INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

APS Course Project

on

"Study and implementation of dynamic graph algorithms- Matching/Shortest Path/Transitive Closure"

Submitted By:

Shraddha

Roll No. : 2019201001

CONTENTS:

1.INTRODUCTION :

What is a Dynamic Graph?

In computing and graph theory, a **dynamic connectivity** structure is a data structure that dynamically maintains information about the connected components of a graph.

The set *V* of vertices of the graph is fixed, but the set *E* of edges can change. The three cases, in order of difficulty, are:

- Edges are only added to the graph (this can be called ***incremental connectivity***);
- Edges are only deleted from the graph (this can be called ***decremental connectivity***);
- Edges can be either added or deleted (this can be called ***fully dynamic connectivity***).

  Expectation: After each addition/deletion of an edge, the dynamic connectivity structure should adapt itself such that it can give quick answers to queries.

Operations/Queries/Algorithms on Dynamic Graphs:

- Matching Problem
- All Pair Shortest Path
- Transitive Closure
- Depth First Search

All these problems are attempted to be solved one by one in optimal possible complexity in this report.

2. MATCHING-PROBLEM:

2.1 Introduction to Problem:

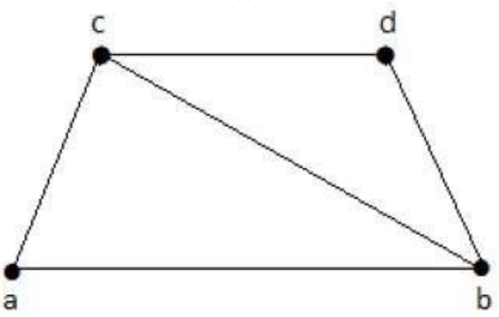
“Given a graph, a matching is a set of edges, such that no two edges share the same vertex. In other words, matching of a graph is a subgraph where each node of the subgraph has either zero or one edge incident to it.”

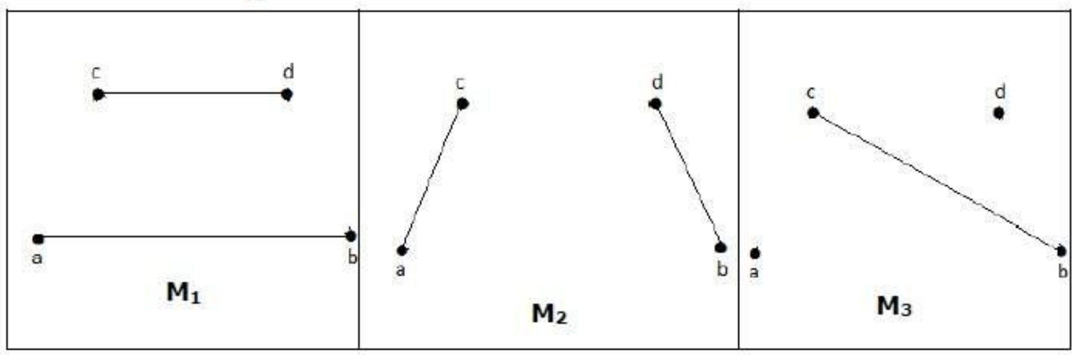
A vertex is said to be **matched** if an edge is incident to it, **free** otherwise.

Some more terminologies related with matching are:

- **Maximal Matching –** A matching M of graph G is said to be maximal if on adding an edge which is in G but not in M, makes M not a matching. In other words, a maximal matching M is not a proper subset of any other matching of G.

In the figure below M1, M2 and M3 denote the maximal matchings for given graph G.



