of a graph using Kruskal's Algorithm.**

C
5. **Program to find shortest path between two vertices in a graph using Disjkstra's Algorithm.**
6. **Program to find shortest path between every pair of vertices in a graph using Floyd-Warshall's Algorithm.**
7. **Program to solve find shortest path between two vertices in a graph using Bellman-Ford's Algorithm.**

D
8. **Program to find maximum matching for bipartite graph.**
9. **Program to find maximum matching for general graph.**

E
10. **Program to maximum flow from source node to sink node using Ford-Fulkerson Algorithm.**

Software to be used: DEV C++

Ford–Fulkerson algorithm

The **Ford–Fulkerson method** (named for L. R. Ford, Jr. and D. R. Fulkerson) is an algorithm which computes the maximum flow in a flow network. It was published in 1956. The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a specialization of Ford–Fulkerson.

The idea behind the algorithm is simple. As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

Algorithm

Let $G(V, E)$ be a graph, and for each edge from u to v, let $c(u, v)$ be the capacity and $f(u, v)$ be the flow. We want to find the maximum flow from the source s to the sink t. After every step in the algorithm the following is maintained:

Capacity constraints:	$\forall (u, v) \in E \; f(u, v) \leq c(u, v)$	The flow along an edge can not exceed its capacity.
Skew symmetry:	$\forall (u, v) \in E \; f(u, v) = -f(v, u)$	The net flow from u to v must be the opposite of the net flow from v to u (see example).
Flow conservation:	$\forall u \in V : u \neq s$ and $u \neq t \Rightarrow \sum_{w \in V} f(u, w) = 0$	That is, unless u is s or t. The net flow to a node is zero, except for the source, which "produces" flow, and the sink, which "consumes" flow.

This means that the flow through the network is a *legal flow* after each round in the algorithm. We define the **residual network** $G_f(V, E_f)$ to be the network with capacity $c_f(u, v) = c(u, v) - f(u, v)$ and no flow. Notice that it can happen that a flow from v to u is allowed in the residual network, though disallowed in the original network: if $f(u, v) > 0$ and $c(v, u) = 0$ then $c_f(v, u) = c(v, u) - f(v, u) = f(u, v) > 0$.

Algorithm Ford–Fulkerson

Inputs Graph G with flow capacity c, a source node s, and a sink node t

Output A flow f from s to t which is a maximum

1. $f(u, v) \leftarrow 0$ for all edges (u, v)
2. While there is a path p from s to t in G_f, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$:
   1. Find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
   2. For each edge $(u, v) \in p$
      1. $f(u, v) \leftarrow f(u, v) + c_f(p)$ (*Send flow along the path*)
      2. $f(v, u) \leftarrow f(v, u) - c_f(p)$ (*The flow might be "returned" later*)

The path in step 2 can be found with for example a breadth-first search or a depth-first search in $G_f(V, E_f)$. If you use the former, the algorithm is called Edmonds–Karp.

When no more paths in step 2 can be found, s will not be able to reach t in the residual network. If S is the set of nodes reachable by s in the residual network, then the total capacity in the original network of edges from S to the remainder of V is on the one hand equal to the total flow we found from s to t, and on the other hand serves as an upper bound for all such flows. This proves that the flow we found is maximal. See also Max-flow Min-cut theorem.

If the graph $G(V, E)$ has multi Sources and Sinks, we act as follows. Suppose that $T = \{t | t \; is \; a \; sink\}$ and $S = \{s | s \; is \; a \; source\}$. Add a new source s^* with an edge (s^*, s) from s^* to every node $s \in S$, with capacity $f(s^*, s) = d_s$ $(d_s = \sum_{(s,u) \in E} f(s, u))$. And add a new sink t^* with an edge (t^*, t) from t^* to every

node $t \in T$, with capacity $f(t^*, t) = d_t$ ($d_t = \sum_{(v,t)\in E} f(v,t)$). Then applying the Ford–Fulkerson algorithm.

Also if every nodes u has constraint d_u, we replace this node with two nodes u_{in}, u_{out}, and an edge (u_{in}, u_{out}), with capacity $f(u_{in}, u_{out}) = d_u$. and then applying the Ford–Fulkerson algorithm.

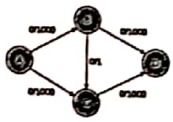
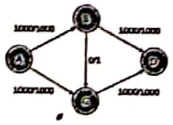
Complexity

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. However, there is no certainty that this situation will ever be reached, so the best that can be guaranteed is that the answer will be correct if the algorithm terminates. In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values. When the capacities are integers, the runtime of Ford-Fulkerson is bounded by $O(Ef)$ (see big O notation), where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in $O(E)$ time and increases the flow by an integer amount which is at least 1.

