GRAPH REACHABILITY

Rabin Thapa
B.S. Mathematics | Classs of '21
Youngstown State University
Youngstown, Ohio 44555
rthapa01@student.ysu.edu

Advisor: **Dr. Feng George Yu**
Department of Computer Science
Youngstown State University
Youngstown, Ohio 44555
fyu@ysu.edu

December 7, 2019

ABSTRACT

In the recent decades, graph data has been used extensively in myriad applications. Its increasing popularity and applicability have been a motivation for rigorous studies in graph data mining and management techniques. In this paper, we introduce the concept of graph reachability query, summarize different approaches that have been proposed to answer reachability and compare those approaches on the basis of query processing time, storage required and their ease of implementation.

***K*eywords** Graph mining · Graph management · Graph reachability · Graph reachability query · Transitive closure · Topological sort · Tree cover · Dual labeling

1 Introduction

Graph, the visual aspect of mathematics, is the art and science of connecting dots by lines and curves to represent structural, sequential and logical information[1]. It works particularly well in situations where characteristics such as connectivity/relationship between the vertices is important irrespective of the actual distances between them[1].
In computer science, a graph data structure $G(V, E)$ consists of a set of vertices (V) and edges (E) joining them. Some popular examples of graph data are social networks, web sites/internet, XML documents, biological/chemical networks, musical pieces, etc. Due to its applications in wide variety of data mining problems, graph mining and graph management have become an important topic of research.

This paper is organized in the following way: Section 2 presents the motivation behind this research along with different applications of graph reachability, Section 3 contains some graph algorithms and concepts of graph theory that are essential in studying graph reachability approaches, and Section 4 contains various approaches used for answering reachability queries.

2 Motivation

We mentioned that the application of graph theory includes variety of fields. Some of them are social networks[2] (targeted advertising, pattern extraction), Web Data Mining[2] (XML documents), Biological Data Mining[2] (drug discovery, DNA analysis), etc. Now, let's list some of the well-known applications[2] of graph reachability:

- Social networks: sites like facebook[6] which has millions of users interacting with one another
- Security: finding possible connections between suspects (criminal investigation)
- Biological data: Is a particular protein involved directly or indirectly in the expression of a gene?
- Pattern matching: It could be a key to many pattern matching and other graph theory problems

3 Preliminary Concepts

3.1 Tree, Weighted Graph, Digraph and Directed Acyclic Graph (DAG)

A graph that contains pairwise oriented edges between its vertices is called a *directed graph* or simply, a *digraph*. Conversely, an *undirected graph* is a graph in which all of its edges are undirected. A *tree* is an undirected graph in which there is a unique path between any two vertices. Alternatively, it can be defined as a graph that has no cycles. If a directed graph is a tree, it is called *directed acyclic graph (DAG)*. Likewise, a graph in which each edge has a weight is called a *weighted graph*.

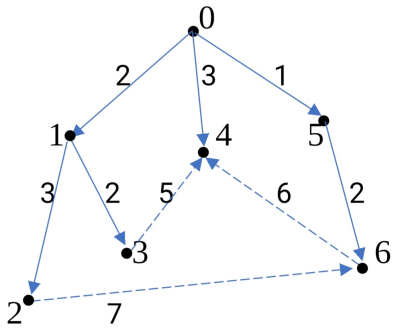

Fig 1. A weighted digraph, G_1

3.2 Minimum Spanning Tree

For a graph G, a *spanning tree* T is a connected subgraph of G that contains all the vertices of G. If $G(V, E)$ has $|V| = n$ vertices and $|E| = m$ edges, the spanning tree has exactly $n - 1$ edges. If G is a weighted graph, then the sum of the weights of all the edges in T is said to be the weight of the tree. Out of the spanning trees of G (spanning trees may not be unique), the one with the lowest weight is called the *minimum spanning tree* (T_{min}).
We discuss Prim's and Kruskal's Algorithms here. Both of them are examples of Greedy algorithm.

3.2.1 Prim's Algorithm

Prim's algorithm emphasizes the fact that a spanning tree must be a connected graph. To find the minimum spanning tree using this algorithm[4]:

1. Start with any vertex in the graph. Add in the edge with the lowest weight leaving the vertex.
2. Among the edges connected to the already formed spanning tree. pick the edge with the smallest weight. Check if it forms a cycle with the spanning tree formed so far. If it does not, include the edge. Else, discard it.
3. Repeat step 2 until all the vertices are included in the spanning tree.

Running time: $O(|V| + |E|)log|V|)$

3.2.2 Kruskal's Algorithm

Kruskal's algorithm exploits the fact that an acyclic subgraph consisting of $n - 1$ edges always forms a spanning tree of graph G with n vertices. To find the minimum spanning tree using this algorithm[5]:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the edge with the smallest weight. Check if it forms a cycle with the spanning tree formed so far. If it does not, include the edge. Else, discard it.
3. Repeat step 2 until there are $n - 1$ edges in the spanning tree.

Using Kruskal's algorithm, the following spanning tree can be generated for graph G_2.

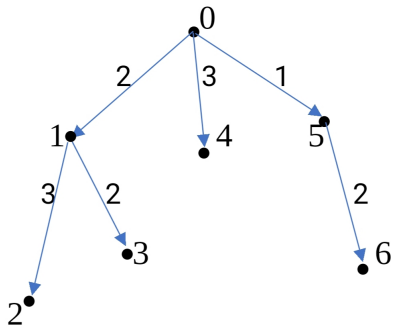

Fig 2. $T_{min}(G_1)$, the minimum spanning tree of G_1

3.3 Dijkstra's Algorithm

This algorithm can be used to find the shortest paths to all the vertices from a vertex in a connected, weighted graph. It is also an example of Greedy algorithm (making current best choice at each stage). The pseudocode for Dijkstra's algorithm[7] is:

- Initialize the cost of the source node (initial current node) to 0. Initialize the cost of every other node to ∞.
- While there are unvisited nodes left in the graph,
  1. Among the nodes connected to the current node, select the unvisited node N with the lowest cost (greedy choice)
  2. Mark N as visited node.
  3. For each node X,
     If (N's cost + cost of $(N, X)) < X$'s cost
     X's cost = N's cost + cost of (N, X)
     Add N to the optimum path formed so far. Mark N as the current node.

Using Dijkstra's algorithm, let's find the shortest path from vertex $1(v1)$ to vertex $4(v4)$. The steps involved are:
First let the weight of $v1$ to 0. Set the weight of $v3$ and 7 to . For $v2$: weight of $v1$ + weight of $(v1, v2) = 0+3 = 3 <=$ weight on $v2$ So, new weight on $v2 = 3$. For $v3$: weight of $v1$ + weight of $(v1, v3) = 0 + 2 = 2 <=$ weight on $v3$ So, new weight on $v3 = 2$.
Among the vertices adjacent to $v1$,$v3$ has the lower new weight. So, let's set $v3$ as the current vertex. The only adjacent vertex of $v3$ is $v4$. And edge $(v3, v4)$ has a weight of 6. Now, $2 + 6 = 8 <=$ weight on $v4$. Thus, the new weight on $v4 = 8$ (which implies that the length of $(v1 - v3 - v4)$ path is 8).
Now, let's set $v1$ as the current working vertex. The only edge going out of $v1$ connects it to $v8$ and has a weight of 7. So, new weight on $v8$ is $3 + 7 = 10$. Finally, setting $v8$ as the working vertex, we get proposed new weight on $v4 =$ weight on $v8+$ weight of edge $(v8, v4) = 10 + 6 = 16 > 8 =$ current weight on $v4$.
No other ways lead to $v4$ from $v1$. So, the shortest path is $v1 - v3 - v4$ and its weight is 8.