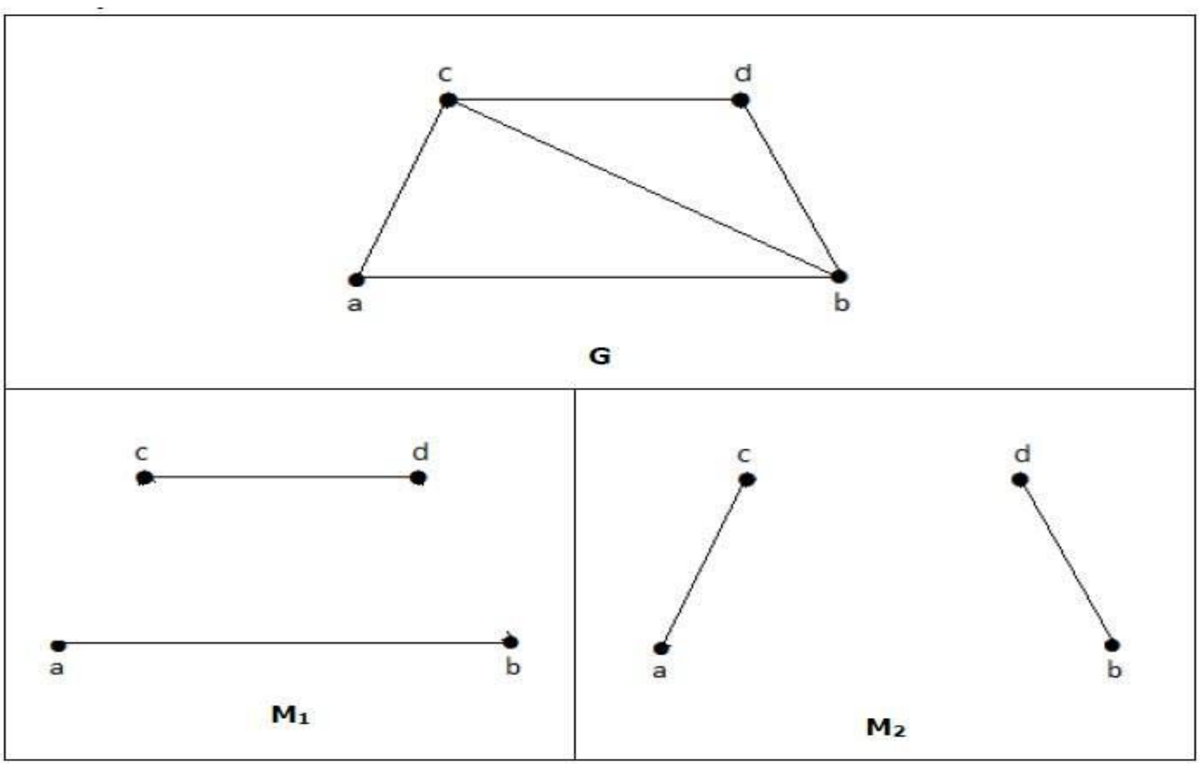


- **Maximum Matching –** A matching M of graph G is said to be maximum if it is maximal and has the maximum number of edges. There may be many possible maximum matchings of a graph. Every maximum matching is a maximal matching but not every maximal matching is a maximum matching.

- **Perfect Matching –** A matching M of graph G is said to be perfect if **every vertex** is connected to exactly one edge. Every perfect matching is a maximum matching but not every maximum matching is a perfect matching. Since every vertex has to be included in a perfect matching, the number of edges in the matching must be V/2 where V is the number of vertices. Therefore, a perfect matching only exists if the number of vertices is even.

A matching is said to be **near perfect** if the number of vertices in the original graph is odd, it is a maximum matching and it leaves out only one vertex.

In the figure below M1 and M2 are both maximum and perfect matching also.



2.2 Problem Statement:

In the dynamic maximum matching problem (DMM), a graph $G = (V, E)$ will be given as an initial input. The vertex set V will remain the same and the edge set E will be changed dynamically.

DMM must handle the following operations:

1. Update

(a) Insert (e) - Insert the edge e. ($E = E \cup \{e\}$)

(b) Delete (e) - Delete the edge e. ($E = E \setminus \{e\}$)

2. Query (q)

(a) Size - Returns the size of the maximum matching.

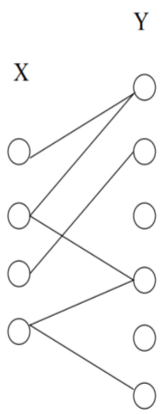
2.3 Approach to solve:

Approach 1: Solving the maximum bipartite matching using **Network Flow problem.**

Algorithm:

- Given bipartite graph G = (A ∪ B, E), direct the edges from A to B.
- Add new vertices s and t.
- Add an edge from s to every vertex in A.
- Add an edge from every vertex in B to t.
- Make all the capacities 1.
- Solve maximum network flow problem on this new graph G.