A variation of the Ford–Fulkerson algorithm with guaranteed termination and a runtime independent of the maximum flow value is the Edmonds–Karp algorithm, which runs in $O(VE^2)$ time.

Integral example

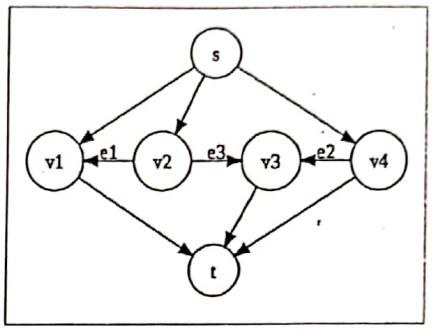
The following example shows the first steps of Ford–Fulkerson in a flow network with 4 nodes, source A and sink D. This example shows the worst-case behaviour of the algorithm. In each step, only a flow of 1 is sent across the network. If breadth-first-search were used instead, only two steps would be needed.

Path	Capacity	Resulting flow network
	Initial flow network	
A, B, C, D	$\min(c_f(A,B), c_f(B,C), c_f(C,D)) =$ $\min(c(A,B) - f(A,B), c(B,C) - f(B,C), c(C,D) - f(C,D)) =$ $\min(1000 - 0, 1 - 0, 1000 - 0) = 1$	
A, C, B, D	$\min(c_f(A,C), c_f(C,B), c_f(B,D)) =$ $\min(c(A,C) - f(A,C), c(C,B) - f(C,B), c(B,D) - f(B,D)) =$ $\min(1000 - 0, 0 - (-1), 1000 - 0) = 1$	
	After 1998 more steps ...	
	Final flow network	

Notice how flow is "pushed back" from C to B when finding the path A, C, B, D.

Non-terminating example

Consider the flow network shown on the right, with source s, sink t, capacities of edges e_1, e_2 and e_3 respectively 1, $r = (\sqrt{5}-1)/2$ and 1 and the capacity of all other edges some integer $M \geq 2$. The constant r was chosen so, that $r^2 = 1 - r$. We use augmenting paths according to the following table, where $p_1 = \{s, v_4, v_3, v_2, v_1, t\}$, $p_2 = \{s, v_2, v_3, v_4, t\}$ and $p_3 = \{s, v_1, v_2, v_3, t\}$.



Step	Augmenting path	Sent flow	Residual capacities		
			e_1	e_2	e_3
0			$r^0 = 1$	r	1
1	$\{s, v_2, v_3, t\}$	1	r^0	r^1	0
2	p_1	r^1	r^2	0	r^1
3	p_2	r^1	r^2	r^1	0
4	p_1	r^2	0	r^3	r^2
5	p_3	r^2	r^2	r^3	0

Note that after step 1 as well as after step 5, the residual capacities of edges e_1, e_2 and e_3 are in the form r^n, r^{n+1} and 0, respectively, for some $n \in \mathbb{N}$. This means that we can use augmenting paths p_1, p_2, p_1 and p_3 infinitely many times and residual capacities of these edges will always be in the same form. Total flow in the network after step 5 is $1 + 2(r^1 + r^2)$. If we continue to use augmenting paths as above, the total flow converges to $1 + 2\sum_{i=1}^{\infty} r^i = 3 + 2r$, while the maximum flow is $2M + 1$. In this case, the algorithm never terminates and the flow doesn't even converge to the maximum flow.

Python implementation

```
class Edge(object):
    def __init__(self, u, v, w):
        self.source = u
        self.sink = v
        self.capacity = w
    def __repr__(self):
        return "%s->%s:%s" % (self.source, self.sink, self.capacity)

class FlowNetwork(object):
    def __init__(self):
        self.adj = {}
        self.flow = {}

    def add_vertex(self, vertex):
        self.adj[vertex] = []

    def get_edges(self, v):
```

```
        return self.adj[v]

    def add_edge(self, u, v, w=0):
        if u == v:
            raise ValueError("u == v")
        edge = Edge(u,v,w)
        redge = Edge(v,u,0)
        edge.redge = redge  #redge is not defined in Edge class
        redge.redge = edge
        self.adj[u].append(edge)
        self.adj[v].append(redge)
        self.flow[edge] = 0
        self.flow[redge] = 0

    def find_path(self, source, sink, path, path_set):
        if source == sink:
            return path
        for edge in self.get_edges(source):
            residual = edge.capacity - self.flow[edge]
            if residual > 0 and not (edge,residual) in path_set:
                path_set.add((edge, residual))
                result = self.find_path(edge.sink, sink, path +
[(edge,residual)], path_set)
                if result != None:
                    return result

    def max_flow(self, source, sink):
        path = self.find_path(source, sink, [], set())
        while path != None:
            flow = min(res for edge,res in path)
            for edge,res in path:
                self.flow[edge] += flow
                self.flow[edge.redge] -= flow
            path = self.find_path(source, sink, [], set())
        return sum(self.flow[edge] for edge in self.get_edges(source))
```

