EE1103 Quiz 1: Problem 2

Kaushik G. Iyer (EE23B135)

September 24, 2023

Abstract

In this programming assignment we were asked to solve the problem statement described below using different approaches and to additionally find the time taken to run our implementations.

Problem Statement

We are given N initially disconnected nodes (planets), then for the next M queries we are asked to:

1. Output 0, if the two planets have some path connecting them together.
2. Output 1, if the two planets have ***NO*** path connecting them together, then proceed to connect them together.

1 Grouping Method

1.1 Idea

Let us associate each set of connected nodes with a group (i.e. every planet in a given group can be visited from any other planet in the same group by some path).

```
int group[N]; // Where `group[i]` tells us which group the i'th planet belongs to
```

Now for each query we first find the group each planet belongs to, and then do the following:

1. If both planets are of the same group, they are already connected.
2. If the planets are of different groups, they are currently disconnected. Then proceed to merge both groups together.

1.2 Worst Case

Whenever the group of the planets in a query are different, we are forced to iterate over our list of groups (size=N). Therefore the worst case for our algorithm arises when all the queries lead to a new connection being made (i.e. when each query gives planets that are in different groups).

NOTE: The input submitted (`ee23b135_bad_roads.pdf`) follows this and is also the case that contains the most number of writes ($\frac{N*(N-1)}{2}$) times) to the array

1.3 Time Complexity

- Our program must iterate through all of the queries exactly once so it is obvious that the time complexity must be directly proportional to M.
- Notice that whenever a query gives two planets that are of different groups, we iterate over our list of size N (group array). Therefore the time complexity is directly proportional to N.

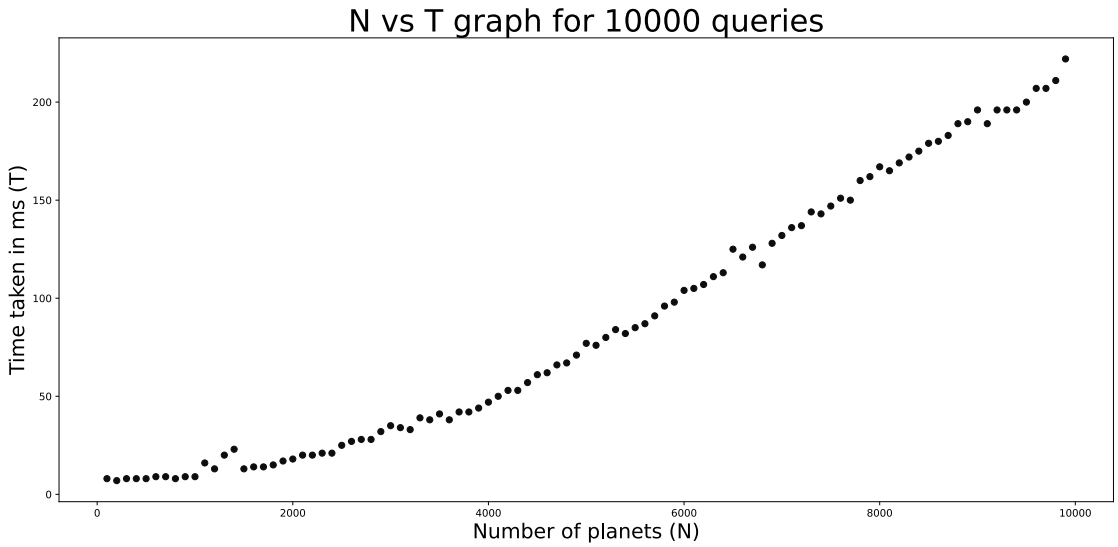

Figure 1: Linear dependence of N and T for `ee23b135_quiz2_q1.c`

Therefore we can conclude that the time complexity of this algorithm is $O(m \cdot n)$

---

2 Quick Weighted Union Find Method

2.1 Idea

The main pitfall of the previous method was the fact that when we had to merge two groups together, we were forced to go through the full list of N elements. Instead, let us define a tree type of structure (More of an inverted tree since we can only move up from the leaves to the root).

```
struct Node{
    int weight;
    struct Node* parent;
};
```

```
// We may also decide to store our data in two arrays
// This is the popular method
int weight[n];
int parent[n];
```

We can identify which 'group' a node is in by the 'root' of the tree it is of. Then we follow a logic similar to the first method:

1. If both planets have the same root, they are already connected.
2. If the planets have different roots, they are currently disconnected. Therefore we just set the parent of the smaller tree (The weight of a tree is calculated as the number of nodes in the tree) to the root of the big tree. (This is to reduce the depth of the resultant tree).

2.2 Worst Case

The worst case arises when our tree is the deepest (Since this leads to the most number of iterations while finding the root of a node). This arises when the queries are of disconnected planets that are the leaves (terminal points) of trees of the same weight. Notice that therefore, the maximum depth that our tree ever can be is $log_2 N$

2.3 Possible Optimizations

Notice that while finding the root of an element in a tree, we can *reparent* all nodes that we traversed through, to the root (i.e. Just set each node's parent directly to the root).

