# Circuit and path matrices in a graph

Create the circuit and path matrices from a given graph

Aneesh Epari (IWM2014502) Kshitij Tripathi (ISM2014501) Rishabh Verma (ICM2014004)

IIIT Allahabad

***Abstract*— This report describes about how to create path matrix and circuit matrix. It describes it both for directed and undirected graphs. It discussed about naïve robust and Johnson's algorithm for finding all the elementary circuits from the graph. This forms the cycle basis to form all possible cycles. For the second part that is, the path matrix it discusses on how to use transitive closure to find reachability among vertices using Warshall's algorithm and DFS based algorithm. The latter approach outperforms Warshall's algorithm.**

***Keywords*— Edges, vertices, Adjacency matrix, Directed graph, Undirected graph, Paths, Elementary Paths, Circuits, Elementary Circuits, Transitive Closure, strongly connected components, reachability, cycles, simple cycles.**

## I. INTRODUCTION

A Graph consists of vertices and edges. These vertices and edges hold certain mutual relations. In this paper we present an algorithm to effectively find the circuit and path matrices of a graph. For the first part i.e. find all the circuits in a graph, the major problems to be solved in producing an efficient algorithm [5] are: 1) to find all elementary circuits, 2) to avoid finding any circuit more than once, 3) to avoid wasting a great deal of effort in fruitless search for non-existent circuits. For the second part we need to devise an efficient algorithm to perform the transitive closure of a graph and in turn build the reachability matrix.

## II. MOTIVATION

Graph simulates all real world scenarios. Similarly study of circuits and paths is proficiently used in solving problems [4] like mail delivery, garbage pickup, bus service, traveling salesman etc. because these problems are NP-complete in nature means that the problem can be solved in Polynomial time using a Non-deterministic Turing machine [2]. The worst-case time complexity of our problem is exponential if given graph is a complete in nature. So, instead of exploring all possible circuits, we look for the elementary circuits which can be solved in polynomial time. The running time is higher if the graph is dense [2][4].

In this report, we tried to discuss various approaches related to finding circuit and path matrix from the given graph in optimal time.

## III. BACKGROUND AND METHODOLOGY

Before proceeding to any further, we explain the terminology used throughout this paper for clarity sake, it is important to define the graph theoretic structures with which we will be working.

**Definitions**:

1. A vertex set, V, is a set of vertices or points.
2. The edge set on V, $E_V$ is the set of all ordered pairs of vertices in V in the graph G.
3. A graph is an ordered pair (V, $E_V$) where V is the vertex set and $E_{V\ is}$ edge set on V.
4. Let G = (V, Ev) be a graph. Suppose a, b belongs to V. We define a path in G from a to b be a sequence of vertices a = v0, v1, v2, ….., v (n – 1), v(n) = b for which vi is in V, i = 0, 1, 2, ... n and for which (v (j – 1), v(j)) belong to Ev, for j = 1, 2, … n.
5. An elementary path in b from a to b is a path in G from a to b for which all the vi is distinct except v0 and vn.
6. Let G (V, Ev) be a graph. A circuit in G is a path in G form a to a, where a belongs to V.
7. Let G (V, Ev) be a graph. An elementary circuit in G is an elementary path in G form a to a, where a belongs to V.

The depth first search technique proves to be an excellent vehicle for solving problems 1) and 2). The depth first search is technique which has been employed in past in such places as garbage collection marking algorithms and has been used for producing highly efficient graph algorithms.

For the second task in this paper, we need to construct a path matrix for the graph which is essentially it's transitive closure. For any directed/ undirected graph, the transitive closure is graph G' such that (i, j) is an edge in G' if there is a directed path from i to j in G. The resultant G' representation in form of adjacency matrix is called the path matrix or the connectivity matrix.

**Algorithm – 1**

**Input**: A graph G = {V, E}
**Output**: The list of all simple cycles in G

**Procedure**: **SimpleCylcesNaive**(G)
1. let n be number of vertices in G
2. let allCycles be NULL
3. for each permutation p of (1, 2, ..., n)
    3.1. add first number in p to the end
    3.2. if p exists in G:
        3.2.1. add p to allCycles
4. return allCycles

Time complexity: NP
Space complexity: O(V)

**Algorithm – 2 (Johnson's)**

- Let *blockedSet* be a set of vertices which are currently not allowed to visit
- Let *blockedMap* map each vertex with a set of vertices such that unblocking the vertex would unblock all the vertices in the set.
- Let *stack* be a stack which stores the vertices currently in consideration
- Let *allCycles* be the list of all the cycles found.

