Comparison of Disjoint-Sets Implementations

Antonio Bruno

May 2024

Contents

1 Introduction

1.1 Problem statement

In this report I will discuss and compare two different methods for implementing data structures for **disjoint sets**, i.e.:

- The implementation based on **linked lists**
- The implementation based on **rooted trees**

In order to compare the efficiency of these implementations, I will test disjoint sets based algorithms for finding **connected components** in non-oriented graphs with different sizes and densities.

1.2 Hardware and software specifications

The code will be run on a computer with the following specifications:

- **Processor**: 11th Gen Intel Core i7-1165G7 @ 2.80GHz
- **RAM**: 16 GB
- **OS**: Windows 11
- **Python version**: 3.12.2

2 Theoretical analysis of the problem

2.1 Fundamental aspects

We will now cover the fundamental information required to understand disjoint sets and the operations I am going to analyze.

2.1.1 Disjoint sets

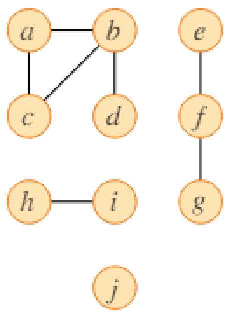
A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, S_1, \dots, S_k\}$ of disjoint dynamic sets, i.e., sets which have no element in common ($S_i \cap S_j = \emptyset, \forall\, i, j$). To identify each set, a **representative** is chosen between the members of the set; it doesn't matter which member is chosen as representative, as long as it doesn't change until the set is modified.

A disjoint-set data structure must support the following three operations:

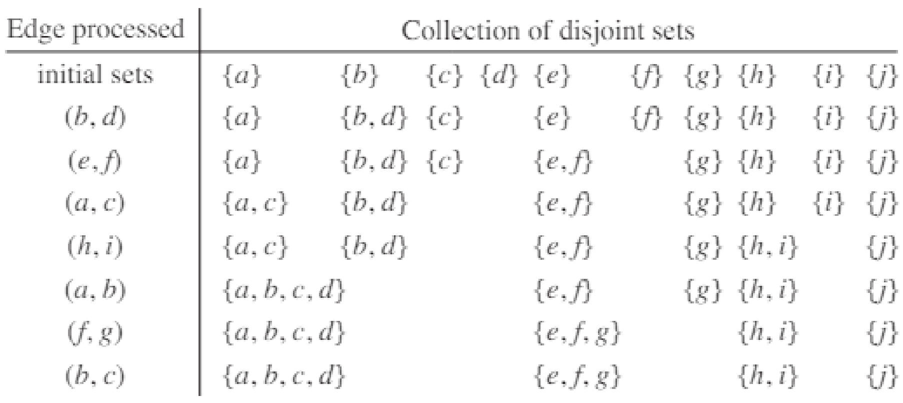
- **make-set(x)**: where object x does not already belong to some other set, creates a new set whose only member (and thus representative) is x.
- **union(x,y)**: unites two disjoint sets containing x and y, say S_x and S_y, into a new set that is the union of these two sets. A new representative is chosen (usually it's one of the representatives of the former sets).
- **find(x)**: returns a pointer of the representative of the unique set containing x.

2.1.2 Connected components of a graph

One of the many applications of disjoint-set data structures is determining the connected components of a non-oriented graph, i.e., subsets of vertices where each vertex is reachable from every other vertex within the same subset by traversing edges; for example the graph in figure 1a has 3 connected components.



(a)



Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,f)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,f\}$		$\{g\}$	$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,f\}$		$\{g\}$	$\{h,i\}$		$\{j\}$
(f,g)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(b)

Figure 1: **(a)** A graph with 4 connected components. **(b)** The collection of disjoint sets after processing each edge.

We can use disjoint-sets data structures to compute connected components in the following way: initially, each vertex **v** is placed in its own set; then, for each edge **(u, v)** the sets containing **u** and **v** are united; after all the edges are processed, two vertices belong to the same connected components if and only if the objects corresponding to the vertices belong to the same set, as shown in figure 1b.

This procedure can be described by the following pseudo-code:

```
Input: Graph $G = (V, E)$
for each vertex $v \in V$ do
    make-set($v$);
end
for each edge $(u, v) \in E$ do
    if find(u) $\neq$ find(v) then
        union($u, v$);
    end
end
```

connected-components(G)

Computing time for this algorithm depends on the specific implementation of disjoint sets.

2.2 Implementation of disjoint-sets data structures

Let's now focus on the two main implementations for disjoint-sets data structures.

