COMS4032A: Applications of Algorithms – Assignment 3

Nihal Ranchod
2427378
BSc Hons

October 16, 2024

Contents

1 Introduction

This assignment examines the performance of distinct operations within disjoint sets data structures from Chapter 12 of the 3rd Edition of *Introduction to Algorithms* by Thomas H et al. 2009. Part A of this assignment assess the performance of the following three Union operations on Disjoint Set data structures:

1. Basic Union
2. Weighted-Union Heuristic
3. Union-by-Rank and Path Compression

The goal is to empirically validate Theorem 21.1 from the textbook, which states:

> Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m Make-Set, Union, and Find-Set operations, n of which are Make-Set operations, takes $\mathcal{O}(m + n \log n)$ time.

The experiments assess the efficiency of each union method across varying set sizes, providing insights into the performance benefits of optimization techniques like weighting and path compression.

All C++ source code and results of the experiments are available at the following GitHub Repository.

2 Theorem 21.1: Empirical Validation

This section outlines the experimental setup used to empirically validate Theorem 21.1 by recording execution times for different union operations, observing their scalability over a range of disjoint set sizes.

2.1 Experimental Setup

1. **Union Operations:**
   - Three different Union operations are tested: Basic Union, Weighted-Union, and Union-by-Rank with Path Compression.
   - Each method is evaluated based on its execution time, with a specific focus on how each handles the union of elements across varying data sizes.
2. **Setup:**

   The experiments evaluated the three Union methods over various sizes of sets, ranging from 100 elements to 10,000,000 elements.
   - The code was implemented in C++ for efficiency and used the `<chrono>` library to measure execution times with high precision.
   - The results were saved to two CSV files:
     - `disjoint_set_experiment.csv`: Detailed per-run results.
     - `disjoint_set_average_results.csv`: Aggregated average times for each set size.

3. **Experimental Procedure:**

   For each Union method and each set size, the following steps were performed:

   (a) **Initialisation:**
   - Created n elements as separate sets.

   (b) **Union Operation:**
   - Sequentially merged each element with the first element to form a single set containing all elements.
   - Recorded the time taken for all merge operations to complete.

   (c) **Repeating and Averaging:**
   - The experiment was repeated 5 times for each set size.
   - Averaged the execution times for Union, Weighted-Union, and Union-by-Rank operations across the runs.

3 Algorithmic Implementation

3.1 Basic Union

1. This serves as a baseline implementation. The algorithm directly connects the elements of set to another without optimisation.
2. **Time Complexity:** Since it lacks any heuristic, the operation can degrade to $\mathcal{O}(n)$ for each union in the worst case, making it inefficient for large sets.

3.2 Weighted-Union

1. This approach attaches the smaller set to the larger set Thomas H et al. 2009, minimising the resulting structure's height through the use of a `size` attribute to easily maintain the length of the list.
2. **Time Complexity:** With the weighted-union heuristic, the operation achieves $\mathcal{O}(\log n)$ complexity for unions, aligning with Theorem 21.1.

3.3 Union-by-Rank with Path Compression

1. This method combines the Union-by-Rank heuristic (attaching the shallower tree under the deeper one) with path compression, which shortens paths during Find-Set operations.
2. **Time Complexity:** With path compression, Find-Set operations achieves $\mathcal{O}(\alpha(n))$ time complexity, where α is a function which grows extremely slowly.

4 Results

4.1 Basic Union Execution Times

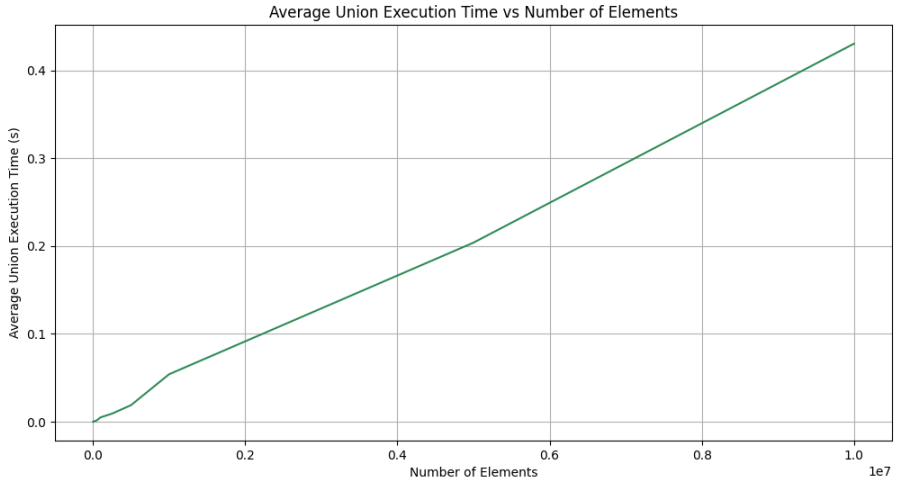

Figure 1: Average Execution Times for Basic Union Operation over increasing n, where n is the number of elements.

4.2 Weighted-Union Heuristic Execution Times

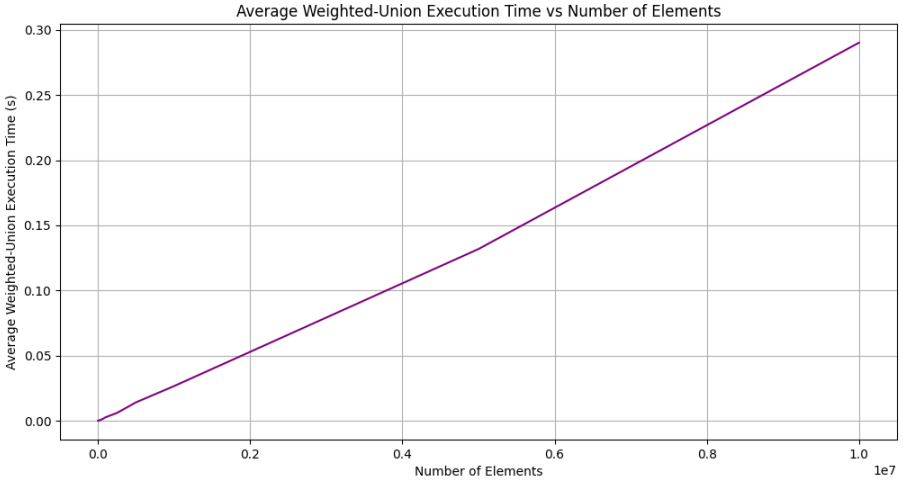

Figure 2: Average Execution Times for Weighted-Union Operation over increasing n, where n is the number of elements.

4.3 Union-by-Rank and Path Compression Heuristics Execution Times

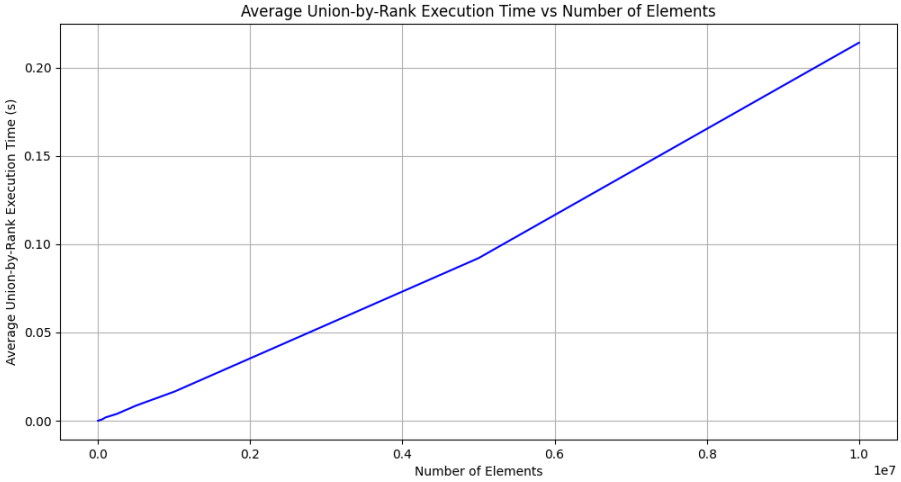

Figure 3: Average Execution Times for Union-by-Rank and Path Compression Heuristics Operations over increasing n, where n is the number of elements.