**Input:** A directed graph G = {V, E}.
**Output:** The list of all simple cycles in G
**Procedure 1: SimpleCylcesDirectedJohnson(**G**):**
1. Let |V| be the number of vertices in G
2. Let startVertex be 1
3. while startVertex is |V|
    3.1. G' = subgraph of G where all vertices >= startVertex
    3.2. G'' = SCC(G')
    3.3. u = least index vertex of G"
    3.4. if u is present:
        3.4.1. clear blockedSet
        3.4.2. clear blockedMap
        3.4.3. FindCyclesInSCG(u, u)
        3.4.4. startVertex = u + 1
    3.5. else break from the while loop
4. return allCycles

**Input:** Two vertices, sv – start vertex, cv – current vertex.
**Output:** False if no cycle found else True
**Procedure 2: FindCyclesInSCG(**sv, cv**):**
1. Let found be false
2. push cv onto stack
3. add cv to blockedSet
4. for each edge e(cv, v) incident on cv:
    4.1. if v is sv:
        4.1.1. push sv onto stack
        4.1.2. cycle = stack
        4.1.3. reverse cycle list
        4.1.4. pop from stack
        4.1.5. add cycles to allCycles
        4.1.6. found = True
    4.2. else if v is not in blockedSet:
        4.2.1. if FindCyclesInSCG(sv, v) is True:
            4.2.1.1. found = True
5. if found is True:
    5.1. Unblock(currentVertex)
6. else:
    6.1. for each edge e (cv, v) incident on cv:
        6.1.1. add cv to blockedMap[v]
7. pop from stack
8. return found

**Input:** Vertex u, on which the unblock operation is performed.
**Output:** None
**Procedure 3: Unblock(**u**):**
1. remove u from blockedSet
2. if blockedMap contains u:
    2.1. for each vertex v in blockedMap[u]:
        2.1.1. if blockedSet contains v:
            2.1.1.1. unblock(v)
    2.2. remove u from blockedMap

Time complexity: O ((C + 1) * (V+E))
Space complexity: O (C * E)

**Algorithm – 3**

**Input:** V - no. of vertices, adj - Adjacency matrix of graph G.
**Output:** Transitive closure matrix or Path matrix for graph G.
**Assumption:** Each vertex is reachable from itself.
**Procedure:**
1. let adj be the Adjacency matrix.
2. let P be a |V|x|V| boolean array of reachability initialized to FALSE.
3. for each vertex v in V
    3.1. DFS(v,v)
4. Return P.

**Procedure: DFS (source,destination)**
1. mark P[source][destination] as 1
2. for all the vertices i reachable through destination (in adj[v])
    2.1. if P[source][i] is 0
        2.1.1. DFS(source,i)

Time complexity: O (V * (V+E))
Space complexity: O (V * V)

**Algorithm – 4 (Warshall's)**

**Input:** V - no. of vertices, adj - Adjacency matrix
**Output:** Transitive closure matrix or Path matrix.
**Procedure:**
1. let adj be the Adjacency matrix.
2. let P be a |V |x |V| Boolean array of reachability initialized to FALSE.
3. for each edge (u, v)
    3.1. P(u, v) = adj(u, v)
4. for k from 1 to |V|
    4.1. for i from 1 to |V|
        4.1.1. for j from 1 to |V|
            4.1.1.1. P[i][j] = P[i][j] || (P[i][k] && P[k][j])
5. return P

After obtaining P, it can be checked whether vertex j is reachable from vertex i. If P [i][j] is FALSE then vertex j can't be reached from vertex i, else there is a path from vertex i to j.

Time Complexity: O (V * V * V)
Space Complexity: O (V * V)

## IV. Implementation And Results

Below figures – 1, 2 demonstrate the time plot for algorithms 3, 4. The graph are generated randomly, with number of vertices varying from 1 to 500 for Warshall's algorithm and till 3000 for DFS algorithm. The values were plotted after taking an average over 10 trials. The plotting was done with the help of GNUPlot tool.

### *A. Ciruit Matrix*

The first approach is the naïve brute force of all permutations Of vertices to find the elementary circuits.

The permutation starts and ends with the same vertex to ensure it's a cycle, and the vertices in between are permuted from length 1 to V – 1, and each being distinct allowing no repeated vertices, which is an essential condition for elementary circuits, then checking for the validity of each.