Result: The edges used in the maximum network flow will correspond to the largest possible matching!
This algorithm is popularly known as **Ford-Fulkerson** algorithm.



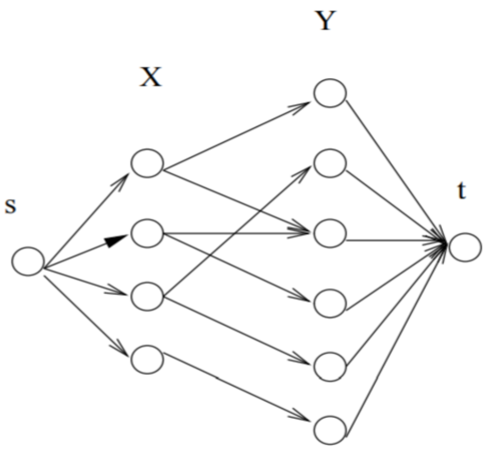


Fig 1

Fig 2

Fig 1: It depicts a bipartite graph whose matching is to be found.

Fig 2: It shows the intermediate step in solving the problem by flows problem.

2.4 Complexity Analysis of Algorithm:

- The running time of the above algorithm is $O(m'C)$ where m' is the number of edges, and C = Summation of all the edges leaving source i.e., $C = |A| = n$.
- The number of edges in G' is equal to number of edges in G (m) plus 2n.
- So, running time is $O((m + 2n)n) = (mn + n^2) = O(mn)$

Note : Assumptions made:

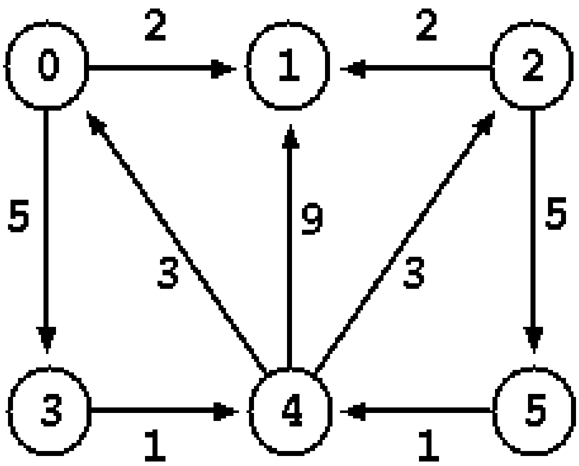
- Graph is bipartite
- Graph is unweighted or uniformly weighted

3. ALL PAIR SHORTEST PATH:

3.1 Introduction to Problem:

In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

- The **all-pairs shortest path** problem finds the shortest paths between every pair of vertices *v*, *v′* in the graph.
- One example illustrating the problem is shown below:



	0	1	2	3	4	5
0	0	2	0	5	0	0
1	0	0	0	0	0	0
2	0	2	0	0	0	5
3	0	0	0	0	1	0
4	3	9	3	0	0	0
5	0	0	0	0	1	0

Fig 3

Fig 3: Given a directed weighted graph, it shows the final result of all pair shortest path problem.

3.2 Problem Statement:

Given Input: A graph G (V, E) which allows dynamic updations (like insertion and deletion of edges with given weights).

Queries:
- Insertion of an edge.
- Deletion of an edge.
- Shortest path between given two vertices.
- Matrix showing all pair shortest paths.

3.3 Approach to solve:

3.3.1 Approach 1: By solving single source shortest path (popularly known as **Dijkstra's Algorithm**) on every vertex (popularly known as **Johnson's Algorithm**).

3.3.1.1 Algorithm:

- First, a new node *q* is added to the graph, connected by zero-weight edges to each of the other nodes.
- Second, the Bellman–Ford algorithm is used, starting from the new vertex *q*, to find for each vertex *v* the minimum weight $h(v)$ of a path from *q* to *v*. If this step detects a negative cycle, the algorithm is terminated.
- Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from *u* to *v*, having length , is given the new length $w(u,v) + h(u) - h(v)$.
- Finally, *q* is removed, and Dijkstra's algorithm is used to find the shortest paths from each node *s* to every other vertex in the reweighted graph.