Now, let's apply Dijkstra's algorithm to find the shortest path from vertex 0 to all other vertices in G_1:

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0_0	2_0	INF_0	INF_0	3_0	1_0	INF_0
5	0_0	2_0	INF_0	INF_0	3_0	1_0	3_5
1	0_0	2_0	5_1	4_1	3_0	1_0	3_5

Fig 3. Dijkstra's Algorithm in G_2

The weights in the final row represent the length of shortest paths from vertex to different vertices. The paths can be traced by following the subscripts. For instance: The shortest path from vertex 0 to vertex 2 has length 5. The weight 5 has a subscript 1. So, vertex 1 falls in the path right before vertex 2. The weight in final row for vertex 1 is 2 which has a subscript 0 (which means the path is complete). And the path is $0 - 1 - 2$.

3.4 Adjacency Matrix

A graph data is often stored in the form of adjacency matrix or adjacency list. *Adjacency matrix* $Adj(G)$ is square matrix used to represent the graph G. An entry a_{ij} of the adjacency matrix A contains information on connectivity or distance between node i and node j of graph G. The values of the entries a_{ij}'s can be:

- *binary (0 or 1)*: 1 if an edge from node i to node j and 0 if there is no edge from node i to node j
- *edge weights*: the weight on the edge connecting node i to node j. The value is assigned infinity if there is no such edge and NULL for the diagonal entries (a_{ij} such that $i = j$)of the matrix

The adjacency matrix for the above digraph G_2 is

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	0	0	1	1	0
1	0	1	1	0	0	0	0
2	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	1	0	0

Fig 4. $Adj(G_2)$

3.5 Floyd-Warshall's Algorithm

Floyd-Warshall's Algorithm is an algorithm to find the transitive closure of a digraph. It can also be used to find the short path from a vertex to all other vertices in a weighted graph.
For a digraph with n vertices, the transitive closure (TC) is given by $TC = W^{(n)}$ where $\forall k, 0 \leq k \leq n$:

$$W^{(k)} = \left[W_{i,j}^{(n)} \right]$$

where $0 \leq i, j \leq n-1$. $W^{(0)}$ is the adjacency matrix of the digraph and $\forall k, 1 \leq k \leq n$, $W_{i,j}^{(k)}$ is defined recursively as[8]:

$$W_{i,j}^{(k)} = W_{i,j}^{(k-1)} \bigvee \left[W_{i,k-1}^{(k-1)} \bigwedge W_{k-1,j}^{(k-1)} \right]$$

[citation needed]

$Adj(G_2) =$

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	0	0	1	1	0
1	0	1	1	0	0	0	0
2	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	1	0	0

$= W^{(0)}(G_2)$

$W^{(1)}(G_2) =$

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	0	0	1	1	0
1	0	1	1	0	0	0	0
2	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	1	0	0

continuing the process gives

$$W^{(7)}(G_2) = \begin{array}{c|ccccccc} & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} \\ \hline \mathbf{0} & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ \mathbf{1} & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \mathbf{3} & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{5} & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \mathbf{6} & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{array} = TC(G_2)$$

3.6 Topological Sort

Topological sort of a graph is the linear arrangement of its vertices in such a way that for any edge (u, v), u precedes v in the ordering[7]. A topological sort of a directed graph can be constructed if and only if the graph is acyclic.
A topological sort of $T_{min}(G_1)$ would be:

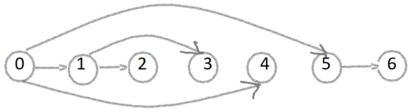

Fig 5. A topological sort of $T_{min}(G_1)$

A DAG can have multiple topological sorts, equivalent to one another. So, topological sort of a graph is not unique. Some of the applications[6] of topological sort are:

- Class inheritance
- Course prerequisites
- Pipeline of computing jobs
- Scheduling problems

4 Reachability Approaches

Reachability[9] in a graph G is the ability to get form one vertex to another with in the graph. A *reachability query*, denoted by $u \rightsquigarrow v$ or *reach (u,v)*, is *true* if and only if there is a directed path from node u to node v.

