Application of Kosaraju’s Algorithm in Determining Closely Related Users in Social Medias

Edward Alexander Jaya / 13517115
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517115@std.stei.itb.ac.id

Abstract— Social medias in Internet are considerably vital to the world. Nowadays, social medias are not just a media for interconnecting people, but these medias can also determine whether group of users are closely related or not. This determination is useful when it comes to clustering-based sampling and recommendation to follow a user or a group of users. Social medias’ network of users can be modeled with directed graph and users who are closely related are modeled with strongly connected components in that graph. Kosaraju’s algorithm is DFS-based algorithm to find all strongly connected components in a directed graph with minimal complexity. This paper will discuss the detailed Kosaraju’s algorithm as one solution of solutions available for solving the closely related users problem.

Keywords—graph; algorithm; DFS; adjacent; connected; Kosaraju

I. INTRODUCTION

The Internet has always been a very novel feature for human kind in this digital era. Since the world is developing faster with its technologies, more and more people are joining the Internet to communicate with each other. One of the most important reasons of why humans need to interact with each other is to fulfill their needs, which develops and varies from time to time. For example, about several centuries ago, humans cannot communicate with another person who lives in a separate continent in a very short period of time. Nowadays, inventors with brilliant ideas are vying for reducing latency time in the Internet when one person is communicating with another person or perhaps a larger group of people. Algorithms play a very vital role in increasing the quality of these “communicating services”, known better as social medias.

Social medias are considerably vital to vast percentage of the world’s human population, especially young people which are born in a specified period of time. This period-of-time is a prerequisite of them (young people) to be called “Generation Z and Alpha”. Social medias are also based on “real friendships”, therefore they simulate and construct the friendship-world as close to the real world friendships as possible. That’s why social medias have an interesting feature called “follow” or “add friend”.

Naturally and digitally, a user who knows another user closely are going to click the “follow” button or perhaps “add friend button”. Have this procedure followed, that user will be called “a friend” of that another user. As time goes, social medias are more sophisticated than just identifying friends of users only. Social medias nowadays can even notify and recommend new unfriended users to a user to follow or add.

How do social medias recommend new users to a particular user? By checking the probability of the user’s probability to recognize recommended users. The probability will be high enough if that particular user is more closely related with recommended user’s friends. Social medias use artificial intelligence to do that function, but simpler approach has been done in this paper to explain how Kosaraju’s algorithm can determine new recommendations. Social medias’ list of friendship networks can be modelled with a directed graph.

II. FUNDAMENTAL THEORY

A. Graph Definition

A graph can be represented as pair of sets (V, E). V is a notation for nonempty set of vertices and E is a notation for set of edges. Furthermore, graph can be denoted as G (V, E). A vertex can be depicted with shape of a point which represents an object. An edge can be depicted with shape of a line. The visualization of graph is as follows:

![Fig. 1 Graph](source: Munir, R. (2009), Matematika Diskrit. Bandung, Jawa Barat: Institut Teknologi Bandung)

A vertex in a graph is denoted by alphabetical characters or real numbers or the combination of those two. An edge can be denoted as $e = (v_a, v_b)$, which means edge e connects a vertex $v_a$ with another vertex $v_b$. 
B. Graph Types

A graph can be divided into two types by considering the direction of edges in a graph:

1. Undirected Graph

This type of graph does not have an edge with direction. The edges are a set of unordered pairs which means that edges (a, b) and (b, a) are considered as the same.

2. Directed Graph

This type of graph does have all edges with direction. The direction is depicted with the shape of an arrow. The edges are a set of ordered pairs which means that edges (a, b) and (b, a) are considered different. Consider we have an edge (a, b). It can be said that a is an initial vertex and b is a terminal vertex. The visualization of this type of graph is as follows:


C. Fundamental Graph Terminology

1. Adjacent

For undirected graph, consider this notation: \( e = (v_a, v_b) \). It can be inferred that \( v_a \) and \( v_b \) share the same edge \( e \). Therefore, \( v_a \) and \( v_b \) can be called adjacent.

For directed graph, \( v_a \) and \( v_b \) can be called adjacent if there exists an edge from \( v_a \) to \( v_b \) or from \( v_b \) to \( v_a \).

2. Path

A path which has the length of \( n \) from vertex \( v_0 \) to \( v_n \) is a sequence of vertices and edges by turns that have sequence of \( v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n \).

3. Connected (Directed Graph)

In a directed graph \( G \), if the undirected graph of \( G \) connected, \( G \) is a connected graph. If there exists a directed path from \( v_i \) to \( v_j \) and \( v_j \) to \( v_k \), those two edges are strongly connected (there is always a path between all pairs of vertices).

4. Subgraph

Graph \( G' \) is a subgraph of \( G \) if all vertex sets and edges sets of \( G' \) are the subsets of the vertex sets and edges sets of \( G \).

5. Strongly Connected Components

Strongly connected components in a directed graph \( G \) is a maximal strongly connected subgraph.

D. Depth First Search

Depth First Search (DFS) is a type of graph-traversing algorithm alongside Breadth First Search (BFS). Recursion is often implemented for this algorithm to run although there exists an iterative-based DFS algorithm. In this paper, recursive DFS will be discussed. The algorithm runs as follows:

Let \( G \) be the notation of graph which has \( n \) vertices. Let’s say the traversal will be started at vertex \( v \). The next visited vertex will be the vertex \( w \) which is adjacent to vertex \( i \). Do the traversal (DFS) again from vertex \( w \) recursively. If the traversal finally reaches vertex \( u \) which has no unvisited neighbors (adjacent vertices), back-track the traversal until it finds the last visited vertex and adjacent to the unvisited vertex \( w \). Start the traversal again from vertex \( w \). The traversal is said to be over if all of the vertices in \( G \) have been visited and these vertices can be reached from the starting vertex. (In this case, vertex \( v \)). The example of Depth First Search will be discussed in the next sections.

E. Brute force

Brute force is considered as straightforward approach for problem solving, usually based on problem statement and given concept definition. Brute force algorithm solves the problem with very simple way, for example by enumerating all possibility in the problem and solution space. This type of algorithm is also often said to be naïve algorithm.

III. Kosaraju’s Algorithm

Kosaraju’s algorithm is found by Sambasiva Rao Kosaraju in 1978. Kosaraju’s algorithm is used to find all strongly connected components in a directed graph. As cited in preceding section, strongly connected components is a maximal strongly connected subgraph, which means there is always a directed path from vertex \( v_i \) to vertex \( v_j \) and in reverse, there is always a directed path from vertex \( v_j \) to vertex \( v_i \). Vertex \( v_i \) and vertex \( v_j \) are different vertices and these vertices are subsets of the set of all vertices in subgraph of graph.

This algorithm is based on DFS traversal which will be run twice. A stack which contains all vertices in graph \( G \) is also needed in the algorithm. The detailed Kosaraju’s algorithm will be explained in succeeding sentences.

First step, let there exists a graph \( G \). Create an empty stack, let’s say the notation of stack is \( S \). Do the DFS algorithm, starting from vertex \( v \). After the traversal reaches a vertex \( u \) which has no unvisited adjacent vertices, push vertex \( u \) to stack \( S \). Then, do DFS again from the last visited vertex and adjacent to unvisited vertex \( v \). Repeat the same process from vertex \( w \) until DFS has already done to be executed and stack \( S \) contains all vertices in \( G \). If DFS has already finished to execute but stack \( S \) does not contain all vertices in \( G \), find unvisited vertex \( x \) in \( G \), then start the process again from vertex \( x \) until stack \( S \) contains all vertices in \( G \).

Second step, reverse all directions of all edges in graph \( G \) (for example: from pointing left to pointing right). New graph obtained from reversing the edges is called transpose graph \( G^T \).
Third step, pop top vertex from stack S. Let the popped vertex be $v_o$. Start the DFS algorithm again from vertex $v_o$ as the starting point. All reachable vertices from vertex $v_o$ are strongly connected component. Remove all vertices (from stack S) which exist in previously cited strongly connected component’s vertices. While S is not empty yet, do the third step again. If S is empty, all strongly connected components have been found in graph G and Kosaraju’s algorithm is done.

The complexity of this algorithm is very fast, which is $O(V + E)$ because it utilizes DFS only two times. $V$ is the number of vertices in graph G and $E$ is the number of edges in graph G. Kosaraju’s algorithm complexity is derived by computing first step complexity: $O(V+E)$ and third step complexity: $O(V+E)$. It is known that $O(V+E) + O(V+E) = O(V+E)$. The pseudocode of this algorithm is shown below:

```python
def KosarajuAlgorithm():
    initializeEmptyStack()
    markAllVerticesVisited(false)
    # Fill vertices in stack
    for i in range [0..vertices_number]:
        if visited[i]==False:
            addToStack(i) #recursive
    reverseGraph()
    markAllVerticesVisited(false)
    while stack_is_not_empty:
        i = popStack()
        if visited[i]==False:
            DFS(i)
```

Fig. 3 Pseudocode of Kosaraju’s Algorithm

(Source: https://www.geeksforgeeks.org/strongly-connected-components/, with some modification by writer)

IV. MODELING THE PROBLEM

Let user $a$ and user $b$ exist in a social media X. Model user $a$ as vertex $v_a$ and model user $b$ as vertex $v_b$. Let’s say user $a$ has followed user $b$ but user $b$ has not followed user $a$. From the vertex $v_a$, draw an edge $arc$ to the vertex $v_b$. Do not draw an edge from $v_b$ to $v_a$. The visualization can be seen below:

![Graph modeling](image1.png)

Now, let username “Edward” in social media X has followed username “Felix” but, he has not followed “Edward” yet. Username “Felix” has followed “Ivan”, but he has not followed “Felix” yet. Then “Ivan” recently follows “Anton” but “Anton” has not yet followed “Ivan”. But surprisingly, “Anton” has followed “Edward” but “Edward” has not yet followed “Anton”. The question is, is there any possibility that the four users: Edward, Felix, Ivan, and Anton are real friends? It is later known that are real friends, but social media X has already sent the notification to every user to follow each other. Social media X uses Kosaraju’s algorithm to find the relationship. Consider the graph modeling below:

![Graph modeling of users in social media X](image2.png)

Fig. 5 Graph modeling of users in social media X

From the figure above, it can be inferred that the constructed graph is a strongly connected graph. To prove it, we have to show that for every vertex $i$ there is a path to vertex $j$ in graph and for every vertex $j$ there exists a path to vertex $i$. Based from the figure above, there is a path from “Edward” to “Felix”. There is a path also from “Edward” to “Ivan” by visiting “Felix”. And “Edward” can reach “Anton” by visiting “Felix” then “Ivan”. The same goes for “Felix” to other three friends, and so on.

With this modeling, it is expected that strongly connected components in a graph determines the closely-related vertices in the subgraph. Another usage of finding strongly connected components is to know whether they belong to the same group or not. Or have similar hobbies. This is very useful for improving quality of social medias.

V. IMPLEMENTATION OF KOSARAJU’S ALGORITHM TO DETERMINE STRONGLY CONNECTED COMPONENTS IN SOCIAL MEDIA

Consider the graph G below:
The goal is to find all strongly connected components in graph above with Kosaraju’s algorithm. Before going on with Kosaraju’s, set a convention that lower indexes have a higher priority to be visited first. The problem solving is as follows:

First step: Initialize empty stack S. Then set vertex 0 as the starting point because lower indexes have higher priority to be visited first. It can be seen that vertex 0 has one adjacent vertex, which is vertex 1. Visit vertex 1. Then, visit vertex 2 (because this vertex is adjacent to vertex 1). It can be seen that vertex 2 has two different adjacent vertex, which are vertex 3 and vertex 4. Visit vertex 3 first as it has higher priority. Now, it is known that vertex 3 does not have any unvisited adjacent vertex. Vertex 3 is indeed adjacent to vertex 0, but vertex 0 is already visited. Therefore, push vertex 3 to the stack S so stack S has one element.