But also realize that since the deepest our tree ever gets is $\log_2(N)$, and the maximum n provided is 10^8 (Mentioned in the addendum), the maximum depth we get is atmost 27. Therefore the performance improvement we get by using this optimization is not that big (But it is still welcome).

2.4 Time Complexity

- Our program still has to iterate through all of the queries exactly once so the time complexity must be directly proportional to M.
- Recall that the number of elements we traverse through to get to the root from any element in any tree is atmost $log_2 N$. Therefore our time complexity is proportional to $log(N)$

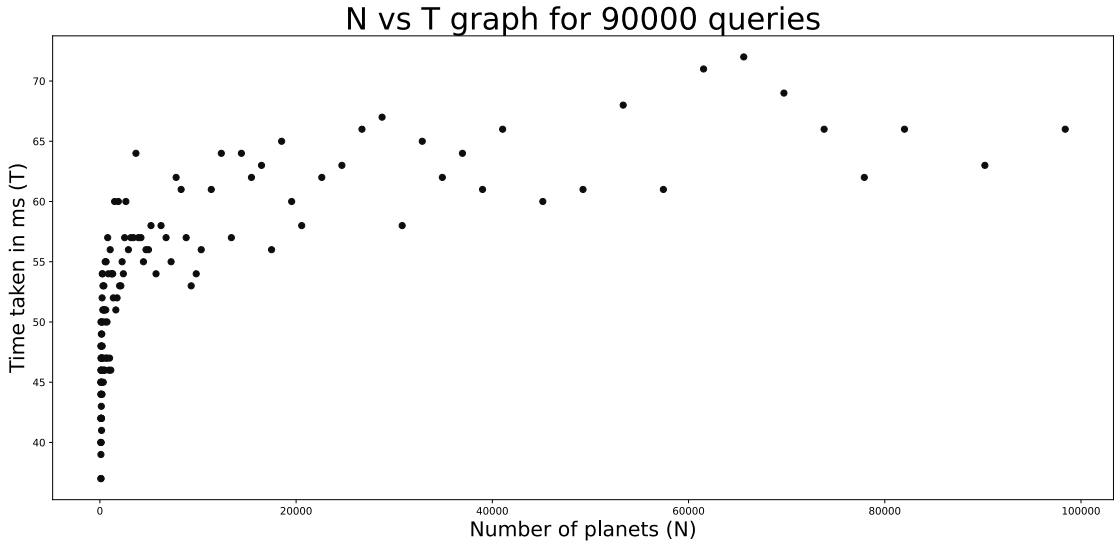

Figure 2: Logarithmic dependence of N and T for `ee23b135_quiz2_q3.c`

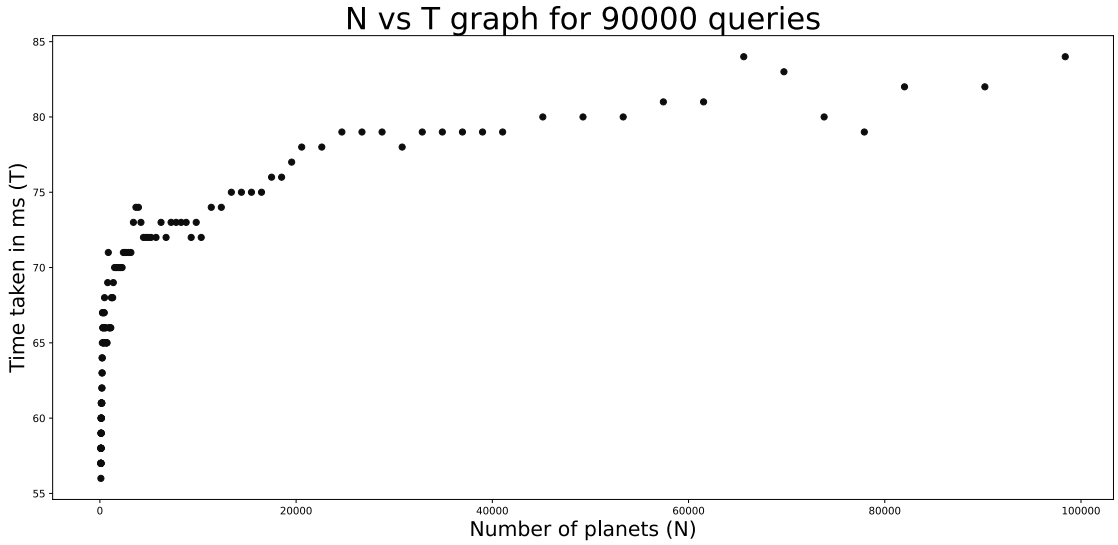

Figure 3: Logarithmic dependence of N and T for `ee23b135_quiz2_q4.c`

Therefore we can conclude that the time complexity of this algorithm is $O(m \cdot log(n))$