Usage example

For the example flow network in maximum flow problem we do the following:

```
>>> g = FlowNetwork()
>>> [g.add_vertex(v) for v in "sopqrt"]
[None, None, None, None, None, None]
>>> g.add_edge('s','o',3)
>>> g.add_edge('s','p',3)
>>> g.add_edge('o','p',2)
>>> g.add_edge('o','q',3)
>>> g.add_edge('p','r',2)
>>> g.add_edge('r','t',3)
```

2. Introduction to Applied Graph Theory

In mathematics and computer science, **graph theory** is the study of *graphs*, which are mathematical structures used to model pair wise relations between objects. A graph in this context is made up of *vertices* or *nodes* or *points* and *edges* or *arcs* or *lines* that connect them. A graph may be *undirected*, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be *directed* from one vertex to another. Graphs are one of the prime objects of study in discrete mathematics.

Graph theory is becoming increasingly significant as it is applied to other areas of mathematics, science and technology. It is being actively used in fields as varied as biochemistry (genomics), electrical engineering (communication networks and coding theory), computer science (algorithms and computation) and operations research (scheduling). The powerful combinatorial methods found in graph theory have also been used to prove fundamental results in other areas of pure mathematics.

The powerful combinatorial methods found in graph theory have also been used to prove significant and well-known results in a variety of areas in mathematics itself. The best known of these methods are related to a part of graph theory called matching, and the results from this area are used to prove Dilworth's chain decomposition theorem for finite partially ordered sets. An application of matching in graph theory shows that there is a common set of left and right coset representatives of a subgroup in a finite group. The existence of matching in certain infinite bipartite graphs played an important role in Laczkovich's affirmative answer to Tarski's 1925 problem of whether a circle is piecewise congruent to a square.

Some Typical Graph Applications

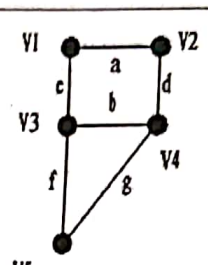
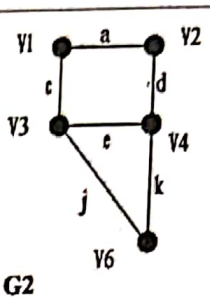
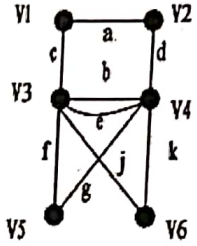
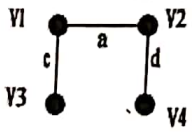
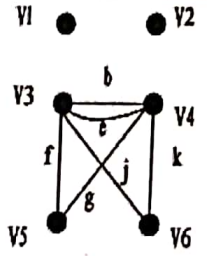
- Modeling a road network with vertexes as towns and edge costs as distances.
- Modeling a water supply network. A cost might relate to current or a function of capacity and length. As water flows in only 1 direction, from higher to lower pressure connections or downhill, such a network is inherently an acyclic directed graph.
- Modeling the recent contacts of someone who has become ill with a notifiable illness, e.g. Sars or Meningitis. Edge costs might be a function of the probability that the contact resulted in an infection.
- Dynamically modeling the status of a set of routes by which traffic might be directed over the Internet.
- Modeling the connections between a number of potential witnesses or suspects who were reported or came forward as having been within the vicinity of a serious crime within an hour of when it occurred.
- Minimizing the cost and time taken for air travel when direct flights don't exist between starting and ending airports.
- Using a directed graph to map the links between pages within a website and to analyze ease of navigation between different parts of the site.

3. Explanation of the concerned methods

1. Even vertices and odd vertices: If the degree of a vertex is even (odd) then the vertex is called an even (odd) vertex.

2. Union, intersection and ring-sum of two graphs:

Basic Operations:

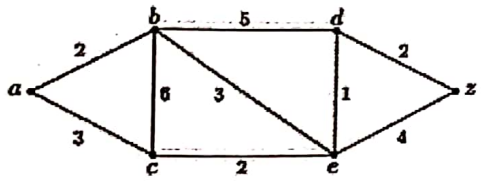
If G1 = (V1, E1) and G2 = (V2, E2), then:	 G1	 G2
Union G3 = (V3, E3) = G1 U G2 ⇔ V3 = V1 U V2 and E3 = E1 U E2.		G1 U G2
Intersection G4 = (V4, E4) = G1 I G2 ⇔ V4 = V1 I V2 and E4 = E1 I E2.		G1 I G2
Ring Sum G5 = (V5, E5) = G1 U G2 ⇔ V5 = V1 U V2 and E5 = (E1 U E2) – (E1 I E2)		G1 ⊕ G2

