# CPT_S 515: Homework #2

*Instructor: Zhe Dang*

**Sheng Guan**

# Problem 1

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 $\alpha$ from $v_1$ to $v_2$ and a walk $\beta$ from $v_1$ to $v_3$ such that

- $\alpha$ is longer than $\beta$;
- $\alpha$ contains only red nodes (excluding the two end nodes);
- $\beta$ contains only green nodes (excluding the two end nodes).

**Answer:**
For simplicity, we can assume the edge that connects two distinct nodes or self-loop edge on one node all is with the weight as 1 to indicate the distance. And I believe this assumption satisfies the requirement of this problem since this problem doesn't explicitly require us to consider the weight of the edge. The most difficult criteria to tackle is the criteria 1 which requires us to compare the walk length between $\alpha$ and $\beta$ Then for this criteria, we have two different scenarios:

- Scenario1. the walk $\alpha$ and walk $\beta$ both have no loops. In this case, the efficient algorithm for this problem is:
  Step 1: keep $v_1$ and $v_2$, only all red nodes in the graph to form a new graph $G_1$, similarly, keep $v_1$ and $v_3$, only all green nodes in the graph to form a new graph $G_1'$.
  Step 2: check whether $v_1$ in $G_1$ connects with $v_2$ by running the DFS algorithm starting from $v_1$, if no such path, the $\alpha$ doesn't exist, return false; if such path exists, we recursively run DFS from $v_1$ to $v_2$ using adjacency list representation to find all paths from $v_1$ to $v_2$. The detailed algorithm implementation can be found at Answer 4-(2). The key part here is we use path[] to store actual vertices and $path_{index}$ is current index in path[]. Then we select the path with the largest length, here we call it $l_1$.
  Step 3: similarly check whether $v_1$ in $G_1'$ connects with $v_3$ by running the BFS algorithm starting from $v_1$, if no such path, the $\beta$ doesn't exist, return false;if such path exists, since BFS discovers all vertices at distance k from $v_1$ before discovering any vertices at distance k+1. We here keep the distance when we identified $v_3$ and $v_1$. Then the length of $\beta$ equals to $l_1' = D(v_3) - D(v_1)$, this simple path from $v_1$ to $v_3$ also corresponds to a shortest path from $v_1$ to $v_3$ in $G_1'$.
  Step 4: compare whether $l_1 > l_1'$, if yes, return true; else, return false.

- Scenario2. the walk $\alpha$ contains loop. In this case, the efficient algorithm for this problem is:
  Step 1: keep $v_1$ and $v_2$, only all red nodes in the graph to run the SCC algorithm on $v_1 \rightarrow v_2$, if there's one or more strongly connected components founded with the size ¿1 or at least one strongly connected component is with the self loop, then potentially the walk $\alpha$ with infinite length exists. If not, return false since the $\alpha$ cannot be infinitely long in this case. Then the problem becomes we must be certain that $v_1$ can reach $v_2$ at this time.
  Step 2: Since in each strongly connected component, one node can always find a way to reach to all other nodes. We can symbolically treat each SCC as one big node (potentially save some scan and computation here but implementation detail is not given here, for example we can store the edge information across the boundaries when we identify SCCs, then we no longer need to do DFS within one SCC since we can treat one SCC as a very big node)and we run DFS to search whether $v_1$ can reach $v_2$ in this case . If $v_2$ can be reached, at this time, the walk of $\alpha$ can always become longer. Even if at this time walk $\beta$ also contains loop, we can fix a walk $\beta$ here we call it $\beta_1$. We are always able to increase the length of loop in $\alpha$ to make it longer than $\beta_1$, so here return true. If not, such $\alpha$ doesn't

exist, return false

# Problem 2

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.

**Answer:**

From $c1 + c2 < K$, we can get $c1 + c2 <= K - 1$

This problem differs from the traditional network flow problem is that for specific edges $e_1$ and $e_2$ we don't have the capacity bound for these two edges. Firstly I drop these two edges and find a network flow $F_1$ as the baseline Hence as a starting point , we can equally divide the $k - 1$ and assign them as the capacity bound for edge $e_1$ and $e_2$. Then we apply the traditional FordFulkerson algorithm to update the residual graph compute the augmentation path and adjust f until no augmenting path can be found. In this case I will get a $F_2$. I compute the percentage gain when comparing $F_1$ and $F_2$Then I would like to dynamically adjust the weight of $c_1$ and $c_2$ into two different directions at this time to apply the greedy update to find the new augmenting path if I have the chance, with the bound $|c_1| + |c_2| < K$. In this case I will get a $F_3$. I compute the percentage gain when comparing $F_1$ and $F_3$ One possible approach is I always try to use bi-partition in the given range to get the new capacity values. Then I try to find the new augmenting path until I cannot. The same approach I will get $F_4$, $F_5$, ..., and so on. When my percentage gain is convergent, I will stop the approach and say the current capacity allocation is optimal and solve the network flow problem.

# Problem 3

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 $\alpha$ starting from $v_0$, $\alpha$ 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.

