ICS 311 #16: Disjoint Sets and Union-Find

Outline

1. Disjoint Dynamic Sets
2. Finding Connected Components with Disjoint Sets
3. Linked List Representations of Disjoint Sets - *Skip in 2015: this is not as good as the forest representation*
4. Forest Representations of Disjoint Sets

Readings and Screencasts

- Chapter 21 of CLRS sections 21.1 and 21.3, with focus on 21.3. (You need only know the result of the analysis in 21.4.)
- Screencast 16A (also in Laulima and iTunesU)

Dynamic Disjoint Sets (Union Find)

Two sets A and B are **disjoint** if they have no element in common.

Sometimes we need to group n distinct elements into a collection Š of disjoint sets Š = $\{S_1, ..., S_k\}$ that may change over time.

- Š is a set of sets: $\{\{x, ...\}, ..., \{y, ...\}\}$
- Each set $S_i \in$ Š is identified by a **representative**, which is some member of the set (e.g., x and y).
- It does not matter which member is the representative, as long as the representative remains the same while the set is not modified.

Disjoint set data structures are also known as **Union-Find** data structures, after the two operations in addition to creation. (Applications often involve a mixture of searching for set membership and merging sets.)

Operations

Make-Set(x): make a new set $S_i = \{x\}$ *(x will be its representative)* and add S_i to Š.

Union(x, y): if $x \in S_x$ and $y \in S_y$, then Š ← Š − S_x − S_y ∪ $\{S_x \cup S_y\}$ *(that is, combine the two sets S_x and S_y).*

- The representative of $S_x \cup S_y$ is any member of that new set (implementations often use the representative of one of S_x or S_y.)
- Destroys S_x and S_y, since the sets must be disjoint (they cannot co-exist with $S_x \cup S_y$).

Find-Set(x): return the representative of the set containing x.

Analysis

We analyze in terms of:

- n = number of Make-Set operations, i.e., the number of sets initially involved
- m = total number of operations

Some facts we can rely on:

- $m \geq n$
- Can have at most $n-1$ Union operations, since after $n-1$ Unions, only 1 set remains.
- It can be helpful for analysis to assume that the first n operations are Make-Set operations (put all the elements we will be working with in singleton sets to start with).

Applications of Disjoint Sets

Union-Find on disjoint sets is used to find structure in other data structures, such as a graph. We initially assume that all the elements are distinct by putting them in singleton sets, and then we merge sets as we discover the structure by which the elements are related.

Finding Connected Components

Recall from Topic 14 that for a graph $G = (V, E)$, vertices u and v are in the same **connected component** if and only if there is a path between them.

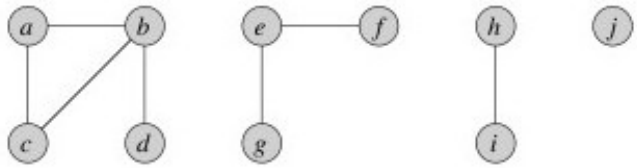
Here are the algorithms for computing connected components and then for testing whether two items are in the same component:

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)
```

```
SAME-COMPONENT(u, v)
1  if FIND-SET(u) == FIND-SET(v)
2      return TRUE
3  else return FALSE
```

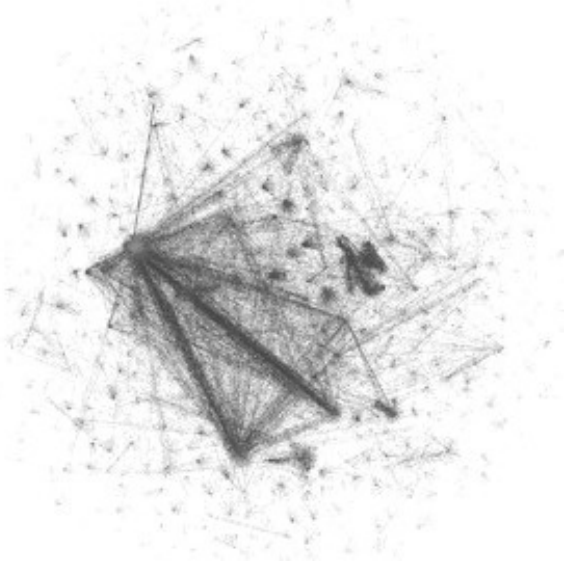
Would that work with a directed graph?

Example



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,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

Although it is easy to see the connected components above, the utility of the algorithm becomes more obvious when we deal with large graphs (such as pictured)!



Alternatives

In a *static* undirected graph, it is faster to run Depth-First Search (exercise 22.3-12), or for static directed graphs the strongly connected components algorithm of Topic 14 (section 22.5), which consists of two DFS. But in some applications edges may be added to the graph. In this case, union-find with disjoint sets is faster than re-running the DFS.

Minimum Spanning Trees

In the next topic we cover algorithms to find *minimum spanning trees* of graphs. Kruskal's algorithm will use Union-Find operations.

Linked List Representations of Disjoint Sets

One might think that lists are the simplest approach, but there is a better approach that is not any more complex: this section is mainly for comparision purposes.

Representation

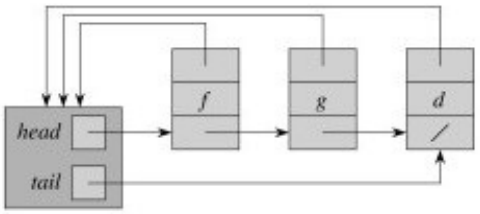
Each set is represented using an unordered singly linked list. The list object has attributes:

- **head**: pointing to the first element in the list, the set's representative.
- **tail**: pointing to the last element in the list.

Each object in the list has attributes for:

- **next**
- The **set member** (e.g., the vertex in the graph being analyzed)

- A pointer to the list object that represents the **set**

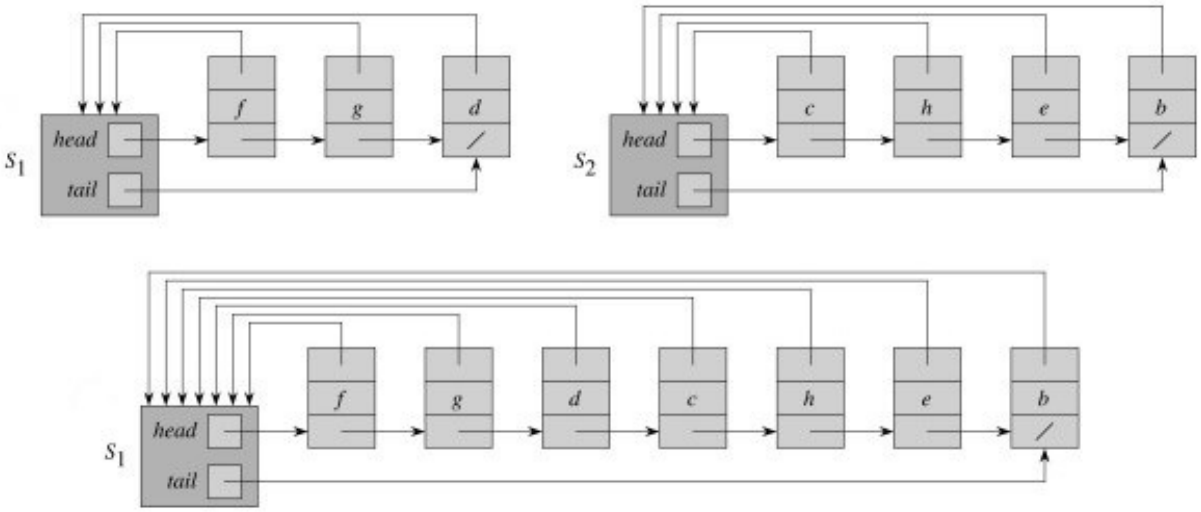


Operations

First try:

- Make-Set(x): create a singleton list containing x
- Find-Set(x): follow the pointer back to the list object, and then follow the head pointer to the representative
- Union(x, y): append y's lists onto the end of x's list.
  - Use x's tail pointer to find the end.
  - Need to update the pointer back to the set object for every node on y's list.

For example, let's take the union of S_1 and S_2, replacing S_1:



This can be slow for large data sets. For example, suppose we start with n singletons and always happend to append the larger list onto the smaller one in a sequence of merges:

Operation	# objects updated
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
UNION(x_5, x_4)	4
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$
	$\Theta(n^2)$ total

If there are n Make-Sets and n Unions, the amortized time per operation is $O(n)$!

A **weighted-union heuristic** speeds things up: always append the smaller list to the larger list (so we update fewer set object pointers). Althought a single union can still take $\Omega(n)$ time (e.g., when both sets have $n/2$ members), a sequence of m operations on n elements takes $O(m + n \lg n)$ time.

Sketch of proof: Each Make-Set and Find-Set still takes O(1). Consider how many times each object's set representative pointer must be updated during a sequence of n Union operations. It must be in the smaller set each time, and after each Union the size of this smaller set is at least double the size. So:

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\lg n$	$\geq n$

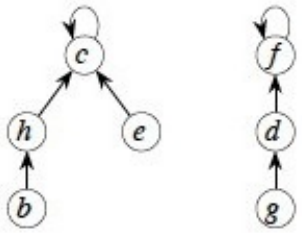
Each representative set for a given element is updated $\leq \lg n$ times, and there are n elements plus m operations. However, we can do better!

Forest Representations of Disjoint Sets

The following is a classic representation of Union-Find, due to Tarjan (1975). The set of sets is represented by a forest of trees. The code is as simple as the analysis of runtime is complex.

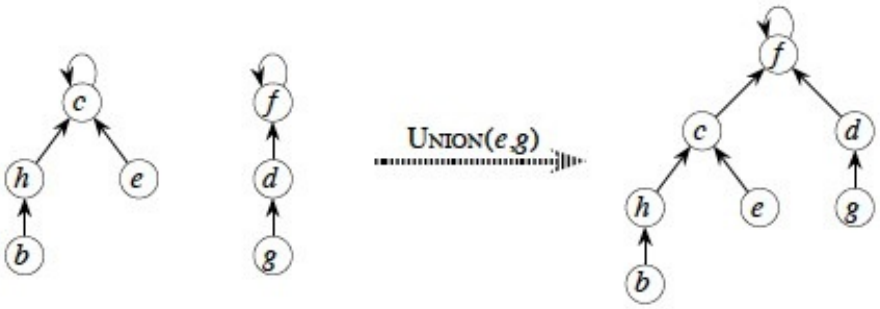
Representation

- Each tree represents a set.
- The root of the tree is the set representative.
- Each node points only to its parent (no child pointers needed).
- The root points to itself as parent.



Operations

- Make-Set(x): create a single node tree with x at the root
- Find-Set(x): follow parent pointers back to the root
- Union(x, y): make one root a child of the other. (This in itself could degenerate to a linear list-like tree, but we will fix this below.)



Heuristics

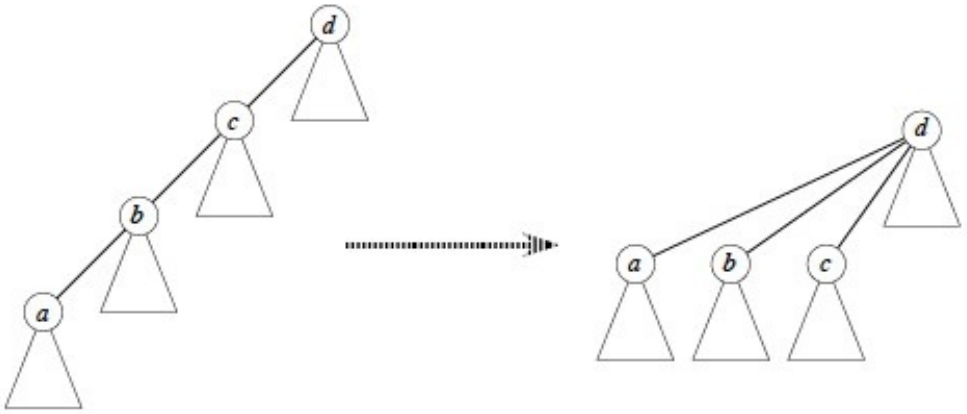
In order to avoid degeneration to linear trees, and achieve amazing amortized performance, these two heuristics are applied:

Union by Rank: make the root of the "smaller" tree a child of the root of the "larger" tree. But rather than size we use **rank**, an upper bound on the height of each node (stored in the node).

- Rank of singleton sets is 0.

- When taking the Union of two trees of equal rank, choose one arbitrarily to be the parent and increment its rank by one. *(Why is it incremented?)*
- When taking the Union of two trees of unequal rank, the tree with lower rank becomes the child, and ranks are unchanged. *(Why does this make sense?)*

Path Compression: When running Find-Set(x), make all nodes on the path from x to the root direct children of the root. For example, Find-Set(a):



Algorithms

The algorithms are very simple! (But their analysis is complex!) We assume that nodes x and y are tree nodes with the client's element data already initialized.

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2        y.p = x
3  else x.p = y
4        if x.rank == y.rank
5              y.rank = y.rank + 1

FIND-SET(x)
1  if x ≠ x.p
2        x.p = FIND-SET(x.p)
3  return x.p
```