3.3.1.2 Pseodo-code:

```
Johnson(G)
   1.
   create G` where G`.V = G.V + {s},
      G`.E = G.E + ((s, u) for u in G.V), Johnson(G)
   1.
   create G` where G`.V = G.V + {s},
      G`.E = G.E + ((s, u) for u in G.V), and
      weight(s, u) = 0 for u in G.V
   2.
   if Bellman-Ford(s) == False
      return "The input graph has a negative weight cycle"
   else:
```

```
    for vertex v in G`.V:
      h(v) = distance(s, v) computed by Bellman-Ford
    for edge (u, v) in G`.E:
      weight`(u, v) = weight(u, v) + h(u) - h(v)
  3.
    D = new matrix of distances initialized to infinity
    for vertex u in G.V:
      run Dijkstra(G, weight`, u) to compute distance`(u, v) for all v in G.V
      for each vertex v in G.V:
        D_(u, v) = distance`(u, v) + h(v) - h(u)
    return D
and
    weight(s, u) = 0 for u in G.V
  2.
  if Bellman-Ford(s) == False
    return "The input graph has a negative weight cycle"
  else:
    for vertex v in G`.V:
      h(v) = distance(s, v) computed by Johnson(G)
  1.
  create G` where G`.V = G.V + {s},
    G`.E = G.E + ((s, u) for u in G.V), and
    weight(s, u) = 0 for u in G.V
  2.
  if Bellman-Ford(s) == False
    return "The input graph has a negative weight cycle"
  else:
    for vertex v in G`.V:
      h(v) = distance(s, v) computed by Bellman-Ford
    for edge (u, v) in G`.E:
      weight`(u, v) = weight(u, v) + h(u) - h(v)
  3.
    D = new matrix of distances initialized to infinity
    for vertex u in G.V:
      run Dijkstra(G, weight`, u) to compute distance`(u, v) for all v in G.V
      for each vertex v in G.V:
```

```
            D_(u, v) = distance`(u, v) + h(v) - h(u)
    return DBellman-Ford
    for edge (u, v) in G`.E:
        weight`(u, v) = weight(u, v) + h(u) - h(v)
3.
    D = new matrix of distances initialized to infinity
    for vertex u in G.V:
        run Dijkstra(G, weight`, u) to compute distance`(u, v) for all v in G.V
        for each vertex v in G.V:
            D_(u, v) = distance`(u, v) + h(v) - h(u)
    return D
```

3.3.1.3 Complexity Analysis:

The main steps in algorithm are Bellman Ford Algorithm called once and Dijkstra's called V times. Time complexity of Bellman Ford is O(VE) and time complexity of Dijkstra's is O(VLogV). So overall time complexity is O $(V^2\log V + VE)$.

The time complexity of Johnson's algorithm becomes same as Floyd Warshell when the graphs is complete (For a complete graph E = O (V^2). But for sparse graphs, the algorithm performs much better than Floyd Warshell.

3.3.2 Approach 2: (Using dynamic dijkstra's)
By considering each edge of graph as intermediate between every pair of nodes to see if it minimizes the distance between the two.

3.3.2.1 Algorithm:

- For the case of adding an edge (u,v) - then using your already built-distance matrix - do the following :
For every pair of nodes x and y check if d((x,u))+c((u,v))+d((v,y))<d((x,y)) - this can be done in O(n^2) since you are comparing every pair of nodes.
- For the case of edge deletion: Given the distance matrix already built, then you can have for every node u a shortest-path tree rooted at u. If the deleted edge e is not in that tree, then the shortest paths from u to every other is not affected - (they remain the same).
- If e is in the shortest path tree of u, then for every node v such that the shortest path π(u,v) includes e, the paths will change. Therefore, compute the shortest path from u to v. Now, repeat the previous for every node -- this is not the best solution. In fact, in its worst case it is asymptotically equivalent to doing everything from scratch, but can be better on average.

3.3.2.2 Complexity Analysis:

As mentioned in the algorithm above itself the complexity of this algorithm ends up in O(n^2).

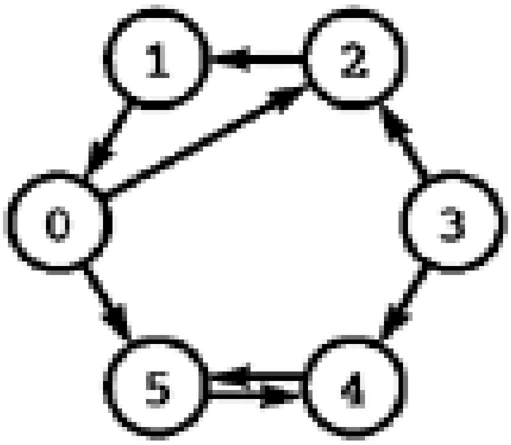
4.TRANSITIVE CLOSURE:

4.1 Introduction to Problem:

Formally, we define the transitive closure (TC) problem as follows. Given a directed graph G = (V, E) with |V | = n, |E| = m, we aim to output an n × n matrix where C(u, v) 6= 0 iff v is reachable from u. For the static version of the problem, there are two natural algorithms. Using depth-first search from every node, we can compute TC in O(mn) time.

In the dynamic version of transitive closure, we must maintain a directed graph G = (V, E) and support the operations of deleting or adding an edge and querying whether v is reachable from u as quickly as possible.

One example illustrating this problem is as shown below:



	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1

In this figure for every entry (u,v) in given reachability matrix; v is reachable from u if (u,v) is 1 else not reachable.

4.2 Problem Statement:

The algorithm for transitive closure must support following operations:

1. Update

(a) Insert (e) - Insert the edge e. ($E = E \cup \{e\}$)

(b) Delete (e) - Delete the edge e. ($E = E \setminus \{e\}$)

2. Query (u,v)

(a) Reachability - Returns true if v is reachable from u else returns false.

4.3 Approach to solve:

Approach 1: Using static approach i.e. solving insertion and deletion queries in O(1) time while solving query operations in O(n^2) time.

Algorithm:

1. Create a matrix tc[V][V] that would finally have transitive closure of given graph. Initialize all entries of tc[][] as 0.
2. Call DFS for every node of graph to mark reachable vertices in tc[][]. In recursive calls to DFS, we don't call DFS for an adjacent vertex if it is already marked as reachable in tc[][].

   But the complexity here results in O(n^2) for each query operation.
   The next approach gives O(1) complexity for query processing.

Approach 2: Using King and Sagert Approach.

The key idea behind King/Sagert's strategy is to maintain a full transitive closure matrix C and update it as necessary. Clearly, then the query time is constant. Notice however, that any approach that explicitly stores a transitive closure matrix cannot do better than Ω(n^ 2) time for updates. To see this, consider a graph consisting of an edge e = (u, v) and where v points to Ω(n) nodes B, and Ω(n) nodes A point to u. Here e connects Ω(n^2) pairs of nodes. Repeatedly removing and inserting e would correspond to updating Ω(n^2) entries of the TC matrix in each update. In this sense, King and Sagert's algorithm is optimal for algorithms that explicitly store a TC matrix.

Algorithm:

Let G be a DAG. We will maintain a matrix C where C(u, v) = the number of paths from u to v.

- insert(x, y): For all pairs (u, v), update C(u, v) ← C(u, v) + C(u, x) · C(y, u).
- delete(x, y): For all pairs (u, v), update C(u, v) ← C(u, v) − C(u, x) · C(y, u).
- query(x, y): Return Reachable iff C(u, v) 6= 0.

This algorithm gives an amortised complexity of O(1) for reachability queries.

REFERENCES:

https://en.wikipedia.org/wiki/Matching_(graph_theory)

https://en.wikipedia.org/wiki/Flow_network

https://en.wikipedia.org/wiki/Shortest_path_problem

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Leiserson, C. *CLRS*. Retrieved June 2, 2016, from http://citc.ui.ac.ir/zamani/clrs.pdf