Mark Shimozaki
Cpts 350
Hw 9

1. Design an efficient algorithm to count the number of paths from u to v.

Here is the algorithm written in python.

```
def count_paths(graph, u, v):
        count = [0] * len(graph)
        count[u] = 1
        for node in topological_order(graph):
            for neighbor in graph[node]:
                count[neighbor] += count[node]
        return count[v]

def topological_order(graph):
    visited = set()
    order = []

    def dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)
        order.append(node)
    for node in range(len(graph)):
        if node not in visited:
            dfs(node)
        return order[::-1]
graph = {
    0: [1,2]
    1: [3]
    2: [3]
    3: [4,5]
    4: []
    5: []
}
    u = 0  v = 4    print("# of paths from, u, 'to', v, ':',
                          count_paths(graph, u, v)
```

The algorithm has a time complexity of $O(V+E)$ where V is the # of vertices and E is the # of edges in the graph.

2. Design an efficient algorithm to count the number of good paths from U to V. also in python

```
def count_good_paths(graph, colors, u, v):
        green_count = [0] * len(graph)
        yellow_count = [0] * len(graph)
        for node in topological_order(graph):
            for neighbor in graph[node]:
            if    colors [neighbor] == 'green':
                    green_count[neighbor] += green_count[node] + 1
                    yellow_count[neighbor] += yellow_count[node]
            elif colors [neighbor] == 'yellow':
                    green_count [neighbor] += green_count [node]
                    yellow_count[neighbor]  += yellow_count[node] + 1
        good_paths = 0
        for node in range(len(graph)):
            if green_count[node] > yellow_count [node]:
                good_paths += 1
        return good_paths

def topological_order(graph):
        visited = set().
        order = []
    def dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)
        order.append(node)
    for node in range(len(graph)):
        if node not in visited:
            dfs(node)
    return order[::-1]
```

grap=

0	1 2
1	3
2	3
3	4 5
4	[]
5	[]

Colors =

0	green
1	Yellow
2	green
3	green
4	yellow
5	green

U = 0
V = 4

```
print(count_good_paths(
    graph, colors, U, V))
```

3. Design an efficient algorithm to count the number of ugly paths from u to v.

*

For each SCC that contains a cycle:

→ Check if the color sequence satisfies the regular expression γ

- If it does, return ∞.
- If no SCC with a cycle satisfies the regular expression γ, proceed to calculate the count of finite ugly paths
- Use probability 1 to count the # of paths that satisfy the regular expression γ from u to v.

define:

- $dp(u, c)$ as the number of paths from node u to v that end with color c and satisfy the regular expression γ.
- $next(x, c)$ as the set of nodes that can be reached from node u with an edge labeled with color c.

$$dp(u, c) = \begin{cases} 1 & \text{if } u = v \text{ and } c \text{ satisfies the regex } \gamma \\ 0 & \text{if } u = v \text{ and } c \text{ does not satisfy regex } \gamma \\ \sum_{next(u,c)} dp(u', c') & \text{if } u \neq v \text{ and } c \text{ satisfies regex } \gamma \\ 0 & \text{if } u \neq v \text{ and } c \text{ does not satisfy regex } \gamma \end{cases}$$

If there exist any infinite paths, the count is infinite; otherwise we use dynamic programming to calculate the count of finite paths.

4.

1. let A_1 and A_2 denote the adjacency matrices of the two graphs respectively

2. perrons theorem guarantees the existence of a unique largest real eigenvalue for non-negative irreducible matrices, which can represent strongly connected graphs

3. if $\lambda_1 > \lambda_2$, it suggests that, on average, there are more paths in the first graph compared to second graph

4. If $\lambda_1 < \lambda_2$, there are more paths in the second graph compared to the first graph.

if $\lambda_1 = \lambda_2$ both graphs have a similar number of paths

Graph 1

$A_1 =$ [[0,1,0],
[1,0,1],
[0,1,0]]

$A_2 =$ [[0,1,1],
[1,0,1],
[1,1,0]]

the perron eigenvalue is approx 1.414 for graph 1 and 2 for graph 2

5. Mini paper: Utilizing Control Flow Diagram for testing C-program

1. path Enumeration:

for each node u in the Control Flow diagram, its understood you can compute the total number of C(U) of paths from the root of the diagram to U, while some paths may be infinite due to loops in the programs, the count for each path can be obtained.

2. Maximal path:

By sorting the C(U)s for all v; you can identify a node U^* with maximal C(U). This node represents a critical point in the programs control flow, this is as it is reached by the largest number of paths from the root.

3. testing

The maximum path U^*, you can design targeted test cases to cover various Scenarios and branches around this node. By focusing our testing efforts on this crucial path.

6. $((0+(1+101)^*(11011))^*$,

let DP[i] denote the # of binary strings of length i that satisfy the given expression

$$DP[i] = DP[i-1] + DP[i-2] + DP[i-3] + DP[i-4]$$

The base cases are DP[0] = 1 (empty string) and DP[i] = 0 for i<0.

It has a time complexity of O(N),