4.4 Combined Graphs

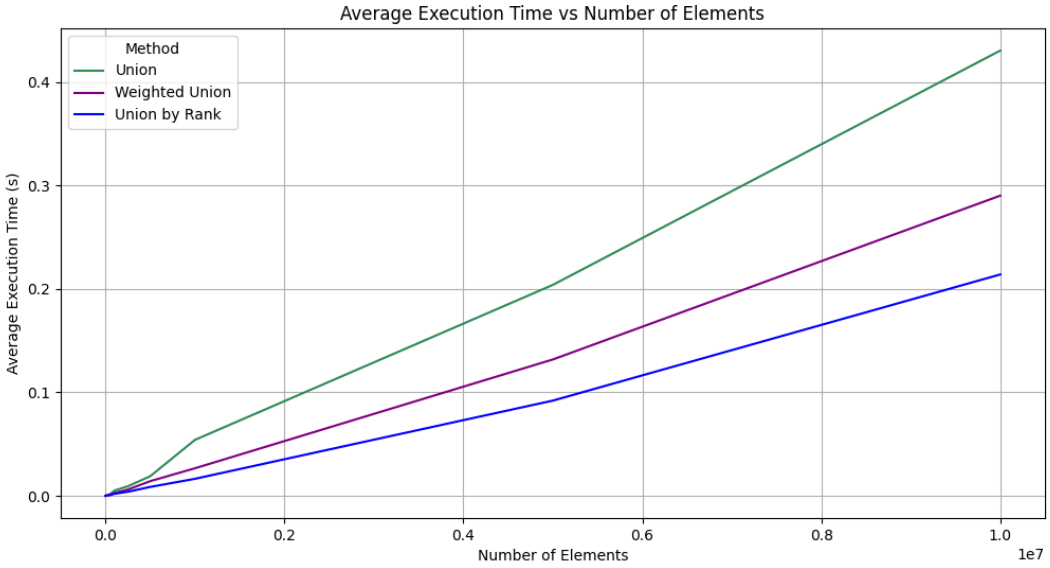

Figure 4: Average Execution Times for Increasing n)

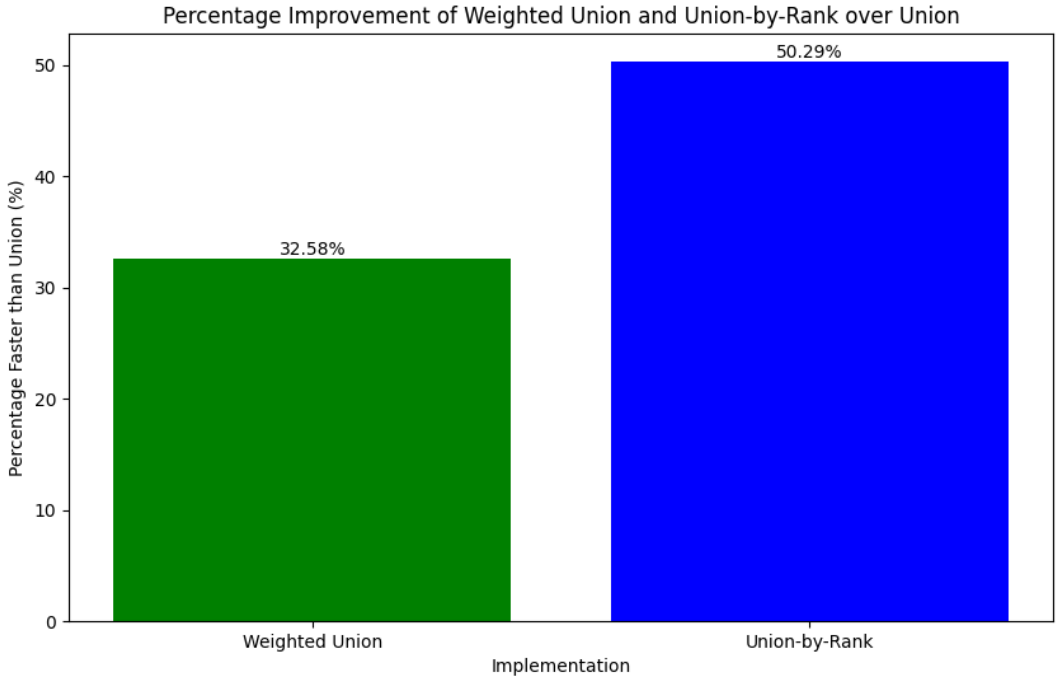

Figure 5: Percentage Improvement of Weighted-Union and Union-by-Rank over Basic Union

5 Analysis

Figure 4 shows the average execution time for Union operations across different disjoint set sizes, using three union strategies: **Basic Union**, **Weighted- Union**, and **Union-by-Rank with Path Compression**. The results reveal the following key observations:

5.1 Analysis of Average Execution Times vs. Number of Elements

1. **Basic Union** consistently exhibits the highest execution time as the number of elements increases. This result is expected since Basic Union lacks any optimization heuristic, resulting in a linear increase in execution time due to the lack of balancing mechanisms.