Do the back track from vertex 3 to vertex 2. It is known that vertex 2 has vertex 4 as its unvisited adjacent vertex. Therefore, visit vertex 4. From vertex 4, visit vertex 5, then visit vertex 6. It is known that vertex 6 does not have an unvisited adjacent vertex, so push vertex 6 to the stack. Back track again to vertex 5 and push vertex 5 to the stack. Back track to vertex 4 and push vertex 4 to the stack. Repeat the same process to vertex 2 and push vertex 2 to the stack, then backtrack and push vertex 1 to stack, backtrack and finally push vertex 0 to stack. DFS has already finished but consider again the stack contents which has not contained all of the vertices in the graph.

So, search the unvisited vertex with lowest index. Start DFS from vertex 7. Vertex 7 has two different adjacent vertices, but one has already visited. Therefore, visit vertex 8. It is known that vertex 8 does not have any unvisited adjacent vertex, so push 8 to stack, then do the backtrack and push 7 to stack. Now, DFS is finally completed and first step of Kosaraju’s algorithm is done.

Now, compute graph transpose G (G^T) by reversing all directions of all edges in G. Graph G^T is shown below:

Because G^T is already computed, second step is marked finished and it is the time to execute third step. See the contents of the stack. The visualization of the stack is shown below:

It can be seen that vertex 7 is located at the top of stack. Now, pop vertex 7 from stack so now vertex 8 is currently top of stack. Do the DFS traversal from vertex 7. Because vertex 7 does not have adjacent vertex, then it is known that vertex 7 is the strongly connected component. Now, pop vertex 8 from the stack and start DFS traversal from vertex 8. Vertex 8 has an adjacent vertex but that vertex has already visited. So, stop the DFS traversal and conclude that vertex 8 is a strongly connected component.

Now, pop vertex 0 and start DFS from that vertex. From vertex 0, vertex 3 can be reached, then vertex 2, and then vertex 1. Vertex 1 does not have any unvisited adjacent vertex, so DFS is terminated and conclude that 0-3-2-1 is a strongly connected component. Before popping another vertex from stack, remove vertex 1, vertex 2 and vertex 3 from the stack so the stack can be shown like this:
Pop vertex 4 from stack and start DFS from that vertex. From vertex 4, visit vertex 6 as its unvisited adjacent vertex. Then, visit vertex 5. DFS is now terminated because all other vertices have already been visited. It can be concluded that 4-6-5 is a strongly connected component. Before popping another element from stack, remove vertex 5 and vertex 6 from stack. Now, stack is empty and Kosaraju’s algorithm has already done to search all strongly connected components in graph G.

![Fig. 9 The contents of stack, updated.](image)

Fig. 10 Final result of Kosaraju’s algorithm

The conclusion, vertex 0-3-2-1 is more closely related and social media should notify them to follow each other because they are close friends. Then vertex 4-6-5 is closely related also and social media should notify them to follow each other also. Vertex 7 and vertex 8 are not closely related enough to be close friends.

VI. BRUTEFORCE APPROACH TO FIND STRONGLY CONNECTED COMPONENTS

Finding strongly connected components can also be used with bruteforce algorithm. Let graph G contains n number of vertices. To simplify the problem, find a strongly connected component which includes vertex i first. For clarity, let’s say all vertices in G are in integer types from 0 to n-1. One easy approach is to initialize a list L of all vertices of G from vertex 0 to n – 1.

For every vertex from 1 to n-1 in L, check if these vertices can be reached from vertex 0 or in other words, there exists a path from vertex 0. Check again whether vertex i can reach vertex 0 or not. This can be easily checked by using DFS traversal twice. One for checking the path from vertex 0 to vertex i and one for checking the path from vertex i to vertex 0. If vertex i, with 1 <= i <= n - 1 (i is integer) is unable to be visited from vertex 0 (path from vertex 0 to vertex i does not exist), then remove vertex i from L. If vertex i can be visited from vertex 0, but in reverse, vertex 0 cannot be reached from vertex i, then remove vertex i also from L. Do remove vertex i which satisfies the condition cited before for n-1 times.