2.2.1 Linked-list representation of disjoint sets

A simple implementation of disjoint-sets data structures is based on **linked lists**. In particular, every set is a linked list which has a pointer to its head object, i.e., the first element of the list as well as the **representative** of the set, and a pointer to its tail object, i.e, the last element of the list. Each object in the set contains a set member, a pointer to the next object in the list and a pointer back to the set object (figure 2a).

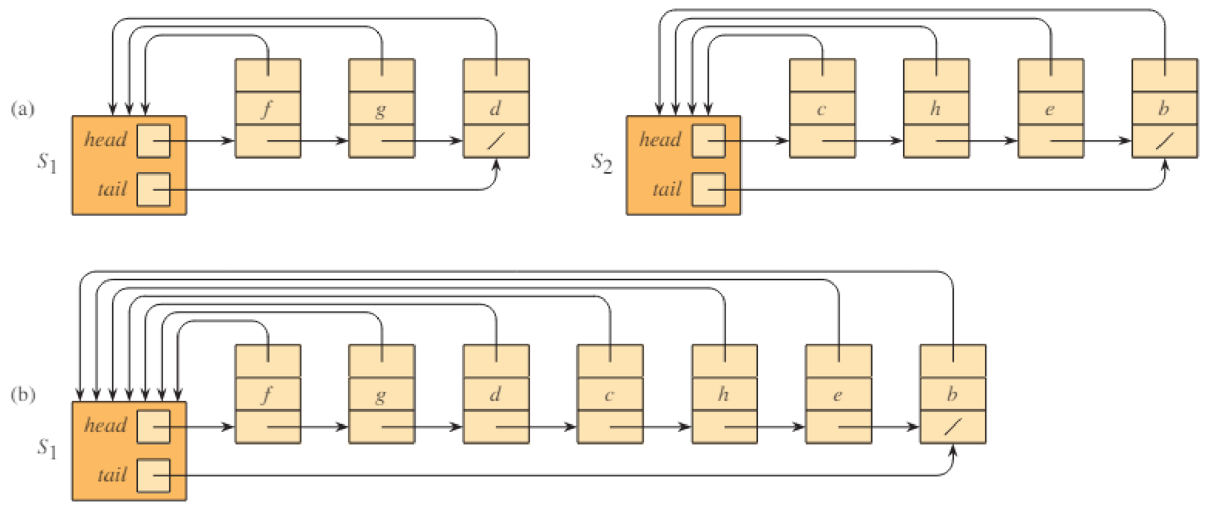


Figure 2: **(a)** Linked-list representation of two sets. **(b)** The result of union(g,e).

With this representation, a simple **union(x,y)** operation appends **y**'s list onto the end of **x**'s list and updates the pointer to the set of every object of **y**'s list, which now has to point to **x**'s set (figure 2b). As you can imagine, the **union** operation takes significantly more time than the **find** and **make-set** operations, because it must iterate through a whole set.

Sometimes the simple **union** operation may append a longer list onto a shorter list, resulting in a waste of time. The **weighted-union heuristic** solves this issue by appending the shorter list to the longer list (this requires keeping a **size** attribute for every set object), resulting in a faster running time for the **union** operation.

2.2.2 Disjoint-sets forests

A faster implementation of disjoint sets represents sets by **rooted trees**, with each node containing one member and the pointer to its parent and each tree representing one set, with the **representative** being its root (figure 3a).

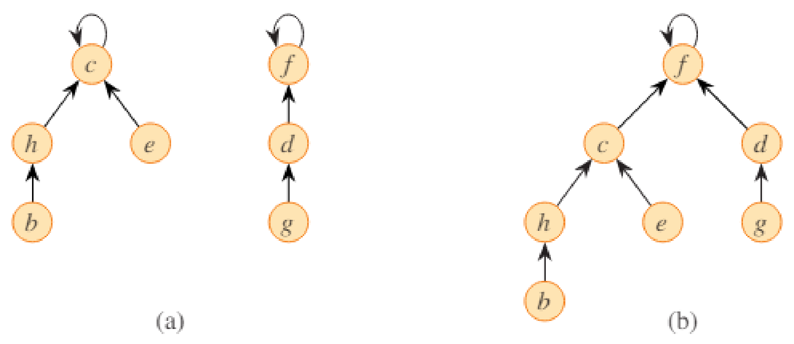


Figure 3: **(a)** Trees representing the sets of figure 2. **(b)** the result of union(g,e).

The **union(x,y)** operation simply sets the root of **x** as the new root of the root of **y** (figure 3b). However the pointer to root of each tree is returned by the **find(x)** operation by following parent pointers from **x**; thus the **union** operation takes linear time.

