Assignment -2

Advanced Algorithms (CPT_S 515)

Name: Coral Jain
WSU ID: 11632780

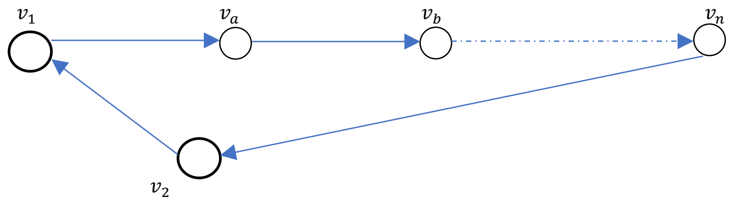
1. In Lesson 3, we talked about the Tarjan algorithm (SCC algorithm). Now, you are required to find an efficient algorithm to solve the following problem. Let G be a directed graph where every node is labeled with a color. Many nodes can share the same color. Let v_1,v_2,v_3 be three distinct nodes of the graph (while the graph may have many other nodes besides the three). I want to know whether the following items are all true: there is a walk α from v_1 tov_2 and a walk β from v_1 tov_3 such that

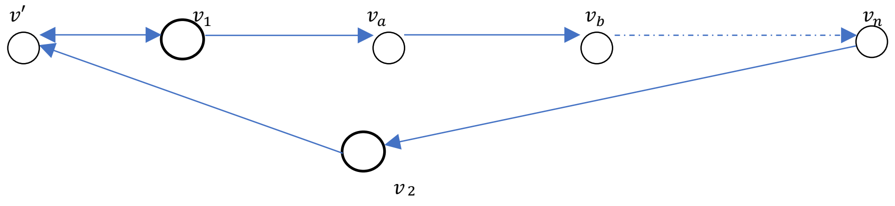
   - α is longer than β;
   - α contains only red nodes (excluding the two end nodes);
   - β contains only green nodes (excluding the two end nodes).

Solution:

In this question we are given a directed graph which consists of three nodes: There could a finite number of cases which we can consider which will inform us about the arrangement of and within the graph . These are mentioned as below:

Case 1: If the arrangement of the graph is such that v_2 and v_1 are connected in a way which takes v_2 back to v_1 or its ancestors in the given directed graph. Such two kinds of graphs will look like this:

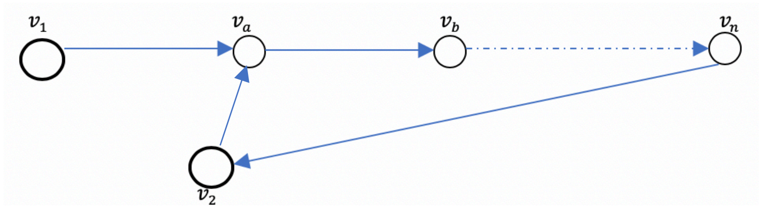




In the above graph, if we want to want to match the three conditions given in the question, we can keep a count of the number of edges that we come across while traversing through the graph and keep a counter which will give us the number of edges crossed and hence the length of the walks and .
Now in order to verify the second and the third condition we can use Tarjan's algorithm for finding the number of red nodes or green nodes that occur in between and . This can be done by using the DFS algorithm which is used within the Tarjan's algorithm. For example, if we take a walk and traversing from to , we may come across a number of nodes (say ,). Now, whether these nodes are a part of the walk between and is the question. Now with the help of Tarjan's algorithm we can justify that and are a part of same SCC and hence, the value of the low-link

value for and would be same (which is either the discovery time of   or its ancestors) i.e. low[] = low[] = disc[] or disc[] where could be any ancestor of The count of all such nodes can be stored in a counter. If, while traversing, the color of an edge comes out to be red, we can first check whether it is a part of the same SCC and then increase the counter by 1. If the condition is false, a 0 or 1 value can be returned.

Case 1: If the arrangement of the graph is such that   and are connected in a way that does not go back to or its ancestors in the given directed graph. However, it goes to a vertex or a number of vertices after the vertex in question while traversing through the graph. Such graph will look like this:



In the case we can use the simple DFS technique where we can go across each and every node and push it onto the stack. Every node visited would be checked for the color of the node and the counts of red and green node will be stored in two counters.

The algorithm can therefore be considered efficient because instead of using merely DFS for traversing through the graph and checking the condition, we can just check whether the colored nodes are a part of the SCC and if they do, they take back to . The condition for the length of and can be checked by storing the number of edges in counters as said above.

2. In Lesson 4, we learned network flow. In the problem, capacities on a graph are given constants (which are the algorithm's input, along with the graph itself). Now, suppose that we are interested in two edges e_1 and e_2 whose capacities c_1 and c_2 are not given but we only know these two variables are nonnegative and satisfying $c_1+c_2 < K$ where K is a given positive number (so the K is part of the algorithm's input). Under this setting, can you think of an efficient algorithm to solve network flow problem? This is a difficult problem.