The second approach is the Johnson's algorithm [3] which uses depth first search and strongly connected components to detect the cycles. It requires some ordering of the nodes. We assign the arbitrary ordering given by the strongly connected components [4]. There is no need to track the ordering as each node removed as processed. This method assumes input in directed graph manner, for undirected graphs we perform a preprocessing step, which involves addition of 2 directed edges for each edge in the undirected graph.

This method avoids finding any circuit more than once, and henceforth saving great deal of effort which could have been involved in fruitless search for non-existent circuits.

Following are the steps for building the circuit matrix.

1)To construct the circuit matrix for the graph, we need to extract all possible circuits first, then we could simply create a matrix of Circuits [C] [E], where C is the number of circuits and E is the number of edges where,

Circuit [i] [j] = true, if circuit i has edge j
Else Circuit [i] [j] = false

2). To find the all the circuits, we first find the elementary circuits/ simple cycles from the given graph (As mentioned in above algorithm either brute force or using 2nd approach)

3). After obtaining the elementary circuit basis, we can generate all possible cycles by performing XOR over and then check if it's a valid circuit of the graph. This procedure enumerates all circuits from the graph.

A circuit is simply a path having repeated vertices but no repeated edges. So, now the task only remains to fill the entries in Circuit Matrix.

### *B. Path Matrix*

The path matrix denoted by G' has been introduced by Randić. The path matrix of a vertex-labeled connected simple graph G is a square V x V matrix whose entries are defined as follows:

$$[G'] = \begin{cases} p(i,j) & \text{if } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

where p (i, j) is the total number of paths in the subgraph G' obtained by removing the edge i-j from graph G and p is total number of paths in G.

We have used the property of *Transitive closure* to solve the problem. The Closure says "First a relation may or may not be transitive but after assuming it's transitivity, it can be completed upto a transitive one".

Given a graph, we have to find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j). The reach-ability matrix is called *Transitive closure* of a graph. Algorithm - 3 describes this process. Another approach for calculating the Path Matrix P has been described in Algorithm 4 In this algorithm, we first have to initialize all entries in path-array P which is of order |V| x |V| as 0. Then we have to call DFS function for every node of graph to mark reachable vertices in P [V] [V].

Recursion terminating condition is that "We don't call DFS for an adjacent vertex if it is already marked as reachable in P [V] [V]" [3]. The query whether any vertex is reachable from any other vertex can be resolved easily. We can also use BFS (breadth first search) in place of DFS(depth first search).[2][1] If the given graph is dense , time complexity of the algorithm will be O(V*V*V) as E will tend to O(V*V).

## V. Conclusion

We have worked out different types of implementations in various languages such as python, JAVA to help visualize the graph from its Incidence and Adjacency matrix. Thus we successfully implemented different algorithms mentioned above to find out circuit and path matrix of a given graph in optimal time.

## VI. References


[1]"3.9 Case Study: Shortest-Path Algorithms", Mcs.anl.gov, 2017. [Online]. Available: http://www.mcs.anl.gov/~itf/dbpp/text/node35.html. [Accessed: 04- Sep- 2017].

[2]"Floyd–Warshall algorithm", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm. [Accessed: 04- Sep- 2017].

[3]"GeeksforGeeks | A computer science portal for geeks", GeeksforGeeks, 2017. [Online]. Available: http://Geeksforgeeks.org. [Accessed: 04- Sep- 2017].

[4]T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to algorithms. Cambridge, Massachusetts: The MIT Press, 2014.

[5]D. Johnson, "Finding All the Elementary Circuits of a Directed Graph", SIAM Journal on Computing, vol. 4, no. 1, pp. 77-84, 1975.

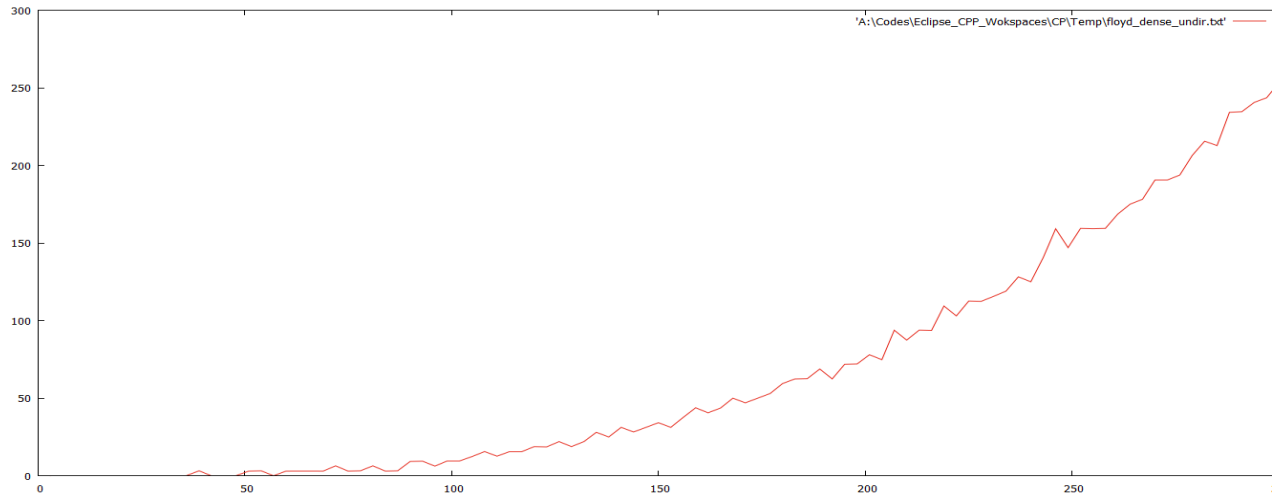


**Figure 1 Plot of time vs vertices for Warshall's Algorithm**

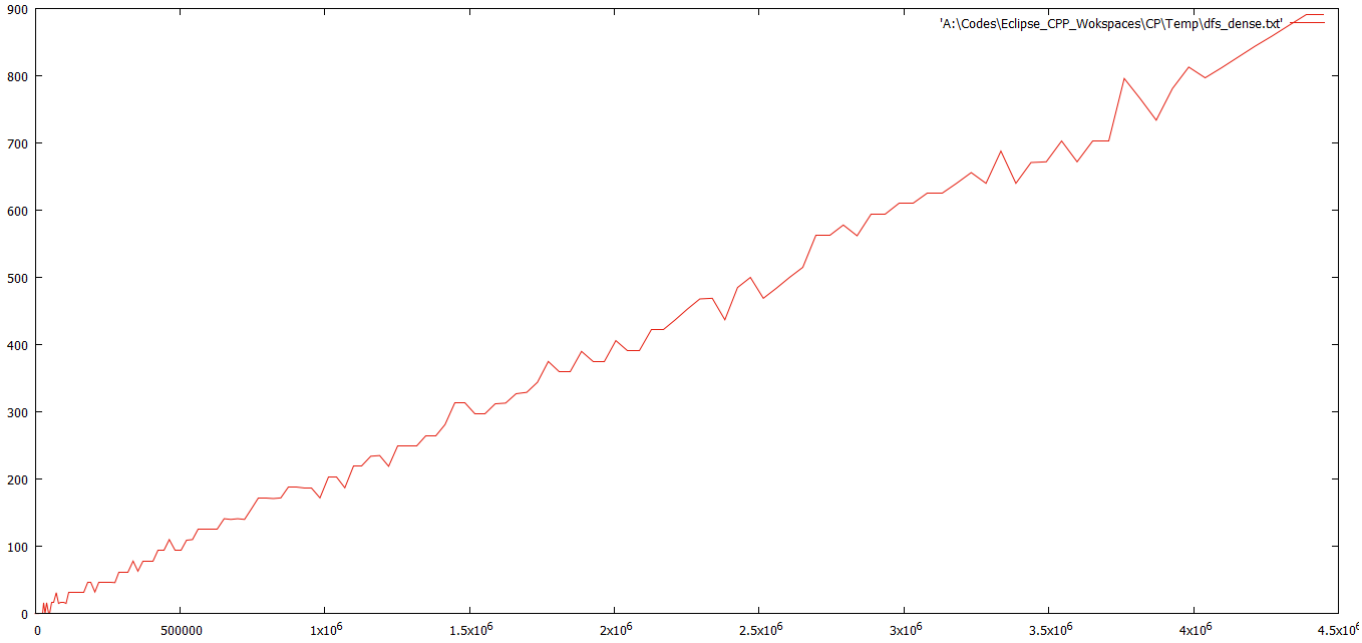


**Figure 2 Plot of time vs edges for DFS approach**