Link implements the union by rank heuristic.

Find-Set implements the path compression heuristic. It makes a recursive pass up the tree to find the path to the root, and as recursion unwinds it updates each node on the path to point directly to the root. (This means it is not tail recursive, but as the analysis shows, the paths are very unlikely to be long.)

Time Complexity

The analysis can be found in section 21.4. It is very involved, and I only expect you to know what is discussed below. It is based on a very fast growing function:

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0\,, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1\,, \end{cases}$$

$A_k(j)$ is a variation of **Ackerman's Function**, which is what you will find in most classic texts on the subject. The function grows so fast that $A_4(1) = 16^{512}$ is *much* larger than the number of atoms in the observable universe (10^{80})!

The result uses **$\alpha(n)$,** a single parameter inverse of $A_k(j)$ defined as the lowest k for which $A_k(1)$ is at least n:

$$\alpha(n) = \min\{k : A_k(1) \geq n\}$$

n	$\alpha(n)$
0–2	0
3	1
4–7	2
8–2047	3
2048–$A_4(1)$	4

$\alpha(n)$ grows *very* slowly, as shown in the table. We are highly unlikely to ever encounter $\alpha(n) > 4$ (we would need input size much greater than the number of atoms in the universe). Although its growth is strictly larger than a constant, for all practical purposes we can treat $\alpha(n)$ as a constant.

The analysis of section 21.4 shows that the running time is **$O(m\,\alpha(n))$** for a sequence of m `Make-Set`, `Find-Set` and `Union` operations, or **$O(\alpha(n))$** per operation. Since $\alpha(n) > 4$ is highly unlikely, for all practical purposes the cost of a sequence of m such operations is $O(m)$, or $O(1)$ amortized cost per operation!!

Wrapup

We now return to Graphs. We'll see Union-Find used when we cover minimum spanning trees.

Dan Suthers
Last modified: Thu Mar 26 15:42:07 HST 2015
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition.