2.4.2 Union-Find Disjoint Sets

The Union-Find Disjoint Set (UFDS) is a data structure to model a collection of *disjoint sets* with the ability to efficiently[24]—in $\approx O(1)$—determine which set an item belongs to (or to test whether two items belong to the same set) and to unite two disjoint sets into one larger set. Such data structure can be used to solve the problem of finding connected components in an undirected graph (Section 4.2.3). Initialize each vertex to a separate disjoint set, then enumerate the graph's edges and join every two vertices/disjoint sets connected by an edge. We can then test if two vertices belong to the same component/set easily.

These seemingly simple operations are not *efficiently* supported by the C++ STL `set` (and Java `TreeSet`), which is not designed for this purpose. Having a `vector` of `set`s and looping through each one to find which set an item belongs to is expensive! C++ STL `set_union` (in `algorithm`) will not be efficient enough although it combines two sets in *linear time* as we still have to deal with shuffling the contents of the `vector` of `set`s! To support these set operations efficiently, we need a better data structure—the UFDS.

The main innovation of this data structure is in choosing a representative 'parent' item to represent a set. If we can ensure that each set is represented by only one unique item, then determining if items belong to the same set becomes far simpler: The representative 'parent' item can be used as a sort of identifier for the set. To achieve this, the Union-Find Disjoint Set creates a tree structure where the disjoint sets form a forest of trees. Each tree corresponds to a disjoint set. The root of the tree is determined to be the representative item for a set. Thus, the representative set identifier for an item can be obtained simply by following the chain of parents to the root of the tree, and since a tree can only have one root, this representative item can be used as a unique identifier for the set.

To do this efficiently, we store the index of the parent item and (the upper bound of) the height of the tree of each set (`vi p` and `vi rank` in our implementation). Remember that `vi` is our shortcut for a vector of integers. `p[i]` stores the immediate parent of item `i`. If item `i` is the representative item of a certain disjoint set, then `p[i] = i`, i.e. a self-loop. `rank[i]` yields (the upper bound of) the height of the tree rooted at item `i`.

In this section, we will use 5 disjoint sets {0, 1, 2, 3, 4} to illustrate the usage of this data structure. We initialize the data structure such that each item is a disjoint set by itself with rank 0 and the parent of each item is initially set to itself.

To unite two disjoint sets, we set the representative item (root) of one disjoint set to be the new parent of the representative item of the other disjoint set. This effectively merges the two trees in the Union-Find Disjoint Set representation. As such, `unionSet(i, j)` will cause both items '`i`' and '`j`' to have the same representative item—directly or indirectly. For efficiency, we can use the information contained in `vi rank` to set the representative item of the disjoint set with *higher rank* to be the new parent of the disjoint set with *lower rank*, thereby `minimizing` the rank of the resulting tree. If both ranks are the same, we arbitrarily choose one of them as the new parent and increase the resultant root's rank. This is the 'union by rank' heuristic. In Figure 2.6, top, `unionSet(0, 1)` sets `p[0]` to 1 and `rank[1]` to 1. In Figure 2.6, middle, `unionSet(2, 3)` sets `p[2]` to 3 and `rank[3]` to 1.

For now, let's assume that function `findSet(i)` simply calls `findSet(p[i])` recursively to find the representative item of a set, returning `findSet(p[i])` whenever `p[i] != i` and `i` otherwise. In Figure 2.6, bottom, when we call `unionSet(4, 3)`, we have `rank[findSet(4)] = rank[4] = 0` which is smaller than `rank[findSet(3)] = rank[3] = 1`, so we set `p[4] = 3` *without* changing the height of the resulting tree—this is the 'union by rank' heuristic

[24] M operations of this UFDS data structure with 'path compression' and 'union by rank' heuristics run in $O(M \times \alpha(n))$. However, since the inverse Ackermann function $\alpha(n)$ grows very slowly, i.e. its value is just less than 5 for practical input size $n \leq 1M$ in programming contest setting, we can treat $\alpha(n)$ as constant.

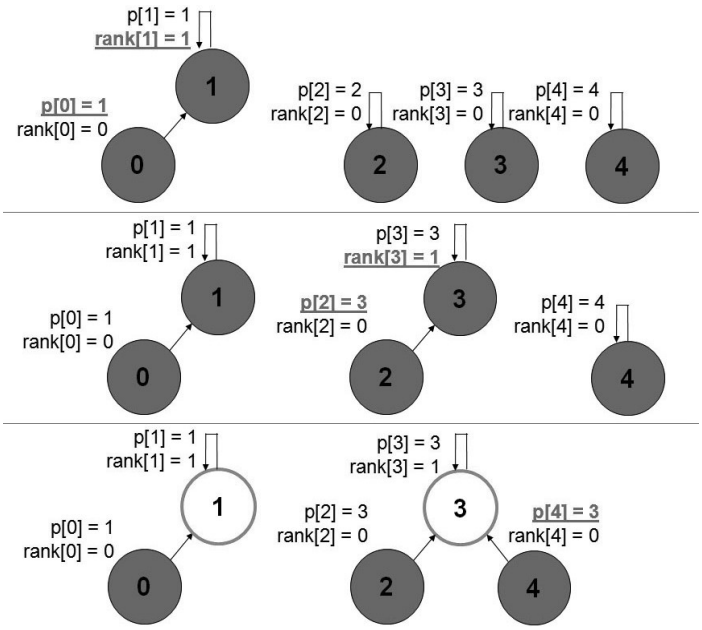

Figure 2.6: `unionSet(0, 1)` → `(2, 3)` → `(4, 3)` and `isSameSet(0, 4)`

at work. With the heuristic, the path taken from any node to the representative item by following the chain of 'parent' links is effectively minimized.