Solution:

Method 1:

In this question, we are given the equation which says that c1 + c2 < k. In order to write an algorithm by varying the value of c1 and c2, we must use the process as said below:

C1: 1 to k-1

C2: 1 to k-1-c1

After this we can use Ford Fulkerson method and use it on the range as said above to find the maximum flow of the graph.

Therefore, we shall follow the following steps for carrying out this problem:

- First, we shall write a function for Ford Fulkerson algorithm:

  Here, f(e) is flow of the edge and E is the number of edges:

```
FF_Algorithm(c1, c2, Graph)

{

Initialize f(e) = 0 for every edge  in E

Initialize the residual capacity as original capacity

Repeat

Update residual graph

Find an augmenting path (with the help of BFS/DFS)

Augment flow until no augmenting path is found

Return max_flow_value
}
```

Now, since we need to use the above algorithm on the range of c1 and c2, we have to use another function inside which we can call the above function to receive the maximum flow from Ford Fulkerson algorithm. The value received is then compared to the other previously received value from the last iteration. The final output is the value received from the comparison. The other function looks something like this:

Here k is the given value of the constant given in the expression:

MaxFlow(k, Graph)

```
{
Initialise max_flow_value = 0
For c1 in 1 to k-1
        {
        For c2 in 1 to k-1-c1
        Initialise max_FF = 0
        max_FF = FF_Algorithm(c1, c2, Graph)
        {
        max_flow_value = (max_FF, max_flow_value)
        }
        }
}
```

Method 2:

For this question, you can also approach the problem through the lens of Linear programming because then we have efficient way to solve it since we will have efficiency or time complexity as linear as we will have a polynomial equation to solve the problem. Otherwise, if we go ahead with techniques like DFS we will have efficiency of exponential functions. Therefore, we can proceed with this step as it will be easier to go ahead, and time taken for execution would also be less. For this, we shall use the following steps in order to come up with a solution:

- First, we will consider a flow variable from one node to another and then consider that there are n nodes on which we need to apply linear programming method.
- Second, we can arrange the edges of the graph in the form of a matrix. We can consider these edges as x_{01}, x_{02} ... ... in the first row. This way we can arrange the edges in the form of a matrix.
- Now after this we shall identify the capacity constraint and the observational constraint for the given graph.

    Here, considering the capacities as y1, y2, ……

    We can write the capacity constraint as:

    (X01 < y1), (x02 < y2), (x03 < y3)………..

- We must keep in mind that we need the flow to be of maximum capacity. Now, considering that the flow that enters thorough sink S comes out of sink as T. Considering this property, we can define the conservation constant as:

  (x01 = x14+x15), (x03=x35), (x15+x25+x35=x57)

- We can now the inputs as:

  x01 - x14 - x15 = 0
  x02 – x24 – x25-x26 = 0
  .
  .
  .
  .

- From the above method, we can identify from the above equations that provides us an expression which will tell us the relationship between the inputs c1, c2 and above values.
- Since we already have an expression which says c1+c2 < k, we can say find the relationship easily.
- The values of the above variables can then be stored in a function thereafter.
- Finally, we can write a max_flow function to calculate the maximum flow for the given graph.

3. There are a lot of interesting problems concerning graph traversal — noticing that a program in an abstract form can be understood as a directed graph. Let G be a SCC, where v_0 is a designated initial node. In particular, each node in G is labeled with a color. I have the following property that I would like to know whether the graph satisfies:

   For each infinitely long path α starting from v_0, α passes a red node from which, there is an infinitely long path that passes a green node and after this green node, does not pass a yellow node.

   Please design an algorithm to check whether G satisfies the property.

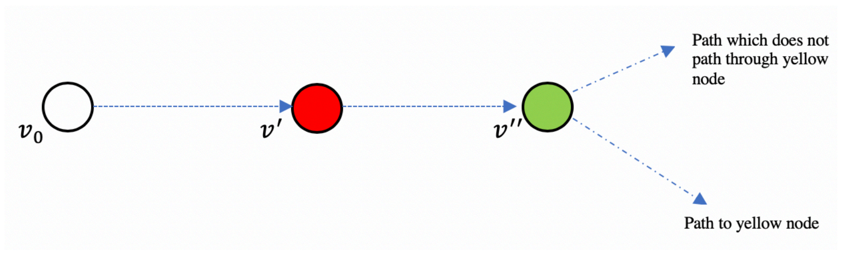
Solution:

The solution to this problem can be approached with two methods. In the first method, we can simply apply the Depth First Search technique in order to come up with an algorithm that can traverse over the graph and verify by using the conditions. This could be done in a number of steps.

DFS technique:

- First, since we are starting from node , we will go along an edge which emanates from . While applying DFS, we traverse along the edge to check whether the next node is a Red node or not. We shall continue the process until a red node is found.
- In the next step, we shall traverse along the edge which starts from the red node to verify whether the next node is a Green node. In this way, the two kinds of given colored nodes have been verified.
- Now, we take the next step to check whether we find a path which does not pass through a Yellow node. In this step also we use DFS technique.

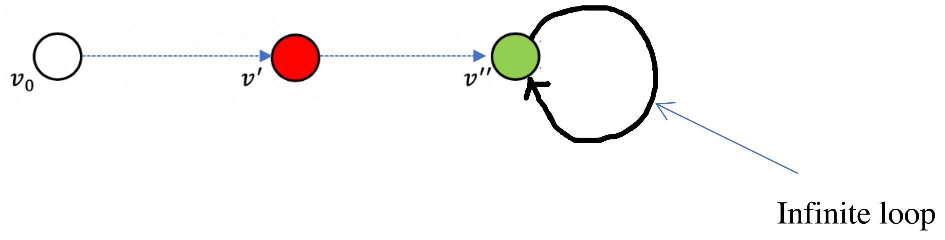
If we find a yellow node, however, we are not able to verify the property and therefore it is invalid for .



Shortest Connected Components technique:

In this second method, we shall apply the SCC technique in order to find out whether or not we are able to find the yellow node. The steps for this are as follows:

- Starting from the node we shall go along a path to find out whether or not we are able to find Red node. This will be done with the help of DFS.
- Now, similarly, like in DFS technique we shall apply DFS again to find out whether or not we are able to find a Green node.
- Now, we do not follow the DFS technique as before to locate Yellow node. Instead, we will try to create a graph which will be equivalent to but does not contain yellow nodes. As far as the programming is concerned, this can be done by removing the yellow nodes.
- Now we can apply the SCC algorithm on the newly constructed graph
- By applying the SCC algorithm, we can verify that whether there exists more than one SCC in the graph or the Green node is a self loop i.e. it runs within itself.



Time complexity:

Time complexity for this algorithm would be linear as the algorithm would run in linear time because of the traversal on nodes. Therefore, if is the input size. The time complexity would be .

4. Path counting forms a class of graph problems. Let G be a DAG where v and v' be two designated nodes. Again, each node is labeled with a color.

(1). Design an algorithm to obtain the number of paths from v to v' in G.

(2). A good path is one where the number of green nodes is greater than the number of yellow nodes. Design an algorithm to obtain the number of good paths from v to v' in G.

Solution:

{1}. For this part, we can approach simply with the DFS technique for obtaining the number of paths. We can carry out the following steps:

- Take as the start node for DFS and traverse it along the edges to reach up to the end node .
- We shall travel to every neighbor node of the given node and apply DFS to find a path to .
- Store the count of the number of paths that reached .
- We shall be using the following variables for the algorithm:

count_path:	count of number of paths to
current_node:	node being currently pointed

The **pseudo code** is given as below:

Function name:	num_paths_dfs()
Function call:	num_paths_dfs(count_path)

```
num_paths_dfs(current_nodecount_path)

{
    if current_node == NULL
    {
        return 0
    }


    if current_node ==
    {
        count_path++
        return count_path
    }
```

```
          for each  in current_node.edges
          {
                    neighbor_node = e.getOtherNode()
                    num_paths_dfs(neighbor,  count_path)

          }

}
```

Time complexity:

The time complexity of this algorithm would depend on the number of vertices of . If we consider number of vertices as , time complexity would be .

{2} For this part, we can approach similarly as above with the DFS technique for obtaining the number of paths. We can carry out the following steps:

- Take as the start node for DFS and traverse it along the edges to reach up to the end node .
- We shall travel to every neighbor node of the given node and apply DFS to find a path to .
- Store the count of the number of green nodes and yellow nodes in two different count variables that we come across while reaching .
- At the end, when we all the nodes have been visited, we shall verify whether the number of green nodes is greater than the number of yellow nodes (in order to count the number of good paths).
- We shall be using the following variables for the algorithm:

count_path:	count of number of paths to
current_node:	node being currently pointed
count_green:	number of green nodes in current path
count_yellow:	number of yellow nodes in current path
count_goodpaths:	number of good paths

The **pseudo code** is given as below:

```
Function name:      num_goodpaths_dfs()
Function call:      num_goodpaths_dfs(count_green, count_yellow,
            count_goodpaths)
```

```
num_goodpaths_dfs(current_nodecount_green, count_yellow,
count_goodpaths)

{
    if current_node == NULL
    {
        return 0
    }


    if current_node ==
    {

        if count_green > count_yellow
            count_goodpaths++
        return
    }

    for each  in current_node.edges

    {
        neighbor_node = e.getOtherNode()
        if current_node.color == “green”
            num_goodpaths_dfs(neighbor.nodecount_green+1,
            count_green, count_yellow,    count_goodpaths)
        elif current_node.color == “yellow”
            num_goodpaths_dfs(neighbor.nodecount_green,
            count_green, count_yellow+1,    count_goodpaths)

    }

}
```

Time complexity:

The time complexity of this algorithm also would depend on the number of vertices of . If we consider number of vertices as , time complexity would be .