2. **Weighted-Union** performs noticeably better than Basic Union, demonstrating lower execution times across all tested set sizes. This improvement aligns with Theorem 21.1, which states that the weighted-union heuristic significantly optimizes performance by minimising the growth of the data structure's height. The near-logarithmic growth pattern of the Weighted Union curve supports the theorem's assertion that the complexity is $\mathcal{O}(m+n\log n)$, where the added heuristic provides considerable improvements in runtime efficiency.

3. **Union-by-Rank with Path Compression** achieves the lowest execution times among the three implementations. The combination of path compression with the rank heuristic results in a nearly flat execution time curve for larger inputs, indicative of an almost constant time complexity for the Find-Set operations. This matches the theoretical

expectations, as path compression drastically reduces the depth of the trees, resulting in further efficiency gains over Weighted-Union alone. Notably, these empirical results provide strong evidence for the claim made on page 572 of Thomas H et al. 2009, that the worst-case runtime for Union operations with Union-by-Rank and Path Compression is $\mathcal{O}(m\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. This function grows very slowly, making $\alpha(n)$ a tiny constant for any practical input size.

5.2 Analysis of Percentage Improvement

Figure 5 illustrates the percentage improvement in execution time of both Weighted-Union and Union-by-Rank relative to Basic Union:

1. **Weighted-Union** 32.58% improvement over Basic Union. This substantial gain demonstrates the effectiveness of the weighted heuristic in reducing unnecessary merging operations and thereby limiting the structure's overall height. The result directly supports Theorem 21.1, which attributes the efficiency of weighted union to its balanced structure.
2. **Union-by-Rank with Path Compression** shows an even greater improvement, at 50.29% over Basic Union. This significant increase underscores the impact of path compression in conjunction with rank balancing. Path compression ensures that repeated Find-Set operations are faster, as elements quickly point directly to their set's representative. This improvement aligns with the theoretical enhancements provided by path compression and ranks beyond just the weighted heuristic.
3. **Implications for Large Data:** The increasing efficiency gap as the set size grows reinforces the expected complexity bounds. For large datasets, optimizations like path compression and rank are critical to achieving scalable performance. The empirical results here demonstrate how these advanced heuristics outperform simpler methods by providing better time complexity in practice

6 Solution to Exercise 21.3-4

To tackle Exercise 21.3-4, we can enhance the disjoint-set forest implementation by adding a single attribute to each node that enables efficient printing of all members in the same set. This is achieved by adding a '`next`' pointer to each node, which points to the next member in the set. This allows us to traverse the members of a set in linear time when executing the '`Print-Set`' operation.

6.1 Node Structure

Each node will have the following attributes:

- **parent**: points to the parent node
- **next**: points to the next node in the set

6.2 Implementation Details

- Each node will have a '`next`' pointer.
- The root of each set will point to the first node in the list.
- The last node in the list will point back to the root, forming a circular linked list for each set.

6.3 Modifying Existing Operations

- **Make-Set(x)**: Set `x.next = x` (the node points to itself).
- **Union(x, y)**: After linking the roots, we need to merge the linked lists:
  1. Save `x_root.next` and `y_root.next`.
  2. Set `x_root.next = y_root.next`.
  3. Set `y_root.next = x_root.next`.
  4. Update the last node of each list to point to the new root.
- **Find-Set(x)**: No changes are needed for this operation.

6.4 New Operation: Print-Set(x)

To print all members of the set containing node `x`:

Algorithm 1 Print-Set(x)

```
y ← Find-Set(x)                ▷ Find the root of the set containing x
print y
z ← y.next
while z ≠ y do
    print z
    z ← z.next
end while
```

6.5 Performance Analysis

- The `Print-Set` operation traverses the members using the '`next`' pointers, leading to a linear time complexity $\mathcal{O}(k)$, where k is the number of members in the set.
- The asymptotic running times for `Make-Set`, `Union`, and `Find-Set` operations remain unchanged.

6.6 Conclusion

This solution meets all the requirements:

- Adds only one attribute ('`next`') to each node.
- `Print-Set` takes linear time in the number of set members.
- Other operations maintain their asymptotic running times.
- Each member can be printed in $\mathcal{O}(1)$ time.

References

Thomas H, Cormen et al. (2009). *Introduction to Algorithms Third Edition*.