WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4032A: Applications of Algorithms – Assignment 3

Nihal Ranchod
2427378
BSc Hons

October 21, 2024

Contents

1 Introduction 3

2 Theorem 21.1: Empirical Validation 3

2.1 Experimental Setup . . . 3

3 Algorithmic Implementation 4

3.1 Basic Union . . . 4

3.2 Weighted-Union . . . 4

3.3 Union-by-Rank with Path Compression . . . 5

4 Validation Check 5

5 Results 6

5.1 Basic Union Execution Times . . . 6

5.2 Weighted-Union Heuristic Execution Times . . . 6

5.3 Union-by-Rank and Path Compression Heuristics Execution Times . . . 7

5.4 Combined Graphs . . . 7

6 Analysis 8

6.1 Analysis of Average Execution Times vs. Number of Elements . . . 8

6.2 Performance Comparison and Efficiency Gains 9

7 Solution to Exercise 21.3-4 9

7.1 Node Structure 10

7.2 Implementation Details 10

7.3 Modifying Existing Operations 10

7.4 New Operation: Print-Set(x) 10

7.5 Performance Analysis 11

7.6 Conclusion 11

1 Introduction

This assignment examines the performance of distinct operations within disjoint sets data structures from Chapter 21 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 100000 elements.
   - The code was implemented in C++ for efficiency and used the `<chrono>` library to measure execution times with high precision.

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) **Validation:**
      - After each union operation, a validation check is performed to ensure that all elements are correctly united into a single set.

   (d) **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 performs a simple union of two disjoint sets by connecting all elements in both sets under a single representative element, following a brute-force approach.
   - **Find Set Representatives**: The FindSet function is used to identify the root (representative) of each set by traversing the parent pointers until the root is found.
   - **Reassign Parent Pointers**: For both sets, all elements have their parent pointers reassigned to the root of the first set. This ensures that all elements in both sets now share a common root, even if they already belonged to the same set.
   - **Set Merging**: The elements of the second set are appended to the first set by linking the next pointer of the last element in the first set to the head of the second set. The process iterates through the entire set, updating ownership and parent pointers for every element in the merged set.
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. The Weighted Union operation improves efficiency by ensuring that the smaller set is always merged into the larger one Thomas H et al. 2009. This reduces the depth of trees formed during the union operation, leading to faster lookups.
   - **Find Set Representatives**: Similar to the Basic Union, the FindSet function is used to identify the root of both sets.

  - **Compare Set Sizes**: The sizes of the two sets are compared. The set with the smaller size is merged into the larger one. This ensures that the resulting tree remains as shallow as possible.
  - **Reassign Parent and Owner Pointers**: The elements of the smaller set are iterated through, and their parent pointers are updated to point to the root of the larger set. Additionally, the ownership of these elements is updated to reflect their membership in the larger set.
  - **Merge Lists**: The tail of the larger set is linked to the head of the smaller set, merging the two sets into one. The size of the larger set is updated to include the number of elements in the smaller set.

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. The Union-by-Rank operation further optimizes the union process by using the rank to decide which set should be merged into the other. This technique reduces the height of the trees and ensures better performance for future operations.
   (a) **Find Set Representatives**: Using the 'FindSetPathCompression' function, the representatives of both sets are found. Path compression is employed during this process to flatten the tree structure, improving future lookups.
   (b) **Compare Ranks**: The rank of the root elements is compared. The set with the smaller rank is merged into the set with the larger rank to ensure the tree remains balanced. If the ranks are equal, one of the roots is arbitrarily chosen as the new root, and its rank is incremented.
   (c) **Reassign Parent Pointers**: The parent pointer of the smaller-ranked root is updated to point to the larger-ranked root, effectively merging the two sets under one common root.
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 Validation Check

The Validation Check ensures the correctness of the union operations by verifying that all elements in the dataset belong to the same set after the union operations have been performed. This is done by:

1. **Selecting a Representative**: The FindSet function is used to find the representative element of the first element in the set.
2. **Checking Consistency**: All other elements are iterated through, and the FindSet function is called on each element. If any element's representative is different from the initially selected representative, the validation fails, indicating that the union operation did not correctly merge the sets.

This validation step is crucial for confirming that the union operations successfully merged all the elements into a single set, as intended.

5 Results

5.1 Basic Union Execution Times

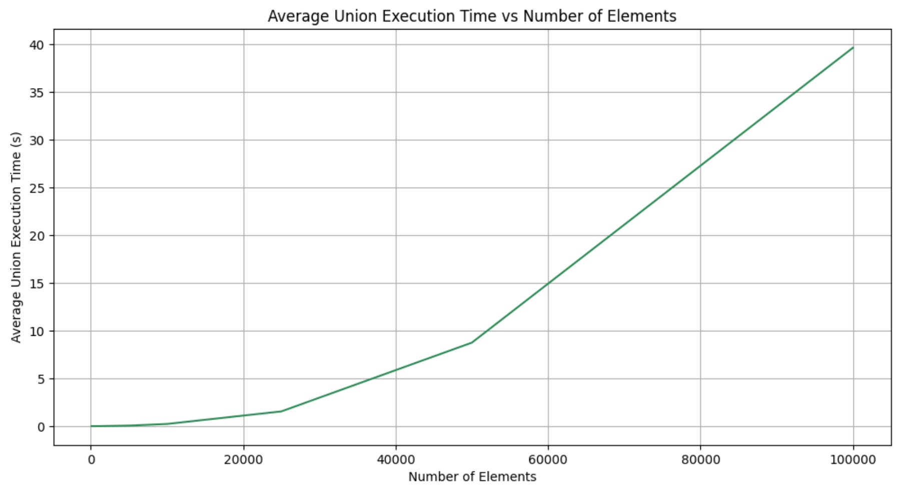


Figure 1: Average Execution Times for Basic Union Operation over increasing n, where n is the number of elements.

5.2 Weighted-Union Heuristic Execution Times

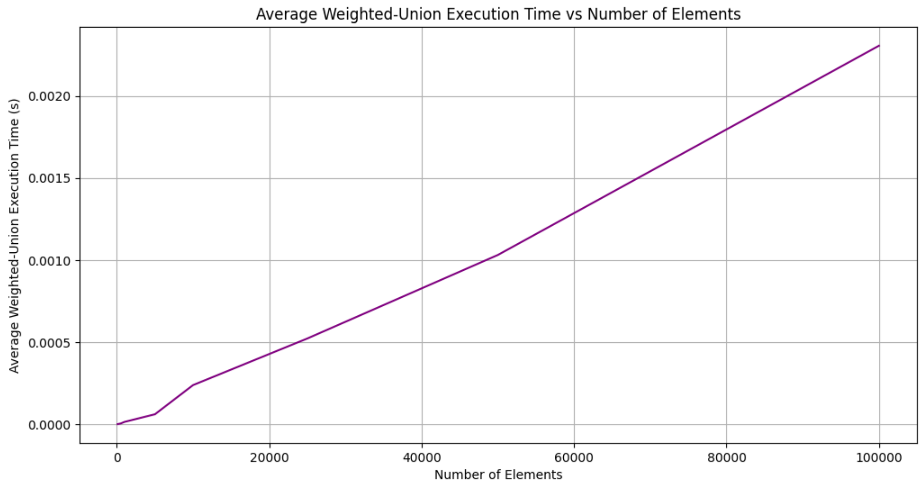


Figure 2: Average Execution Times for Weighted-Union Operation over increasing n, where n is the number of elements.

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

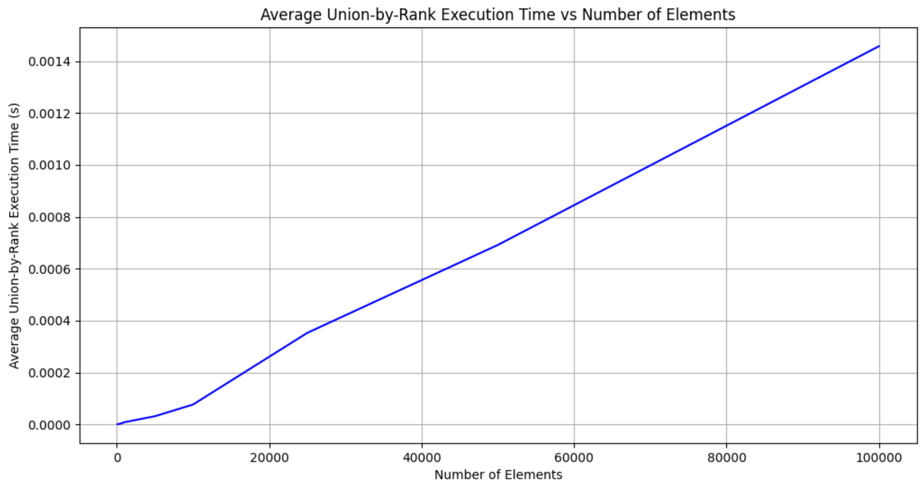


Figure 3: Average Execution Times for Union-by-Rank and Path Compression Heuristics Operations over increasing n, where n is the number of elements.

5.4 Combined Graphs

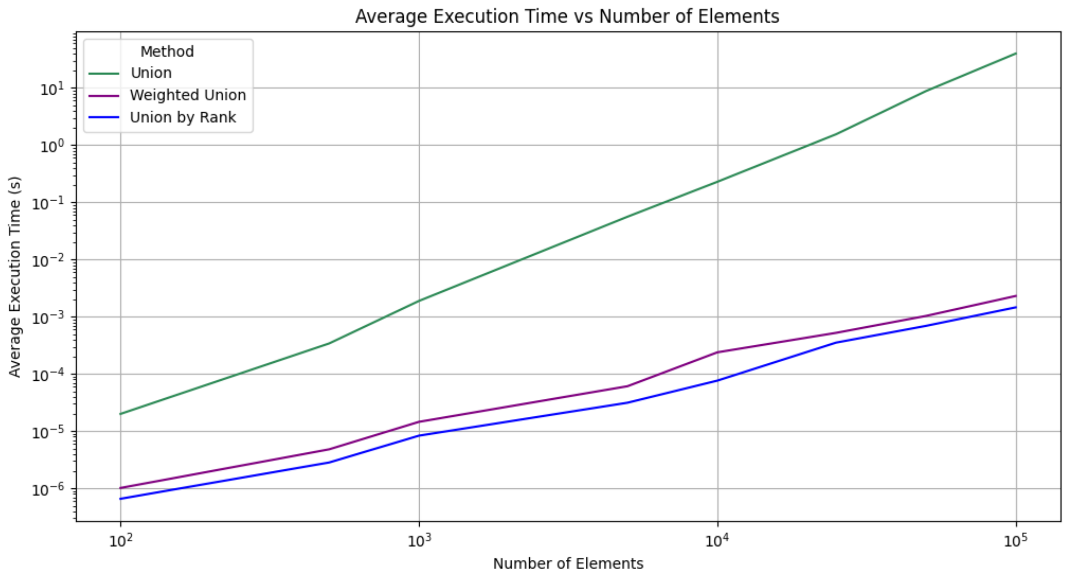
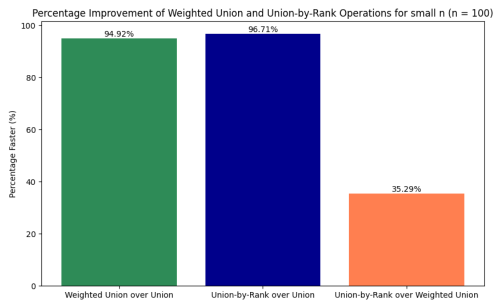
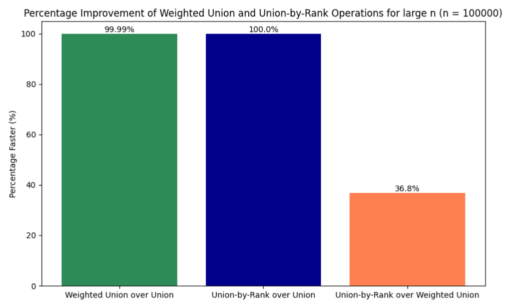


Figure 4: Average Execution Times for Increasing n)



(a) Percentage Improvement of Weighted-Union and Union-by-Rank Operations for small number of elements n



(b) Percentage Improvement of Weighted-Union and Union-by-Rank Operations for large number of elements n

Figure 5: Percentage Improvement of Union operations for different element sizes

6 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:

6.1 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.

6.2 Performance Comparison and Efficiency Gains

Figure 5 illustrates the percentage improvement in execution time of both Weighted-Union and Union-by-Rank operations relative to Basic Union for small and large number of elements.

The empirical results demonstrate the effectiveness of the Weighted-Union and Union-by-Rank operations, compared to the basic Union approach. The following observations can be made:

1. **Weighted-Union Improvement**:
   - For small datasets (n = 100): The Weighted-Union heuristic achieves a 94.92% improvement over the basic Union operation. This substantial gain illustrates the impact of using the weighted heuristic, which limits unnecessary merges and keeps the height of the tree under control, leading to faster operations overall.
   - For large datasets (n = 100,000): The improvement is even more pronounced at 99.99%. This emphasizes how well the weighted-union approach scales with increasing set sizes, supporting Theorem 21.1, which explains the efficiency of maintaining balanced structures.
2. **Union-by-Rank with Path Compression Improvement**:
   - For small datasets (n = 100): Union-by-Rank with path compression shows a 96.71% improvement over basic Union. This highlights how combining path compression with rank balancing ensures quicker Find-Set operations, as elements point directly to their set's representative after compression.
   - For large datasets (n = 100,000): The improvement reaches 100% over basic Union, underscoring the significant efficiency gains from combining path compression with rank-based merging. Path compression helps further flatten the tree, resulting in near-constant time operations, especially in larger datasets.
3. **Union-by-Rank vs Weighted Union**:
   - For small datasets (n = 100): Union-by-Rank is 35.29% faster than Weighted-Union. Although both strategies aim to maintain balanced trees, path compression in Union-by-Rank accelerates repeated Find-Set operations.
   - For large datasets (n = 100,000): Union-by-Rank provides a 36.8% improvement over Weighted-Union. This demonstrates that path compression offers additional benefits, particularly in larger datasets where frequent Find-Set operations occur.
4. **Implications for Large Data**:
   - The increasing efficiency gap as the set size grows further underscores the value of optimizations like path compression and rank-based merging. These techniques reduce time complexity significantly, making them indispensable for handling large datasets. The empirical results align with the theoretical complexity bounds, demonstrating that advanced heuristics can dramatically outperform simpler approaches as the problem size increases.

7 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.

7.1 Node Structure

Each node will have the following attributes:

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

7.2 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.

7.3 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.

7.4 New Operation: Print-Set(x)

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

Algorithm 1 Print-Set(x)

```
1: $y \leftarrow$ Find-Set($x$)        ▷ Find the root of the set containing $x$
2: print $y$
3: $z \leftarrow y$.next
4: while $z \neq y$ do
5:     print $z$
6:     $z \leftarrow z$.next
7: end while
```

7.5 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.

7.6 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*.