For instance, in the following graph G_2: $0 \rightsquigarrow 6$ is $true$ where as $4 \rightsquigarrow 2$ is $false$.

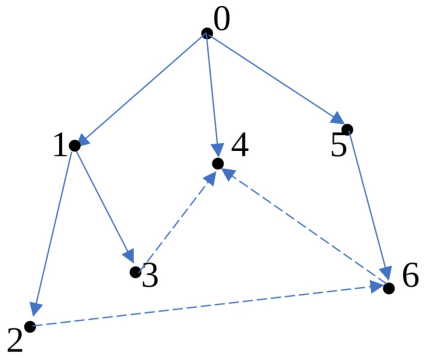

Fig 6. A directed graph, G_2

4.1 Depth First Search

Depth First Search (DFS) is a technique for graph traversal in which all the vertices accessible through a branch are visited before moving to the next branches. It can be employed to find if one vertex is reachable from another.
To answer $u \rightsquigarrow v$, we do the following steps[2]:

1. Start Depth First Traversal from vertex u
2. If the vertex v is found, stop the search. Then report $u \rightsquigarrow v$ is *true*.
3. If all vertices have been visited, then report $u \rightsquigarrow v$ is *false*.

For instance, to determine $0 \rightsquigarrow 6$ in graph G_2 using DFS, we visit the following vertices in order:

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 6$$

So, $0 \rightsquigarrow 6$ is *true* in G_2.

There is another graph traversal technique called Breadth First Traversal (BFS) which can be used similarly. In BFS, all the neighbouring vertices of the 'pointing vertex' are visited before moving down a particular branch. Upon using BFS, to determine $0 \rightsquigarrow 6$ in graph G_2, we visit the following vertices in order:

$$0 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$$

Thus, $0 \rightsquigarrow 6$ is *true* in G_2.

Note: In the above implementation of DFS/BFS, precedence in traversal, among the neighbouring vertices of the 'pointing vertex', is arbitrarily given in ascending order.

For DFS/BFS, no index construction time and space is required. But the query processing time is $O(m+n)$. At worst case, the entire graph should be traversed to answer the query!

4.2 Transitive Closure

The *transitive closure,* $TC(G)$ of a graph G, is a graph which contains an edge (u, v) whenever there is a directed path from u to v (Skiena 1990, p. 203). In other words, it is the smallest relation in the set $G(V, E)$ that:

- contains G $(G \in TC(G))$
- is transitive $((u, v), (v, w) \in TC \Rightarrow (u, w) \in TC) \; \forall \; u, v, w \in G$

The Transitive closure of digraph G_2 is:

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	1	1	1	1
1	0	1	1	0	1	1	1
2	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	1	1	0

Fig 7. $TC(G_2)$

Let's use the above transitive closure of G_2 to determine we can $0 \rightsquigarrow 6$. The element in 0^{th} row abd 6^{th} column is 1. So, $0 \rightsquigarrow 6$ is *true*. But $6 \rightsquigarrow 0$ is *false* since the element in 6^{th} row abd 0^{th} column is 0.

To store transitive closure of a graph, it takes $O(n^2)$ space. The construction of transitive closure takes $O(n^3)$ time. But the reachability query can be answered in constant time, $O(1)$.

4.2.1 Optimal Tree Cover

Agrawal et al. proposes a method to answer reachability queries in a graph using its tree cover. In this method, a spanning tree of the graph is constructed first. This distinguishes *tree edges* from *non-tree edges*. Then, a label is assigned to each vertex in the graph based on the network of *tree edges* only. The label for a vertex u, $label(u)$ is in the form $[u_{start}, u_{end}]$ where u_{end} is the order in which vertex u is traversed during post-order traversal of the tree and u_{start} is the smallest post-order index among those of the descendants.
Then, the *non-tree edges* are taken into account. For every directed *non-tree edge* (x, y), the label of y is added to the

label set of x. So, the label set of x consists of its initial label as well as those labels inherited from neighbouring vertices it is connected to by outward directed *non-tree edge*.

$$label(x) = \{[x_{start}, x_{end}], [y_{start}, y_{end}], ...\}$$

It takes $O(n^3)$ time for index construction and the index size is $O(n^2)$ at worst case.
Query processing:

$$u \rightsquigarrow v \Rightarrow u_{start} \leq v_{end} < u_{end}$$

For query processing, it takes $O(n)$ time.

