Performance Analysis of Union-Find Implementations

Arivoli Ramamoothy EE23B008

September 24, 2023

1 Introduction

In this report, we analyze the performance characteristics of three different Union-Find implementations: Quick Union-Find (q1), Weighted Quick Union-Find with Path Compression (q3), and Weighted Quick Union-Find without Path Compression (q4). These implementations are essential for solving problems involving disjoint sets, such as connecting realms in a world-building scenario.

2 Performance Analysis

2.1 Quick Union-Find (q1)

– No path compression or weighted union.

City Size	Input Size	Runtime (ms)
100 cities	100 requests	0 ms
10^5 cities	10^5 requests	938 ms
10^3 cities	10^7 requests	6544 ms

Table 1: Runtime Measurements for q1

2.2 Weighted Quick Union-Find with Path Compression (q3)

– Utilizes both weighted union and path compression techniques.

City Size	Input Size	Runtime (ms)
100 cities	100 requests	0 ms
10^5 cities	10^5 requests	16 ms
10^3 cities	10^7 requests	1163 ms

Table 2: Runtime Measurements for q3

2.3 Weighted Quick Union-Find without Path Compression (q4)

– Utilizes weighted union but lacks path compression.

City Size	Input Size	Runtime (ms)
100 cities	100 requests	0 ms
10^5 cities	10^5 requests	20 ms
10^3 cities	10^7 requests	1310 ms

Table 3: Runtime Measurements for q4

3 Implementation Details

3.1 Quick Union-Find (q1) Implementation

In the q1 implementation, we use a simple structure with arrays to represent sets.

3.1.1 Functions

– `initialize(int n, int *parent)`: Initializes the sets.

– `find(int x, int *parent)`: Finds the root of a set.

– `connect(int x, int y, int *parent)`: Connects two realms and returns 1 if a road is built, 0 otherwise.

3.2 Weighted Quick Union-Find with Path Compression (q3) Implementation

In the q3 implementation, we use the `DisjointSet` struct to represent sets, and we employ path compression and weighting for efficient operations.

3.2.1 Struct Definition

The `DisjointSet` struct consists of the following fields:

– `int *parent`: An array to store the parent of each element.

– `int *size`: An array to store the size of each set.

3.2.2 Functions

We define the following functions:

– `initialize(int n, DisjointSet* set)`: Initializes the disjoint set data structure with `n` elements.

- `find(int x, DisjointSet* set)`: Finds the root of the set to which element `x` belongs with path compression.
- `connect(int x, int y, DisjointSet* set)`: Connects two elements `x` and `y` if they belong to different sets and returns 1 if a road is built, 0 otherwise.

3.3 Weighted Quick Union-Find without Path Compression (q4) Implementation

In the q4 implementation, we also use the `DisjointSet` struct to represent sets, but we do not use path compression.

3.3.1 Struct Definition

The `DisjointSet` struct has the following fields:

- `int *parent`: An array to store the parent of each element.
- `int *weight`: An array to store the weight (size) of each set.

3.3.2 Functions

We define the following functions:

- `initialize(int n, DisjointSet* set)`: Initializes the disjoint set data structure with `n` elements.
- `find(int x, DisjointSet* set)`: Finds the root of the set to which element `x` belongs without path compression.
- `connect(int x, int y, DisjointSet* set)`: Connects two elements `x` and `y` if they belong to different sets and returns 1 if a road is built, 0 otherwise.

4 Worst-Case Scenario of q1 (q2)

Explanation of q2 assuming number of cities = 2^n:

- In this input configuration, we systematically create pairs of cities with union operations. We start with $2^{(n-1)}$ pairs of adjacent cities (2^n being the total number of cities), and in each union operation, we combine two pairs into one. This process continues until we have only one pair remaining. As a result, we effectively reduce the number of pairs by half in each step, creating a binary tree-like structure. This arrangement leads to a chain of connections where each city points to its immediate neighbor, making find operations in q1 highly inefficient and resulting in a time complexity that grows linearly with the number of cities. This input configuration highlights the worst-case scenario for q1 when dealing with long chains of connections.

5 Summary

In summary:

- q1 is the slowest among the three implementations due to the lack of both path compression and weighted union.
- q4 is faster than q1 due to weighted union but can still be slow for large datasets and repeated operations.
- q3 is the fastest among the three, thanks to both path compression and weighted union. It offers efficient operations for a large number