There are two heuristics which significantly reduce the time complexity of the **union** operation, i.e., **union by rank** and **path compression**; in this report I will only focus on the latter.

Path compression is quite simple yet highly effective: in the **find(x)** operation, once the root of tree has been found, it is set as the parent of all the nodes of the tree, indeed *compressing* the tree, as shown in figure 4; this ensures that succeeding **find** operations finding the root of the tree will take less time.

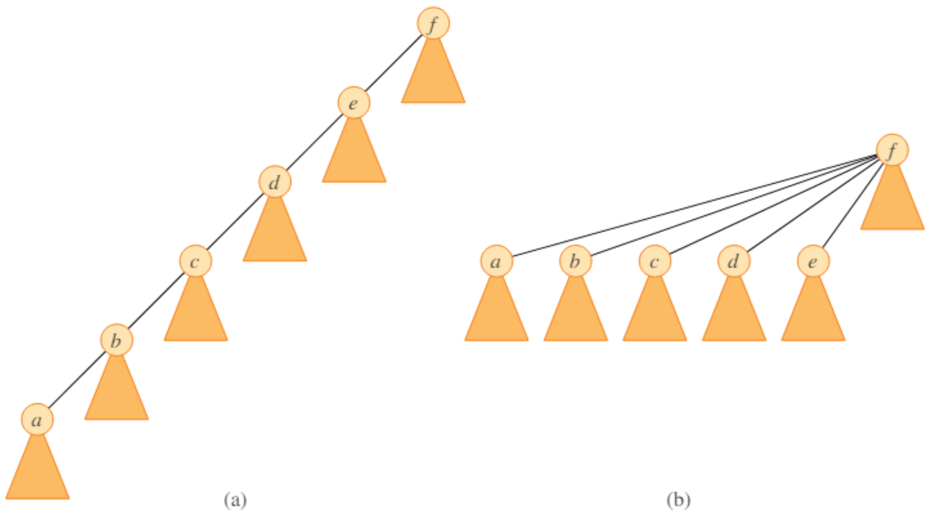


Figure 4: **(a)** A tree representing a set prior to executing find(a) and **(b)** after executing find(a)

2.3 Expected performance

Determining connected components of a graph can be more or less efficient depending on the implementation of disjoint sets and eventual heuristics. We can of course expect the implementation with linked lists and simple union to be in general less efficient than the other two (i.e., linked lists with weighted union and forest trees with path compression). However, the comparison between the latter can be interesting because there's no actual a priori indicator of which one will be more efficient. In particular, the time complexities for determining connected components in each of the implementations are the following:

- **linked lists with simple union:** $\mathcal{O}(m \cdot n)$
- **linked lists with weighted union:** $\mathcal{O}(m + n \cdot \log_2 n)$
- **rooted trees with path compression:** $\mathcal{O}(n + f \cdot (1 + \log_{2+f/n} n)$

where **n** is the number of **make-set** operations, **f** is the number of **find** operations and **m** is the number of total operations. I won't spend time proving these results but we can take them as true. As you can see, the efficiency of the latter implementations really depends on the type of graph we're operating on and it's difficult to make general predictions; in section 4 we will see the actual running time of these algorithms.

3 Code documentation

3.1 Content outline and interaction between modules

I decided to divide the code into five modules, i.e.:

- **linked_list_disjoint_set.py:** this module handles the linked list representation of disjoint sets with its classes and methods.
- **disjoint_set_forest.py:** this module handles the rooted trees representation of disjoint sets with its classes and methods

- **graph.py:** this module handles the graph data structure and provides methods for generating graphs.
- **test.py:** this module provides the methods for testing the efficiency of different implementations of the **connected-components** algorithm.
- **main.py:** the main executable program, where I run the tests.

The classes are organized as follows:

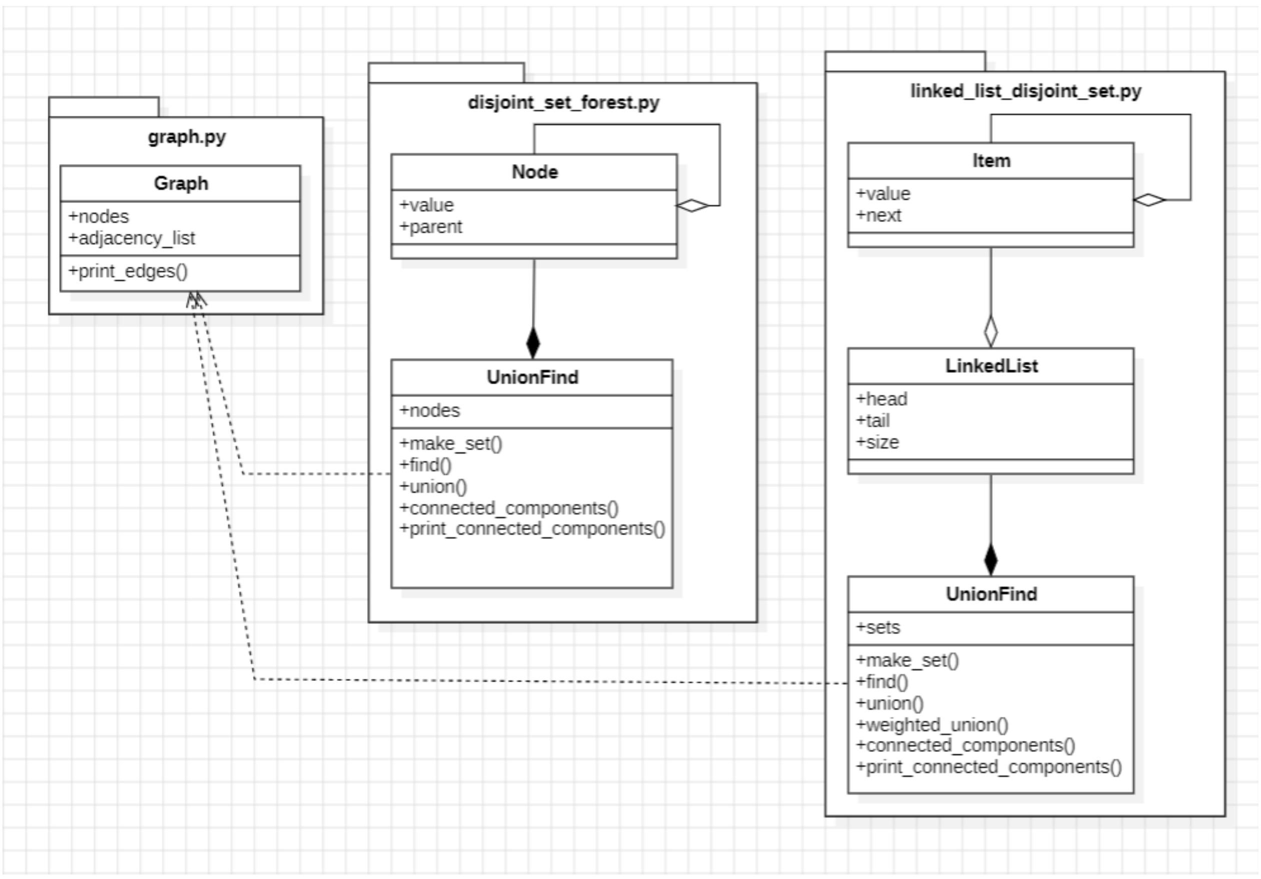


Figure 5: Class Diagram

3.2 Implementation choices

I will now briefly explain the reasons behind some of the implementation choices I made.

- **Nodes and edges:** Nodes are simply represented by integers, whereas edges are represented by pairs of integers. This implementation is simple and makes sure that the graph object is not modified after running the connected-components algorithms, because new data structures are created for storing nodes.
- **Graph generation:** The graphs are generated as follows: given a number of nodes and a number of edges, nodes are generated with sequential values whereas edges are randomly generated; This generation is fast and unbiased.
- **UnionFind classes:** each UnionFind class has a data structure for storing the sets, i.e., a dictionary for linked lists and a list of nodes for rooted trees; this is because the former requires a direct access to the sets, whereas the latter requires following parent pointers.
- **Shuffling edges:** in the connected-components algorithms, the edges of the graph are shuffled before processing them; this is because the order of the edges can affect the running time of

the algorithms. Since each test runs the algorithm multiple times, shuffling the edges ensures a more accurate average running time.

3.3 Description of implemented methods

I will now briefly explain the functionality of each implemented method for each class/module.

- class **Graph**:
  - **__init__()**: the class constructor; receives as input the number of nodes and the density and generates a random non oriented graph.
  - **print_edges()**: simply prints every edge of the graph; this method was only used for debugging purposes.
- class **UnionFind** from module **linked_list_disjoint_set**:
  - **make_set()**: receives an integer value as input and creates a new set (a **LinkedList** object) containing the **Item** with the given value.
  - **find()**: receives a value as input and returns the head (an **Item** object) of the set containing the **Item** with the given value.
  - **union()**: receives two values as input and, if the items with the given values belong to different sets, performs the union between them, as well as updates the size of the new set.
  - **weighted_union()**: same functionality as **union()**, but before performing the union, checks which set is smaller and eventually appends it to the larger set.
  - **connected_components()**: receives a non-oriented graph as input and determines its connected components, which are stored in the **sets** dictionary; for details see paragraph 2.1.2.
  - **print_connected_components()**: prints the list of connected components of the given graph; used only for debugging purposes.
- class **UnionFind** from module **disjoint_set_forest**:
  - **make_set()**: receives an integer value as input and creates a new tree, i.e., a new **Node** with the given value and whose root is itself.
  - **find()**: receives a value as input and returns the root (a **Node** object) of the tree containing the **Node** with the given value, as well as executes **path compression** (see paragraph 2.2.2)
  - **union()**: receives two values as input and, if the nodes with the given values belong to different trees, performs the union between them.
  - **connected_components()**: receives a non-oriented graph as input and determines its connected components, which are stored implicitly the **nodes** list; for details see paragraph 2.1.2.
  - **print_connected_components()**: prints the list of connected components of the given graph; used only for debugging purposes.
- module **test**:
  - **test_linked_list_time()**: receives three parameters as input: a Graph *graph*, a boolean *weighted* and an integer *measurements.* Returns the average running time of **connected_components()** on *graph* using the linked-list data structure, executed *measurements* times with the weighted-union heuristic if *weighted* is True. This method is indeed used for testing both simple and weighted union on linked lists.

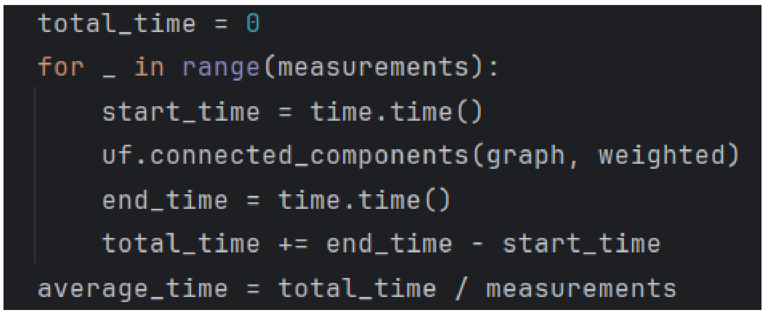
- **test_forest_time()**: receives *graph* and *measurements*. Executes **connected_components()** on *graph* using the rooted-trees data structure (with path compression) *measurements* times, and returns its average running time.
- **compare_all_running_times()**: receives three integer values as input: *edges_nodes_ratio*, *step* and *measurements*. Generates 10 graphs of different dimensions based on *step* and *edges_nodes_ratio*, then calls the aforementioned methods (passing the *measurements* parameter) and finally plots the running times of the three different algorithms on a time-dimension graph.
- **compare_wlist_vs_forest()**: identical to the latter, except it only compares linked lists with weighted-union and rooted trees.

4 Tests and results

In this section I'm going to present the details of the tests conducted and discuss and analyse the results.

4.1 Evaluation metrics

The primary metric used to evaluate the performance of the algorithms will be the running time: we measure the time taken by each implementation to compute the connected components of the generated graphs (figure 6) and then compare them for each test case.

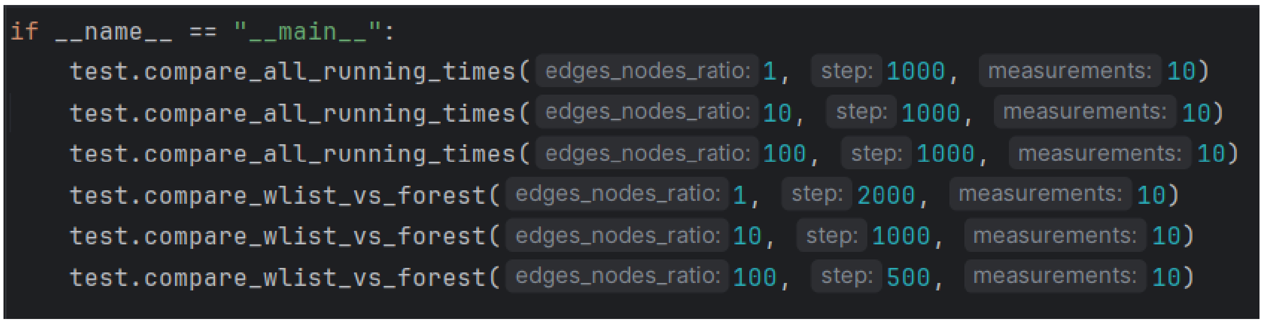


```python
total_time = 0
for _ in range(measurements):
    start_time = time.time()
    uf.connected_components(graph, weighted)
    end_time = time.time()
    total_time += end_time - start_time
average_time = total_time / measurements
```

Figure 6: Time measurements using *time* module.