In Fig 7. the tree cover of graph G_3 is shown in solid edges and dotted lines represent *non-tree edges*. Vertex A has the label $[1, 7]$ because it is the seventh vertex to be traversed during post-order traversal of the tree and the lowest post-order index among those of its descendants is 1.
For Vertex D, $[2, 2]$ is its initial label where as $[4, 4]$ is an inherited label since it is connected to E having label $[4, 4]$.
For Vertex G, the index is $\{[5, 5], [1, 7]\}$. $[4, 4]$, inherited from E, is omitted to avoid redundancy since $[4, 4] \subseteq [1, 7]$, $[1, 7]$ inherited from A.

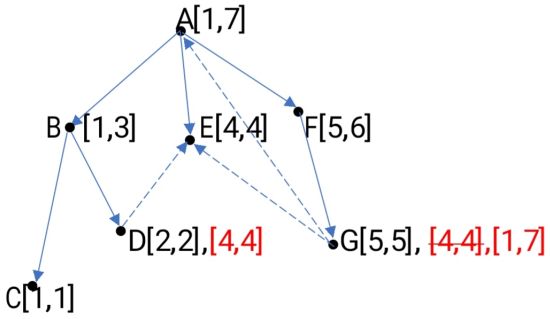

Fig 8. A tree cover for a graph G_3

Now for instance, let's check $G \rightsquigarrow C$ and $E \rightsquigarrow F$ in the above graph G_3. But $G \rightsquigarrow C$ is *true* since $(1 \leq 1 < 7)$ which is true. $E \rightsquigarrow F$ is *false* since $(4 \leq 6 < 4)$ is not *true*.

5 Conclusion

From the above study, it is clear that graph reachability has application in many areas. A number of approaches have been proposed to answer reachability queries. The performance of different approaches for query processing can be summarized in the following table.

Method	Query time	Construction	Index size
DFS/BFS	$O(n+m)$	–	–
Transitive Closure	$O(1)$	$O(n^3)$	$O(n^2)$
Optimal Tree Cover	$O(n)$	$O(n^3)$	$O(n^2)$

Fig 9. Comparison of different approaches for query processing

6 Future Works

As a continuation of this project, I have following as my future plans:

- study other approaches to answer reachability queries. Such approaches include:
  - GRIPP
  - Dual Labeling
  - Optimal Chain Cover
  - HOP Cover (2-HOP, 3-HOP)
- implement the algorithms for reachability approaches (The examples above were solved manually)
- work on a sample data graph to run simulate processing of reachability queries

References

[1] P.Leonardo, A Graph Topological Representation of Melody Scores

[2] P. Parchas, Graph Reachability, http://www.cse.ust.hk/ dimitris/6311/L24-RI-Parchas.pdf

[3] J. Ugander et.al., The Anatomy of Facebook Social Graph, https://arxiv.org/pdf/1111.4503.pdf

[4] Minimum Spanning Trees and Prim's Algorithm, https://www.cse.ust.hk/ dekai/271/notes/L07/L07.pdf

[5] Kruskal's Minimum Spanning Tree Algorithm | Greedy Algo-2, https://www.geeksforgeeks.org

[6] Directed Graphs, https://algs4.cs.princeton.edu/lectures/42DirectedGraphs.pdf

[7] Topo-Sort and Dijkstra's Greedy Idea, https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

[8] Warshall's Algorithm, https://youtu.be/meW98HdJaRI

[9] J X Yu et. al.,Graph Reachability Queries: A Survey