Dynamic Connectivity Using Union-Find

Introduction
The dynamic connectivity problem asks whether two objects in a set are connected by a series of pair connections. This problem is important in computer science because it models many real-world situations where components interact. In our project, we explore how the Union-Find (or Disjoint Set) data structure efficiently handles this problem. The goal of this assignment is to understand different algorithmic strategies for dynamic connectivity and to compare their performance. In particular, we study four variants:

- Quick-Find
- Quick-Union
- Weighted Quick-Union
- Weighted Quick-Union with Path Compression

Each of these methods improves on the previous one by reducing the time complexity of union and find operations, especially when processing large datasets.

Dynamic Connectivity

In dynamic connectivity, the "connectedness" between objects forms equivalence classes. Two objects are connected if there exists a sequence of unions linking them. The Union-Find data structure maintains these connected components with two primary operations:

- **Find(p):** Determines the root or representative of the component that object *p* belongs to.
- **Union(p, q):** Merges the sets containing objects *p* and *q*, if they are not already connected.

Overview of the Union-Find Variants

Quick-Find:

- Uses an array where each element directly stores its component id.
- The union operation involves scanning the entire array to update component ids, making it inefficient for large data sets.

Quick-Union:

- Represents components as trees where each element points to its parent.
- The find operation involves traversing up the tree until the root is found. Although union is faster than in Quick-Find, the tree can become tall, which slows down find operations.

Weighted Quick-Union:

- Improves Quick-Union by always attaching the smaller tree to the root of the larger tree.
- This balancing strategy keeps the trees more shallow, which speeds up subsequent find operations.

Weighted Quick-Union with Path Compression:

- Further optimizes the weighted approach by flattening the tree during the find operation.
- Path compression makes every node point directly to the root, ensuring that future operations are almost constant time.

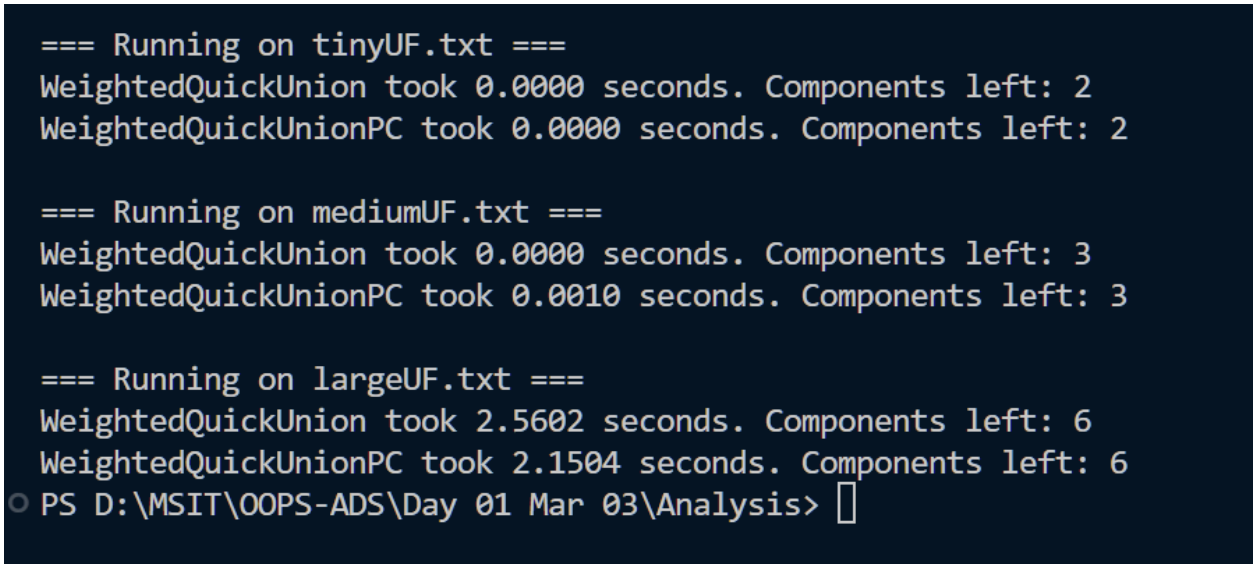
Results

Implementation	tinyUF.txt (s)	mediumUF.txt (s)	largeUF.txt (s)
Quick-Find	0.0000	0.0117	No output even after 5-6 hrs
Quick-Union	0.0000	0.0015	No output even after 5-6 hrs
Weighted Quick-Union	0.0000	0.0000	2.5602
Weighted Quick-Union with Path Compression	0.0000	0.0000	2.1504

```
QuickFind took 0.0000 seconds. Components left: 2
QuickUnion took 0.0000 seconds. Components left: 2
WeightedQuickUnion took 0.0000 seconds. Components left: 2
WeightedQuickUnionPC took 0.0000 seconds. Components left: 2

=== Running on mediumUF.txt ===
QuickFind took 0.0117 seconds. Components left: 3
QuickUnion took 0.0015 seconds. Components left: 3
WeightedQuickUnion took 0.0000 seconds. Components left: 3
WeightedQuickUnionPC took 0.0000 seconds. Components left: 3

=== Running on largeUF.txt ===
```



Findings/Discussion:

The experiment highlights how more advanced algorithms perform better on large datasets. It emphasizes the importance of optimization, where simple approaches like Quick-Find struggle due to inefficient union operations. Quick-Union improves efficiency with tree structures, but unbalanced trees can still slow down operations. Weighted Quick-Union and path compression further enhance performance by balancing the trees and flattening the structure, resulting in nearly constant-time operations. This supports the idea that a well-optimized algorithm, rather than powerful hardware, is key to solving complex problems efficiently.

Quick-Find and Quick-FindUnion Takes unknown time for larger data

WeightedQuickUnion, Weighted Quick-Union with Path Compression is more efficient for any size of data

Conclusion

In this assignment, we explored four different implementations of the Union-Find data structure. By running these implementations on datasets of various sizes, we demonstrated that algorithmic improvements significantly affect performance. The progression from Quick-Find to Weighted Quick-Union with Path Compression clearly shows that careful algorithm design is more impactful than relying on faster hardware alone. This project reinforces the idea that in computational tasks, a well-designed algorithm is key to efficient problem solving.