In Figure 2.6, bottom, `isSameSet(0, 4)` demonstrates another operation for this data structure. This function `isSameSet(i, j)` simply calls `findSet(i)` and `findSet(j)` and checks if both refer to the same representative item. If they do, then '`i`' and '`j`' both belong to the same set. Here, we see that `findSet(0) = findSet(p[0]) = findSet(1) = 1` is not the same as `findSet(4)= findSet(p[4]) = findSet(3) = 3`. Therefore item 0 and item 4 belongs to *different* disjoint sets.

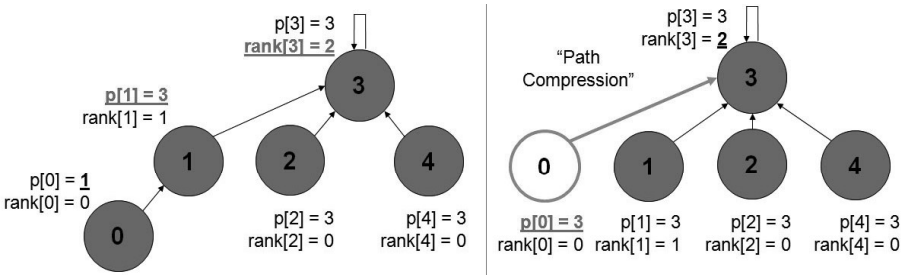

Figure 2.7: `unionSet(0, 3)` → `findSet(0)`

There is a technique that can vastly speed up the `findSet(i)` function: Path compression. Whenever we find the representative (root) item of a disjoint set by following the chain of 'parent' links from a given item, we can set the parent of *all items* traversed to point directly to the root. Any subsequent calls to `findSet(i)` on the affected items will then result in only one link being traversed. This changes the structure of the tree (to make `findSet(i)` more efficient) but yet preserves the actual constitution of the disjoint set. Since this will occur any time `findSet(i)` is called, the combined effect is to render the runtime of the `findSet(i)` operation to run in an extremely efficient amortized $O(M \times \alpha(n))$ time.

In Figure 2.7, we demonstrate this 'path compression'. First, we call `unionSet(0, 3)`. This time, we set `p[1] = 3` and update `rank[3] = 2`. Now notice that `p[0]` is unchanged, i.e. `p[0] = 1`. This is an *indirect* reference to the (true) representative item of the set, i.e. `p[0] = 1` → `p[1] = 3`. Function `findSet(i)` will actually require more than one step to

traverse the chain of 'parent' links to the root. However, once it finds the representative item, (e.g. 'x') for that set, it will *compress the path* by setting `p[i] = x`, i.e. `findSet(0)` sets `p[0] = 3`. Therefore, subsequent calls of `findSet(i)` will be just $O(1)$. This simple strategy is aptly named the 'path compression' heuristic. Note that `rank[3] = 2` now no longer reflects the *true height* of the tree. This is why `rank` only reflects the *upper bound* of the actual height of the tree. Our C++ implementation is shown below:

```
class UnionFind {                                              // OOP style
private: vi p, rank;                            // remember: vi is vector<int>
public:
  UnionFind(int N) { rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {                          // if from different set
      int x = findSet(i), y = findSet(j);
      if (rank[x] > rank[y]) p[y] = x;          // rank keeps the tree short
      else {                 p[x] = y;
                             if (rank[x] == rank[y]) rank[y]++; }
} } };
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/ufds.html`

Source code: `ch2_08_unionfind_ds.cpp/java`

Exercise 2.4.2.1: There are two more queries that are commonly performed in this data structure. Update the code provided in this section to support these two queries efficiently: `int numDisjointSets()` that returns the number of disjoint sets currently in the structure and `int sizeOfSet(int i)` that returns the size of set that currently contains item `i`.

Exercise 2.4.2.2*: Given 8 disjoint sets: {0, 1, 2, ..., 7}, please create a sequence of `unionSet(i, j)` operations to create a tree with rank = 3! Is this possible for rank = 4?

Profiles of Data Structure Inventors

George Boole (1815-1864) was an English mathematician, philosopher, and logician. He is best known to Computer Scientists as the founder of Boolean logic, the foundation of modern digital computers. Boole is regarded as the founder of the field of Computer Science.

Rudolf Bayer (born 1939) has been Professor (emeritus) of Informatics at the Technical University of Munich. He invented the Red-Black (RB) tree used in the C++ STL `map/set`.

Georgii Adelson-Velskii (born 1922) is a Soviet mathematician and computer scientist. Along with Evgenii Mikhailovich Landis, he invented the AVL tree in 1962.

Evgenii Mikhailovich Landis (1921-1997) was a Soviet mathematician. The name of the AVL tree is an abbreviation of the two inventors: Adelson-Velskii and Landis himself.