Project Title

Dynamic Connectivity Using Union-Find

Overview

This project explores the *dynamic connectivity problem*—determining whether any two objects in a given set are connected through a sequence of pair connections. The relation “is connected to” is treated as an equivalence relation (symmetric, transitive, reflexive), which naturally partitions objects into connected components.

The **Union-Find** (also called **Disjoint Set**) data structure provides an efficient way to solve this problem by supporting two key operations:

1. **Find(p)**: Determine the component (or “root”) to which object *p* belongs.
2. **Union(p, q)**: Merge the components containing *p* and *q* if they are not already connected.

In this project, you will implement and compare four Union-Find algorithms:

1. **Quick-Find**
2. **Quick-Union**
3. **Weighted Quick-Union**
4. **Weighted Quick-Union with Path Compression**

By running each implementation on different input files (tinyUF.txt, mediumUF.txt, largeUF.txt), you will measure their performance and observe how algorithmic optimizations can drastically reduce running times—even more effectively than simply using faster hardware.

Objectives

1. **Understand Dynamic Connectivity**
    - Recognize how “connectedness” forms equivalence classes.
    - Appreciate how the Union-Find data structure manages these connections efficiently.
2. **Familiarize with the Union-Find API**
    - Learn how `union(p, q)` and `find(p)` are used in practical scenarios.
    - Understand the differences between the four algorithmic variations.
3. **Compare Algorithmic Performance**

   - Execute all four Union-Find implementations on the three data sets.
   - Record running times and compare the results across different algorithms.
   - Demonstrate that more sophisticated algorithms (like weighted union and path compression) outperform naive approaches (quick-find, quick-union), especially at scale.

4. **Report Findings**

   - Document execution times with screenshots or console logs as proof.
   - Interpret the results to support the statement:
     “Supercomputer won't help much; good algorithm enables solution.”
   - Conclude how algorithmic efficiency is often more critical than raw computing power.

Project Tasks

1. **Review the Provided Files**

   - **tinyUF.txt**: A small dataset with 11 connections (for initial testing and debugging).
   - **mediumUF.txt**: A moderate dataset with ~900 connections.
   - **largeUF.txt**: A dataset with millions of connections, designed to stress-test the implementations.

2. **Implement the Union-Find Variants**

   - **Quick-Find**: Maintain an array where the index represents the object, and the value is the identifier of its connected component.
   - **Quick-Union**: Represent components as a tree structure. Each index points to its parent, and roots represent component identifiers.
   - **Weighted Quick-Union**: Improve quick-union by always attaching the smaller tree to the larger one, balancing the tree height.
   - **Weighted Quick-Union with Path Compression**: Further optimize by flattening the tree whenever you perform a `find` operation.

3. **Compile and Run**

   - For each of the four implementations, run your code on all three input files.
   - Measure the execution time for each run (e.g., using `System.currentTimeMillis()` in Java or a similar timing mechanism).

4. **Analyze Results**

- Create a simple table or chart comparing the running times.
- Include screenshots or logs of your console output for proof of execution.
- Discuss how each optimization (weighted union, path compression) affects performance.

5. **Write a Short Report**

   - **Introduction**: Summarize the dynamic connectivity problem and the purpose of the assignment.
   - **Methodology**: Briefly describe each Union-Find implementation approach.
   - **Results**: Present your timing data in a table/chart.
   - **Discussion**: Interpret the data to illustrate why a better algorithm scales more effectively than simply relying on faster hardware.
   - **Conclusion**: Reiterate how Weighted Quick-Union with Path Compression typically offers the best performance and how algorithmic improvements are crucial in handling large-scale problems.

Deliverables

1. **Source Code**: All four implementations of the Union-Find data structure.
2. **Execution Proof**: Screenshots/logs demonstrating run times for each implementation on each dataset.
3. **Report**: A concise write-up (PDF or similar) that:
   - Introduces the dynamic connectivity problem.
   - Describes the Union-Find variants.
   - Shows and explains the performance results.
   - Concludes with the key lesson that good algorithms can outperform brute force or simpler methods—even on modest hardware.

Conclusion

By completing this project, you will gain practical insight into how algorithmic optimizations can drastically affect program efficiency, especially in large-scale applications. The progression from **Quick-Find** to **Weighted Quick-Union with Path Compression** demonstrates the power of data structure refinements and underpins the statement:

> **“Supercomputer won't help much; good algorithm enables solution.”**