CPTS 223 Advanced Data Structure C/C++ Fall 2024

MA5: Disjoint Sets (Union-Find)

1 Learning Objectives

At the conclusion of this programming assignment, participants should be able to:

- Design and implement disjoint data structure;
- Test the operations (union and find) of disjoint data structure

2 Prerequisite

Before starting this programming assignment, participants should be able to:

- Apply and implement class templates;
- Design, implement and test programs in C++;
- Cross-platform compilation with CMake;
- Maintain coding projects with Git on github.
- Finish reading §8.1 - 8.5 and 8.7 of Textbook.

3 Overview and Requirements

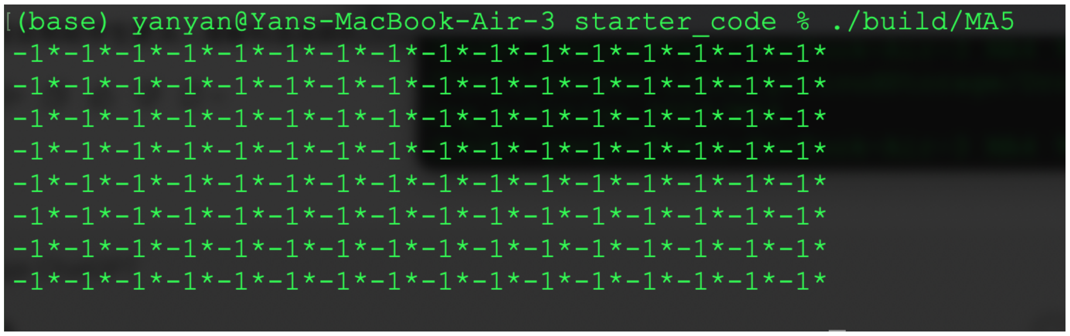
3.1 Overview

In this assignment, you will implement the disjoint set data structure. Disjoint sets, also known as union-find, are a data structure that efficiently manages and tracks a partition of elements into non-overlapping subsets. Each element belongs to one specific subset, and the primary operations supported are "find" and "union". The "find" operation determines which subset a particular element is in, allowing for identification of connected components, while the "union" operation merges two subsets into a single one. Disjoint sets are particularly useful in scenarios where elements need to be grouped or connected based on certain relationships, such as maze generation (see §8.7 in Textbook).

The efficiency of disjoint sets lies in optimizations like path compression and union by rank or size, which make both find and union operations nearly constant time in practical applications, despite the structure's inherent hierarchical nature. Path compression flattens the structure by making elements point directly to their root, reducing the depth of the tree, while union by rank ensures the smaller tree is always added to the larger, keeping the structure balanced. Together, these optimizations lead to a time complexity close to amortized $O(A(n))$, where $A(\cdot)$ is the inverse Ackermann function, making disjoint sets ideal for dynamic connectivity problems in graph theory and network analysis.

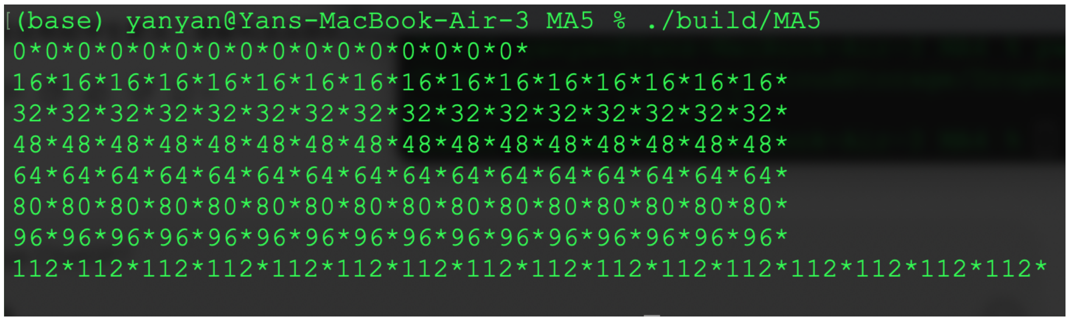
3.2 Requirements

You will implement the disjoint set class with the starter code. The underlying implementation is an array. This array stores the set names for simplicity, following the principle described by Textbook. Three member functions are missing and need to implement:



```
(base) yanyan@Yans-MacBook-Air-3 starter_code % ./build/MA5
-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*
-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*
-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*
-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*
-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*
-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*
-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*
-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*-1*
```