4.2 Test cases and results

I've conducted a total of six tests, three of which comparing all implementations and the other three comparing only linked lists with weighted union to rooted trees, as shown in figure 7. In each test the connected components algorithm is run on 10 different permutations of edges of the same graphs, and the step is adjusted according to the test case. The main difference between the tests is the *density* of the graphs, i.e., the relative number of edges, which is modified through the *edges_nodes_ratio* parameter. In particular, I generated graphs where the number of edges is respectively 1, 10 and 100 times the number of nodes.



```
if __name__ == "__main__":
    test.compare_all_running_times( edges_nodes_ratio: 1,  step: 1000,  measurements: 10)
    test.compare_all_running_times( edges_nodes_ratio: 10,  step: 1000,  measurements: 10)
    test.compare_all_running_times( edges_nodes_ratio: 100,  step: 1000,  measurements: 10)
    test.compare_wlist_vs_forest( edges_nodes_ratio: 1,  step: 2000,  measurements: 10)
    test.compare_wlist_vs_forest( edges_nodes_ratio: 10,  step: 1000,  measurements: 10)
    test.compare_wlist_vs_forest( edges_nodes_ratio: 100,  step: 500,  measurements: 10)
```

Figure 7: Tests conducted.

Let's now see the results for each test case:

test case	implementations compared	|E|	step	measurements
1	all	|V|	1000	10

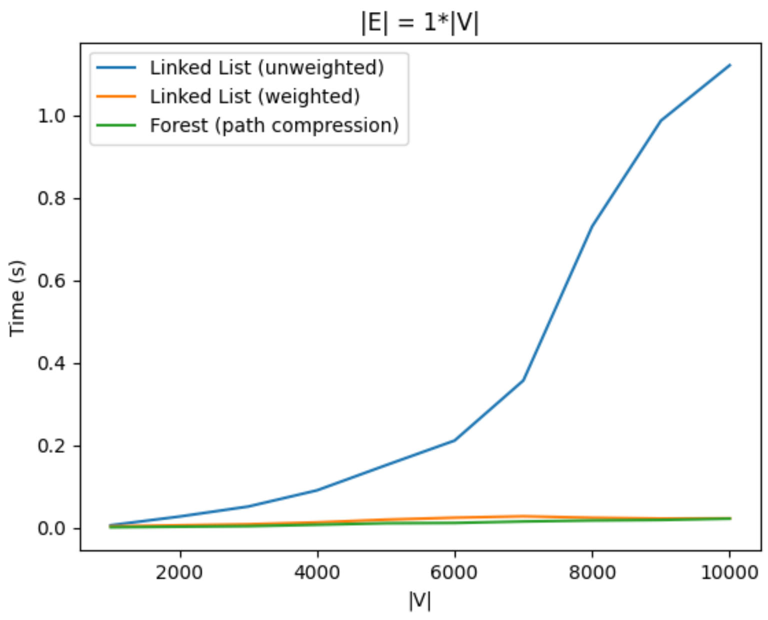


Figure 8: Results of test case 1.

test case	**implementations compared**	$\mathbf{\vert E \vert}$	**step**	**measurements**
2	all	$10 \cdot \vert V \vert$	1000	10

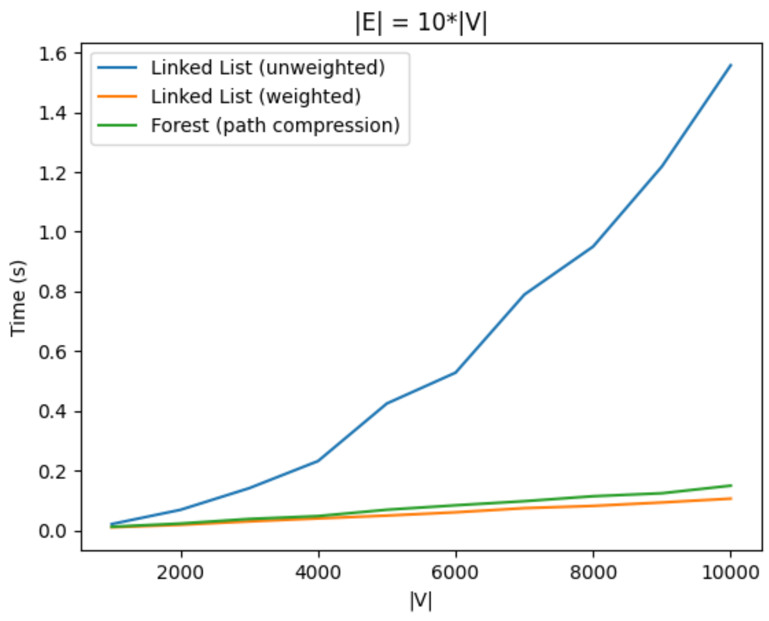


Figure 9: Results of test case 2.

test case	**implementations compared**	$\mathbf{\vert E \vert}$	**step**	**measurements**
3	all	$100 \cdot \vert V \vert$	1000	10

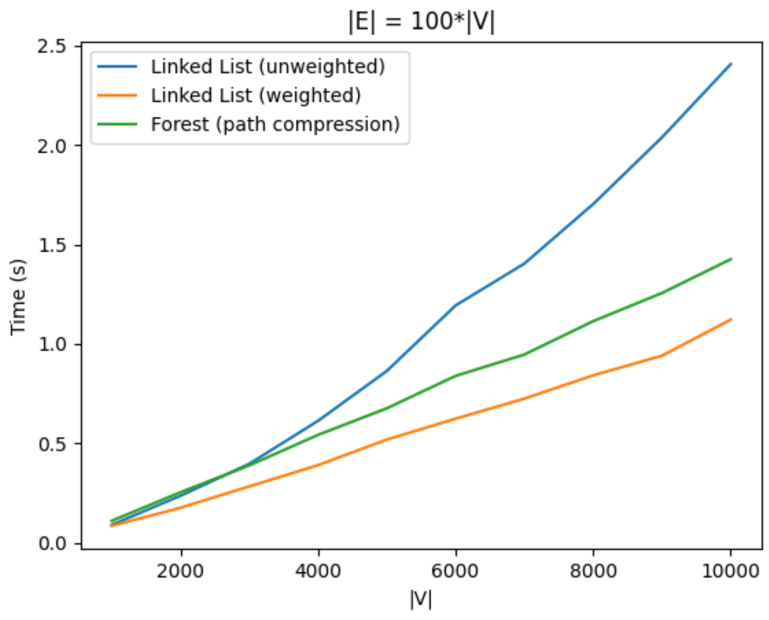


Figure 10: Results of test case 3.

test case	**implementations compared**	**|E|**	**step**	**measurements**
4	weighted-union and forest	$\|V\|$	2000	10

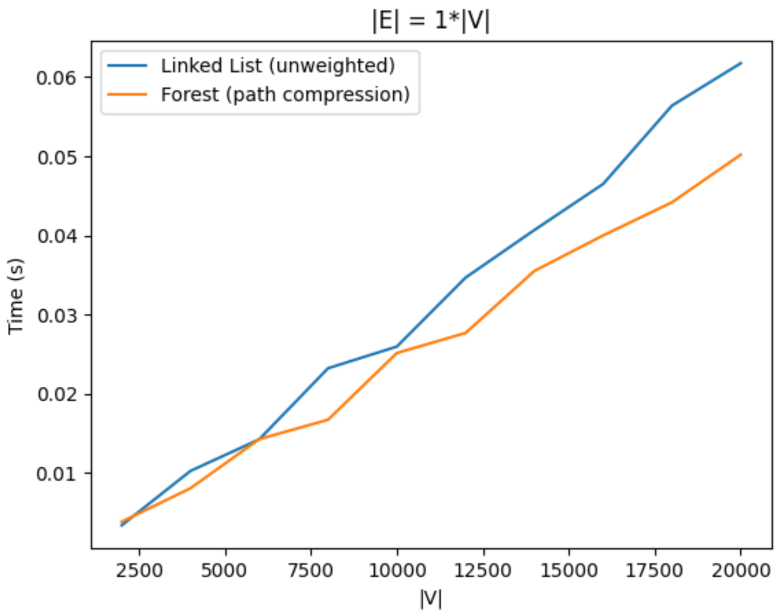


Figure 11: Results of test case 4.

test case	**implementations compared**	**|E|**	**step**	**measurements**
5	weighted-union and forest	$10 \cdot \|V\|$	1000	10

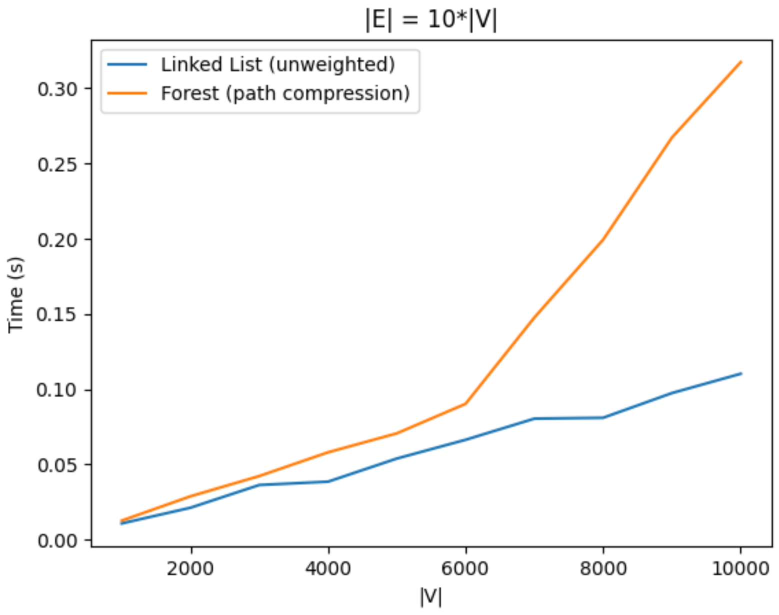


Figure 12: Results of test case 5.

test case	implementations compared	|E|	step	measurements
6	weighted-union and forest	$100 \cdot \|V\|$	500	10

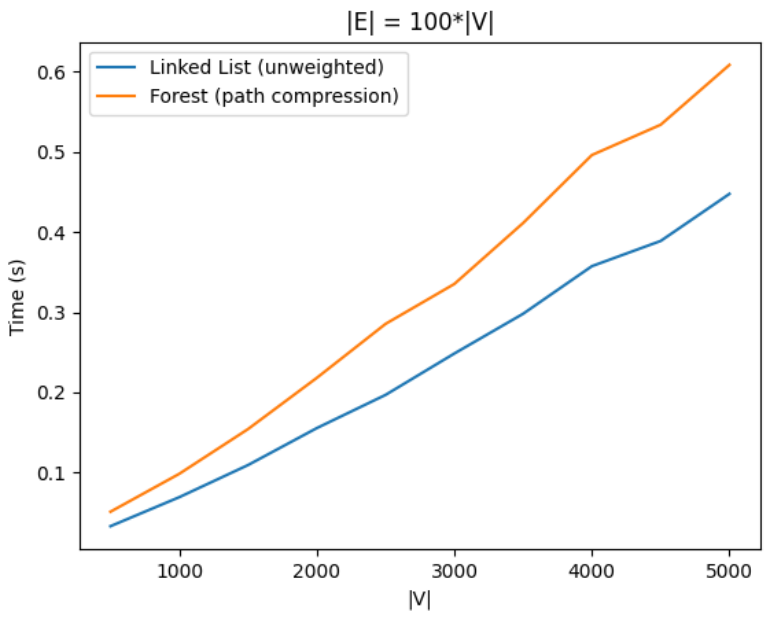


Figure 13: Results of test case 6.

4.3 Analysis of the results

In the first three test cases, we can say to have proved the efficiency of heuristics on running time, since the implementation with linked lists with simple union is significantly slower than the other two (figures 8 and 9). Even though on more dense graphs the time difference is smaller (figure 10), we can still observe a significant difference and it seems to grow as the dimension of the graph grows. In figures 9 and 10 the weighted-union heuristic on linked lists seems to be slightly more efficient than the path-compression heuristic on rooted trees. This comparison is highlighted in the subsequent test cases: we can see that on sparse graphs (i.e., graphs with small density) the path-compression rooted trees implementation is slightly faster (figure 11), however, when density grows, the linked lists implementation is more efficient and its running time seems to grow slower than the other when the graphs dimension grows (figures 12 and 13).

5 Conclusion

In conclusion, this study has shed light on the efficiency of disjoint-set implementations in solving the problem of finding connected components in non-oriented graphs. Through testing and analysis, we have gained valuable insights into the impact of different heuristics and graph characteristics on algorithm performance.

Our results clearly demonstrate the significance of heuristics, such as weighted union in linked lists and path compression in rooted trees, in optimizing the running time of disjoint-set operations. We observed notable variations in performance across different graph densities and sizes, highlighting the importance of considering these factors when selecting the most suitable implementation for a given application.

While path compression in rooted trees shows promise for sparse graphs, linked lists with weighted union emerges as a better choice for denser graphs, particularly as graph size increased. Ultimately, the most efficient implementation of disjoint sets would be the one based on rooted trees combining path-compression and union-by-rank heuristics, which ensures in most practical scenarios a linear time complexity for the connected components algorithm, even though it was not analysed in this report.