Project: Structure Diversity

Zhicong Chen, 521120910256,
2028602614@qq.com

March 31, 2024

1 Introduction

Structure diversity of a node is the number of connected components of the induced graph, which consists of all neighbors of the node and edges between them. In this project, we are required to solve for structure diversity of each node in the graph. To solve this question, I applied two kinds of algorithms and implemented them with designed data structure. The outline of this report is given as below. In Section 2, I describe two algorithms, *Disjoint Set* and *Depth-First Search* and analyze their time complexity and space complexity. Implementation and applied data structure of two algorithms are given in Section 3. Finally, I report results of execution and analyze it in Section 4.

2 Algorithm

To finish the task, I did not exploit or design complicated algorithm, and instead I applied disjoint set and depth-first search, making some task-oriented improvements on them to improve performance. For analysis of complexity, the following notations will be used throughout the report.

- V_i : All the neighbors of node v_i
- E_i : The edges between all vertices in V_i
- $|V_i|$: The quantity of vertices in V_i
- $|E_i|$: The quantity of edges in E_i

2.1 Disjoint Set

Disjoint Set is a kind of data structure to figure out all connected components in a graph. It consists of a collection of disjoint sets, where the union of these disjoint sets forms all the points of the graph. Two core operations during construction of disjoint set are *union* and *find*.

For the basic version, we exploit list to perform *union* and *find*, and the time complexity is $O(N)$ and $O(1)$, respectively. But if we exploit *Union by Size* and *Path Compression*, both of their time complexity is $O(\log N)$ in the worst case. Consequently, when solving for vertex i , time complexity is $O(|E_i| \log |V_i|)$ and space complexity is $O(|V_i|)$ when building disjoint sets. Besides, to get all edges of $G_i(V_i, E_i)$, $O(\sum_{v_j \in V_i} |V_j|)$ time cost is also needed, since we need to traverse neighbors of all vertices in V_i. Given all above, solving for one node costs $O(\sum_{v_j \in V_i} |V_j| + |E_i| \log |V_i|)$ time. As for all nodes, we notice that $|V_i|$ is added in $|V_i|$ times as node i is in $|V_i|$ induced graphs $G_i(V_i, E_i)$ and therefore the final result is $O(\sum_{v_i \in V}(|V_i|^2 + |E_i| \log |V_i|))$ of time cost and $O(|E|)$ of space cost (notice that $\sum_{v_i \in V} |V_i| = |E|/2$). In the worst case the time complexity is $O(|V|^3 + |E| \log |V|)$.

2.2 DFS Tree

Depth first search (DFS) tree is a fundamental data structure for solving various problems in graphs. Following pseudo code illustrates the formation of DFS tree.

Algorithm 1: DFSTree(**G**,v)

```
Data: Graph $G(V, E)$ and root node $v(v \in V)$
Result: DFS Tree $T(V)$
DFS ($G$);
Function DFS($G$):
    for each vertex $v \in G$ do
        if $v$ is not visited then
            mark $v$ as visited;
            DFS(explore neighbors of $v$);
        end
    end
```

This website page [1] gives the following theorem and proof of it, which is fundamental for the algorithm.

Theorem 2.1. *In a DFS tree, endpoints of the edge not in the DFS tree have an ancestor-descendant relationship.*

According to Theorem 2.1, in a connected graph, when we delete root node of DFS Tree from the graph, the number of connected components equals to that of sons of root node in DFS Tree. Since subtrees rooted at its child nodes are disconnected from each other. We can perform DFS until all vertices have been visited and count marked edges starting with root node as its structure diversity.

When solving for node i, the time complexity is $O(|V_i| + \sum_{v_j \in V_i} |V_j|)$ and the space complexity is $O(|V_i|)$. For all nodes, the time complexity is $O(|V|^3)$ in the worst case and the space complexity is $O(|E|)$.

3 Implementation

Requirements for execution are listed as follows.

```
PyThon >=  3.7
numpy
time
```

3.1 Disjoint Set

Process of forming disjoint sets are encapsulated into class `DisjointSet`. To store *setsize* and *parent* information, I used Python `dict` for this. Hence, it costs $O(|V_i|)$ memory space. Function `StructureDiversityWithDJSet` solves for structure diversity question of input graph and you can exploit it in main function to test its performance.

3.2 DFS Tree

Process of forming disjoint sets are encapsulated into class `DFS`. I used a stack with space of $O(|V_i|)$ to store frontier vertices and a dict with space of $O(|V_i|)$ to record whether vertices have been visited and marked. The vertex will be marked if and only if its parent node is root node and is visited for the first time. The quantity of marked vertices is the number of sons of root node in DFS Tree, that is, its structure diversity.

4 Results

Time cost of each algorithm can be referred in the following table.

[1] https://blog.csdn.net/weixin_43848437/article/details/105133155

Algorithm	Time Cost(s)
Disjoint Set	0.070
DFS Tree	0.097

Table 1: Time cost of each algorithm.

By comparison, Disjoint Set performed well than DFS Tree in the test, which is inconsistent with my analysis of time complexity. I give some reasons for the result.

1. The input graph is sparse and the number of vertices and edges is not large enough to meet the upper bound of time complexity.
2. There exists more judge statements in DFS Tree which cost much time compared to other instructions.