COMS4032A - Application of Algorithms Assignment 3

School of Computer Science & Applied Mathematics
University of the Witwatersrand

Lisa Godwin - 2437980

October 22, 2024

Table of Contents

1 Linked-List Representation with Weighted-Union Heuristic

1.1 Theorem 21.1

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 $O(m + n \log n)$ time.

1.2 Algorithm Explanation

The simplest way to implement a disjoint-set data structure is by representing each set as a linked list. Each set has a head pointing to the first object in the list and a tail pointing to the last object. Each object contains:

- A set member.
- A pointer to the next object in the list.
- A pointer back to the set object.

The representative of the set is the first member of the list. Using this representation, both `MAKE-SET` and `FIND-SET` operations run in constant time, $O(1)$. The `UNION` operation, however, takes longer as it involves appending one list to the other. The representative of the resulting set is the first member of the list that is appended to. Updating the pointers for all elements in the appended list takes time proportional to the length of that list.

Without any optimizations, a sequence of m operations on n elements may require $\Theta(n^2)$ time in the worst case. The *weighted-union* heuristic mitigates this issue by always appending the smaller list to the larger one. Theorem 21.1 shows that this approach yields a total running time of $O(m + n \log n)$ for a sequence of m `MAKE-SET`, `UNION`, and `FIND-SET` operations.

2 Tree-Based Representation with Union-by-Rank and Path Compression

2.1 Algorithm Explanation

In this approach, sets are represented by trees. Each node points to its parent, and the root of the tree is the representative of the set. The two heuristics used to improve the performance of this representation are:

- **Union by rank**: When performing a `UNION`, the tree with the smaller rank is made a subtree of the tree with the larger rank. If both trees have the same rank, one root becomes the child of the other, and the rank of the new root increases by 1.
- **Path compression**: During a `FIND-SET`, every node on the path to the root is updated to point directly to the root. This reduces the depth of the trees over time.

These two heuristics together result in a nearly constant running time per operation. Specifically, the worst-case time for a sequence of m operations on n elements is $O(m\alpha(n))$.

By combining the union-by-rank and path compression heuristics, we achieve an amortized time complexity of $O(m\alpha(n))$ for m operations. In practice, $\alpha(n) \leq 4$ for any reasonable input size, making the running time nearly linear.

3 Experiment

Set Sizes: The set sizes were chosen to provide a variety of data points, covering both small and large problem instances. By using this range of sizes, we can observe how the algorithm behaves for a small number of sets and how it scales as the number of sets increases by orders of magnitude.

100, 1000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000

Operations: For each set size, you create n individual sets (using *MAKE-SET*). Then, you use the *UNION* operation to merge all sets into one.

Timing: You record the time before starting the set creation and union operations. After completing all the operations, you stop the timer.

Repetition: Each experiment is run 5 times and the times are averaged to get a more reliable measurement.

4 Results

For the Linked List Weighted-Union heuristic, the time complexity is approximately $O(m + n \log n)$. This is because, while the weighted-union heuristic ensures efficient unions by merging the smaller set into the larger one, the *FIND-SET* operation can become costly in the absence of path compression, particularly as the number of elements grows. In this experiment, as the number of sets increases beyond 10000, the graph shows a steep upward trend for the weighted-union heuristic, indicating that the average time per operation increases significantly with larger input sizes.

In contrast, the Union-by-Rank with Path Compression heuristic achieves a much more efficient time complexity of $O(m\alpha(n))$. As a result, even for very large numbers of sets, the average time per operation increases much more slowly compared to the weighted-union heuristic. The graph shows that the union-by-rank with path compression remains efficient even as the number of sets reaches 100000, growing at a far slower rate.

The results clearly highlight the advantage of the union-by-rank with path compression heuristic, particularly for large datasets. While the weighted-union heuristic performs reasonably well for smaller set sizes, its time complexity causes it to scale poorly with

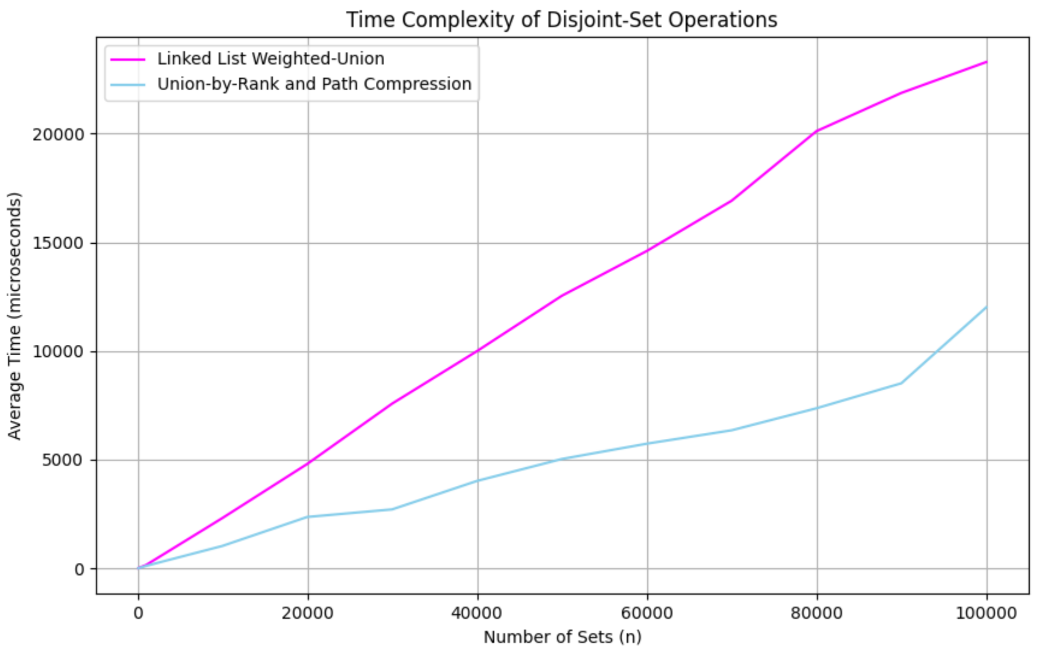

Figure 1: Line Graph comparing time complexity for weighted union heuristic vs union-by-rank and path compression

larger inputs, leading to a noticeable performance gap between the two heuristics as the number of sets increases.

5 Conclusion

We implemented and empirically analysed two different data structures for disjoint sets. Our experiments provided evidence supporting the theoretical time complexities of both approaches: $O(m + n \log n)$ for the linked-list representation with weighted-union and $O(m\alpha(n))$ for the tree-based representation with union-by-rank and path compression.

6 Solution to Exercise 21.3-4

Question:
Suppose that we wish to add the operation *PRINT-SET(x)*, which is given a node x and prints all the members of x's set, in any order. Show how we can add just a single attribute to each node in a disjoint-set forest so that *PRINT-SET(x)* takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that we can print each member if the set in $O(1)$ time.

Answer:
To implement the operation `PRINT-SET(x)`, which prints all members of the set contain-

ing node x, we modify the disjoint-set data structure by adding a single attribute to each node. Specifically, we introduce a pointer to the next member in the set. The modified `Node` structure now includes a `next` pointer, which links to the next node in the same set.

When a new set is created, the `next` pointer of the node is initialized to point to itself, indicating that it is the only member of its set. In the updated `link` function, we introduced logic to maintain the circular linked list structure of the sets when performing unions. Specifically, after linking the parent pointers to combine two sets, we now also update the `next` pointers. This ensures that the last member of one set points to the first member of the other set, and vice versa, maintaining the circular linkage. This change allows the `PRINT-SET` operation to efficiently traverse and print all members of the set containing a given node in linear time.

The operation `PRINT-SET(x)` works by first finding the representative of the set containing x using `FIND-SET(x)`. Once the representative is found, we can traverse through the circular linked list of nodes by following the `next` pointers, printing each node in the process. Since each member is printed in $O(1)$ time, the total running time is linear in the number of members in the set.

Algorithm 1 PRINT-SET(x)

```
1: $y \leftarrow$ FIND-SET(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
```

The time complexity of the `PRINT-SET(x)` operation is $O(m)$, where m is the number of members in the set containing node x. This is because the operation traverses the circular linked list of nodes, printing each member in constant time. The other disjoint-set operations (i.e., `FIND-SET`, `UNION`, and `MAKE-SET`) are unaffected by this modification, as the addition of the `next` pointer does not alter their asymptotic running time. The `FIND-SET` and `UNION` operations still run in $O(\alpha(n))$ and `MAKE-SET` continues to operate in $O(1)$ time. This analysis assumes that each member of the set can be printed in $O(1)$ time, as stipulated by the problem.