**Answer:**

Because G is SCC that makes the things easier since each node can find a way to all other nodes. To facilitate illustration, we define that the last green node as $g_l$ that after $g_l$ the path can no longer have yellow node. We at the same time call our traversed red node as $r_t$. To get the traversed red node $r_t$, without loss generality, we can use a topological ordering to order all the green nodes in G and we will get $r_1$,$r_2$,...,$r_n$. Here the $r_n$ will be $r_t$. The reason we choose one representative $r_t$ is if the path traverses $r_t$ satisfies the property, then for any $r_i, i = 1, 2, ...n$, there definitely has corresponding path satisfies the property since $r_i$ connects with $r_t$ bi-direction. In short, we don't need to trace back to search all possible red nodes in the below recursively DFS based algorithm.

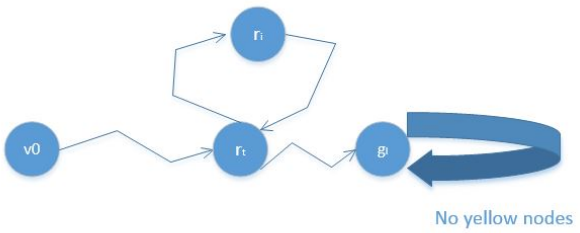
To my best understanding, this problem requires:

(1) $v_0$ connects to the $g_l$, this is trivial since the SCC guarantees this;

(2) there will be $r_t$ lies in the path from $v_0$ to $g_l$;

(3) There is not any yellow node behind $g_l$, and $g_l$ is one part of SCC with self-loop or $|SCC| > 1$.

Then naturally we will get the figure 3-1 as below:




Step1: We start with $v_0$ and use DFS to reach to one red node $r_t$ and push this node into the stack.
Step2: After we reach to $r_t$, we use DFS again to reach to green node $g_l$ and push this node into the stack.
Step3: Then starting from $g_l$, we need to make sure that $g_l$ falls into one SCC and this SCC contains no yellow nodes. Here we run the SCC algorithm after removing all yellow nodes. Then we locate the SCC which contains the green node $g_l$. If this SCC is a self-loop or $|SCC| > 1$, then all the properties satisfy, this path stays true and we move to the next path by tracking back according to the DFS algorithm; Otherwise, if SCC is not a self-loop and $|SCC| = 1$, the path is no longer with the infinite length, return false.
Step4: We restore all the deleted yellow nodes and track back according to the DFS algorithm and from $r_t$ to search all possible green nodes $g'_l$. The remaining step is as same as step2 and step3.
Step5: If all infinitely long path $\alpha$ remain true which satisfy all the above three properties, return true.

# Problem 4

4. Path counting forms a class of graph problems. Let G be a DAG where v and $v_0$ 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 $v_0$ 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 v0 in G.
**Answer:**

- (1). Here we try to determine the number of paths from v to $v_1$ using dynamic programming algorithm. We use a topological ordering to order $v'_1$,$v'_2$,...,$v'_n$ and we let $v'_1 = v$ and $v'_n = v_1$. The subproblems as follows: let $S[i]$ be the number of paths from $v'_i$ to $v'_n$. The solution to these subproblems is
$$S[i] = sum_{j \in adj(i)} S[j]$$
where $adj(i)$ is the set of all vertices v such that $(i, v) \in E$ The total time required to solve the O(V) subproblems is O(E), and the topological sort requires O(V +E) time, so the total running time is O(V + E).

- (2). Step 1: We can recursively run DFS algorithm on v to $v_0$ to obtain a list L of all paths using adjacency list representation from v to $v_0$., the pseudo code to achieve this is shown as Algorithm– Store all paths from a source to destination
Step 2: Then for each path $l_i$ we stored, we scan, count and compare the number of green nodes traversed vs. the number of yellow nodes traversed. If this path is a good path, we increase the number of good paths from v to $v_0$ by 1. In the end, we will get the total number of good paths. Of course, this step can be further combined into step 1. When we get and store the individual path information, we can do the green nodes and yellow nodes counting and comparison at that time. Then when we finish the all paths identification, we get the number of good paths at the same time.

**Algorithm 1** Store all paths from a source to destination

```
1: procedure MyProcedure
2:     #Initialize the vertices and a dictionary to store the graph, numOfGoodPath =0
3:     class Graph:
4:         def init(self, vertices) :
5:             self.V= vertices
6:             self.graph = defaultdict(list)
7:         def addEdge(self, u, v) :
8:             self.graph[u].append(v)
9:         "'A recursive function to collect all paths from 'u' to 'd'. visited[] keeps track of vertices in
   current path. path[] stores actual vertices and path_index is current index in path[]"'
10:        def collectAllPathsUtil(self, u, d, visited, path):
11:            visited[u]= True
12:            path.append(u)
13:            # If current vertex is same as destination, then this path meets the end, we need to
   compare the number of green and yellow nodes traversed
14:            if u==d:
15:                store the current path from v and v_0 into a list L
16:            "'If current vertex is not destination Recur for all the vertices adjacent to this vertex"'
17:            else:
18:                for i in self.graph[u]:
19:                    if visited[i]==False:
20:                        self.collectAllPathsUtil(i, d, visited, path)
21:            # Remove current vertex from path[] and mark it as unvisited
22:            path.pop()
23:            visited[u]= False
```

Refer to:
[1]http://www.geeksforgeeks.org/find-paths-given-source-destination/