Prim's Algorithm

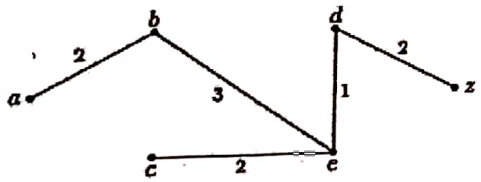
We consider a weighted connected graph G with n vertices. Prim's algorithm finds a minimum spanning tree of G.

procedure *Prim*(G: weighted connected graph with n vertices)
$T :=$ a minimum-weight edge
for $i = 1$ **to** $n - 2$
begin
  $e :=$ an edge of minimum weight incident to a vertex in T and not forming a circuit in T if added to T
  $T := T$ with e added
end
return(T)

Example: Use Prim's algorithm to find a minimum spanning tree in the following weighted graph. Use alphabetical order to break ties.



Solution: Prim's algorithm will proceed as follows. First we add edge $\{d, e\}$ of weight 1. Next, we add edge $\{c, e\}$ of weight 2. Next, we add edge $\{d, z\}$ of weight 2. Next, we add edge $\{b, e\}$ of weight 3. And finally, we add edge $\{a, b\}$ of weight 2. This produces a minimum spanning tree of weight 10. A minimum spanning tree is the following.



4. Kruskal's Algorithm

Let $G = (V, E)$ be the given graph, with $|V| = n$

{

Start with a graph $T = (V, \phi)$ consisting of only the

vertices of G and no edges; /* This can be viewed as n

connected components, each vertex being one connected component */

Arrange E in the order of increasing costs;

for $(i = 1, i \leq n - 1, i++)$

{ Select the next smallest cost edge;

if (the edge connects two different connected components)

add the edge to T;

}

}

5. Shortest Path between two vertices:

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

```
DIJKSTRA (G, w, s)
1. INITIALIZE SINGLE-SOURCE (G, s)
2. S ← { }    // S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., Q ← V[G]
4. while priority queue Q is not empty do
5.    u ← EXTRACT_MIN(Q)    // Pull out new vertex
6.    S ← S ∪ {u}
      // Perform relaxation for each vertex v adjacent to u
7.    for each vertex v in Adj[u] do
8.       Relax (u, v, w)
```

6. Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

7. Bellman–Ford algorithm

```
1  % implements: the SSSP problem
2  function BellmanFord(G=(V,E),s)=
3  let
4     % requires: all{D_v = δ^k_G(s,v) : v ∈ V}
5     function BF(D,k)=
6        let
7           D' = {v ↦ min(D_v, min_{u∈N⁻_G(v)}(D_u + w(u,v))) : v ∈ V}
8        in
9           if (k=|V|) then ⊥
10          else if (all{D_v = D'_v : v ∈ V}) then D
11          else BF(D', k+1)
12       end
13    D = {v ↦ if v=s then 0 else ∞ : v ∈ V}
14 in BF(D,0) end
```

8. Maximum Bipartite Matching

A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matching for a given Bipartite Graph.

Maximum Bipartite Matching Given a bipartite graph G = (A ∪ B, E), find an S ⊆ A × B that is a matching and is as large as possible.

Notes: • We're given A and B so we don't have to find them.

• S is a perfect matching if every vertex is matched.

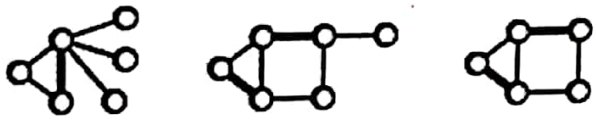
• Maximum is not the same as maximal: greedy will get to maximal.

9.Maximum Matching for general Graphs:

Given a graph $G = (V,E)$, a **matching** M in G is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.

A vertex is **matched** (or **saturated**) if it is an endpoint of one of the edges in the matching. Otherwise the vertex is **unmatched**.

A **maximal matching** is a matching M of a graph G with the property that if any edge not in M is added to M, it is no longer a matching, that is, M is maximal if it is not a proper subset of any other matching in graph G. In other words, a matching M of a graph G is maximal if every edge in G has a non-empty intersection with at least one edge in M. The following figure shows examples of maximal matchings (red) in three graphs.



A **maximum matching** (also known as maximum-cardinality matching[1]) is a matching that contains the largest possible number of edges. There may be many maximum matchings. The **matching number** $\nu(G)$ of a graph G is the size of a maximum matching. Note that every maximum matching is maximal, but not every maximal matching is a maximum matching. The following figure shows examples of maximum matchings in the same three graphs.