Figure 1: Running result from the starter code.



```
(base) yanyan@Yans-MacBook-Air-3 MA5 % ./build/MA5
0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*
16*16*16*16*16*16*16*16*16*16*16*16*16*16*16*16*
32*32*32*32*32*32*32*32*32*32*32*32*32*32*32*32*
48*48*48*48*48*48*48*48*48*48*48*48*48*48*48*48*
64*64*64*64*64*64*64*64*64*64*64*64*64*64*64*64*
80*80*80*80*80*80*80*80*80*80*80*80*80*80*80*80*
96*96*96*96*96*96*96*96*96*96*96*96*96*96*96*96*
112*112*112*112*112*112*112*112*112*112*112*112*112*112*112*112*
```

Figure 2: Running result from the expected code.

- `find() const` (25 pts, refer to Figure 8.9) is a const-qualified `find` function, meaning it does not modify any member variables of the class. It is used to find the root of a given element x without altering the internal structure. It recursively finds the root (or representative) of the set containing x. In a simple implementation, it may perform the operation without any optimizations like path compression. It is useful when you need to check the representative of a set for an element without modifying the data structure. It provides a safe, read-only way to access the root of an element.

- `find()` (35 pts, refer to Figure 8.16) is the standard `find` function, which performs the same function as `find() const`, but allows modifications to the data structure (i.e., the mutator version). It is typically implemented with *path compression*, an optimization that makes future searches faster. When called on an element `x`, it not only finds the root but also updates the structure by making each node on the path from `x` to its root directly point to the root. This "flattens" the structure, reducing the depth of the tree and improving efficiency in subsequent operations. This mutator `find` function is used in most union-find applications because of its efficiency gains through path compression. It modifies the data structure to make future `find` and `unionSets` operations faster.

- `unionSets()` (40 pts, refer to Figure 8.14 for the union-by-rank version) is responsible for merging two disjoint sets into a single set. It connects the sets containing "root1" and "root2" by making one root point to the other. It is implemented with union by rank (or union by size) to keep the tree balanced and avoid making it too deep. If union by rank is used, the set with the lower rank (height) is attached under the root of the set with the higher rank. If both sets have the same rank, one root becomes the parent, and its rank is increased by one. On the other hand, if union by size is used, the smaller set is attached under the root of the larger set, maintaining a balanced structure based on the number of elements rather than tree height.

See Figure 1 and Figure 2 for the running result from the starter code and expected code, respectively.

4 How to Submit: Github (and Share with TA)

1. In your Git repository for this class's coding assignments, **create a new branch called "MA5"** (Refer to this YouTube video about how to create a branch from Github web interface or the terminal). In the current working directory, also create a new directory called "MA5" and place all MA5 files in the "MA5" directory. All files for MA5 should be added, committed and pushed to the remote origin which is your private GitHub repository created when during PA1 (NO NEED TO CREATE A NEW REPO).
2. You should submit at least the following files:
   - the header files: "DisjointSets.h".
   - the main C++ source file: "main.cpp" and "DisjointSets.cpp".
   - a "CMakeLists.txt" file containing your CMake building commands on Linux/WSL/MacOS which can compile your code.
3. You can refer the example GitHub template project for how to use CMake at `https://github.com/DataOceanLab/CPTS-223-Examples`.
4. Please invite the GitHub accounts of TAs (see Syllabus page and check TA's names as well as their Github usernames before submitting) as the collaborators of your repository. You should submit a URL link to the branch of your private GitHub repository on Canvas. Otherwise, we will not be able to see your repository and grade your submission.
5. Please push all commits of this branch before the due date for submitting your Github link to Canvas portal. Otherwise it might be considered as late submission.

Here is a checklist for submitting your code:

- Invite all TAs and instructor as collaborators of your created repository "CPTS223_assignments". Their github username can be found in Syllabus on Canvas.
- Commit and push your local code to your repository.
- Make sure that your repository reflects and contains all your latest files.
- Copy the link to your repository in the Canvas submission portal.

5 Grading Guidelines

This assignment is worth 100 points. We will grade according to the following criteria: See Section 3.2 for individual points.