Now we have a list L with vertex 0 and vertices that can be reached from vertex 0 and vice versa, but it is needed to repeat the above process from vertex j with 2 <= j <= n-1 (j is integer). Looping of process is needed because it is not confirmed if there exists a path from every vertex in L excluding vertex 0 to all other vertices in L excluding vertex 0. The process is again started to check if there exists a path from vertex j (now i is set to 1) to vertex j and if there exists a path from vertex j to vertex i. Do remove vertex j from list L if there is no path for n-2 times. After that, increment the value of i with one, or in notation: i = i + 1. And increment the lower range of j, so the range of j is j+1 <= new j <= n-1. Repeat this process until i = n-1.

Finally, we have a strongly connected component which includes vertex 0. But the goal is to find all strongly connected components in graph G. So, initialize a new list L’, L”,... for every vertex in G. These new lists contain all vertices in graph G and these vertices should not be the subset of vertices in the strongly connected component before. For example, if in list L contained {0,1,2}, then list L’ should be initiated from vertex 3 and L’ contains {3,4,5,...,n-1}.

Compared to the Kosaraju’s algorithm, this bruteforce approach has a higher time complexity. This bruteforce approach has a complexity of O(V^3). The pseudocode of this bruteforce approach is shown below:

```python
def BruteForce:
    for every vertex in graphG:
        if (vertex is not a member of previously found strongly connected components):
            #initialize list
            L = [vertices in graphG - vertices of previously found strongly connected components]
            idx = 0
            for i in range [idx..n-1]:
                for j in range[idx+1..n-1]:
                    check if there is a path from element[idx] to all elements[j]
                    check if there is a path from all elements[j] to element[idx].
                    if (no path):
                        remove element[j] from listL
```

From the complexity, it is reasonable for social medias to use Kosaraju’s algorithm to find strongly connected components instead of brute force algorithm which the execution time differs very significantly for large numbers of vertices in graph G. Assume 10^6 computations can be done in one second by computer. For inputs = 10^8 vertices and 10^8
edges, Kosaraju’s algorithm will return strongly connected components in just two seconds. Bruteforce algorithm will return strongly connected components in $10^{28} / 10^8$ seconds = 10^{19} seconds or in other words, 316,887,646 years!

**VII. KOSARAJU’S ALGORITHM IN EXPERIMENTS**

The real source code of Kosaraju’s algorithm is written in Python3 with source code from GeeksForGeeks website (cited in References) with some modification by user to increase the stack size for recursion. The test cases and results (with I/O time) are shown below:

Testcase 1: 9 vertices, 8 edges (taken from Fig. 6).
Result: Execution time: 0s. Image is shown below:

![Fig. 11 Testcase 1](image1)

Testcase 2: 900 vertices, 1800 edges.
Result: Execution time: 0.007s. Image is shown below:

![Fig. 12 Testcase 2](image2)

Testcase 3: 10000 vertices, 3000 edges.
Result: Execution time: 0.47s. Image is shown below:

![Fig. 13 Testcase 3](image3)

**CONCLUSION**

Kosaraju's algorithm has a complexity of $O(V+E)$, where the number of vertices in graph $G$ is denoted by $V$ and the number of edges in graph $G$ is denoted by $E$. Compared to bruteforce algorithm which has $O(V^3)$, Kosaraju's algorithm may be used in social medias which contain lot of users' data to determine users who are more closely related. This is useful when it comes to clustering-based sampling and recommendation to follow a user or a group of users.

The downside of Kosaraju's algorithm explained in this paper is the recursion-depth problem because system stack is used to store the memory registers of recursion function and the size of stack is limited. To avoid this problem, simply change the recursion-based algorithm to iterative-based algorithm utilizing a stack.

**ACKNOWLEDGMENT**

The author of this paper thanks God for His guidance and His knowledge. The author of this paper also wants to express his gratitude for Dr. Nur Ulfa Maulidewi, S.T., M.Sc as the lecturer for Algorithm Strategy for the knowledge shared in class and also the guidance which allows the author to finish this paper. Last but not least, for all the writers cited in the references, the author wants to thank them for their insights and fruitful knowledge and their willing to share it.

**REFERENCES**


**VIII. PERNYATAAN**

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 25 April 2019

Edward Alexander Jaya
13517115

Makalah IF2211 Strategi Algoritma, Semester II Tahun 2018/2019