CS 261: Data Structures

Week 10: Odds and ends

Lecture 10b: Graphs and union-find

David Eppstein
University of California, Irvine

Spring Quarter, 2025

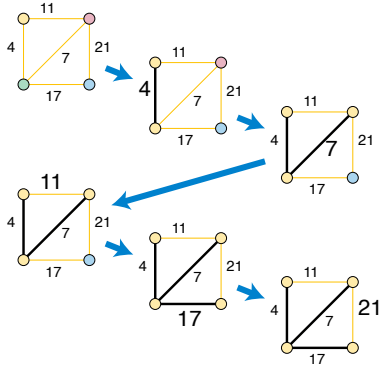
Dynamic graph algorithms

Kruskal's algorithm for minimum spanning trees

Given an undirected graph with weighted edges:

- Start with a forest of one-vertex trees
- For each edge in sorted order by weight, if it connects two different trees, add the edge to the forest

How to perform the connectivity test?



Dynamic graph algorithms

Called algorithms but really data structures

A graph is being modified by local update operations (here, we are adding edges to a forest)

We want to maintain information or answer queries more quickly than recomputing from scratch (here, do two query vertices belong to the same tree of the forest?)

Classification

Most dynamic graph algorithms maintain information about graphs that change by the insertion and deletion of edges

Fully versus partially dynamic:

- Incremental: Only insertions, no deletions
- Decremental: Only deletions, no insertions
- Fully dynamic: Both

Online versus offline:

- Online: Updates and queries must be handled immediately, without knowing what future operations are going to be
- Offline: The whole sequence of operations is known in advance

Dynamic undirected connectivity

Query: Either name the component containing a given vertex, or test whether two vertices are in the same component

Fully dynamic updates: Insert or delete one edge

- Fastest known data structure: Huang, Huang, Kopelowitz, and Pettie, "Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time", SODA 2017
- Lower bound $\Omega(\log n/\log\log n)$ from prefix sums: Fredman and Henzinger, "Lower bounds for fully dynamic connectivity problems in graphs", *Algorithmica* 1998
- Many other publications

Reachability in directed acyclic graphs

Incremental: $O(1)$ per query, $O(n)$ per edge addition (Italiano, *Theor. Comp. Sci.* 1986; La Poutré and van Leeuwen, WG 1987)

Fully dynamic: $O(1)$ per query, randomized, can make false negatives but never false positives, $O(n^2)$ per update (King and Sagert, "A Fully Dynamic Algorithm for Maintaining the Transitive Closure", JCSS 2002)

Matching

Matching: A large set of edges that don't share any endpoints

Incremental: Can maintain $(1+\epsilon)$-approximation to largest matching in constant amortized time per insertion
(Grandoni et al., SODA 2019)

Fully dynamic:

- Largest matching in time $O(n^{1.495})$ per operation
  (Sankowski, SODA 2007)
- 2-approximation in constant amortized time
  (Solomon, FOCS 2016)
- Various more-accurate approximations with
  polynomial times per update

More

Dynamic versions of many other graph problems have been studied

This is an active area of research with both a long history and many recent developments

For a one-hour survey talk by Liam Roditty from 2016, see
`https://www.youtube.com/watch?v=oZGSdfyU_YU`

Union-find

Some history of incremental connectivity

Galler and Fischer 1964: Devised an efficient data structure solving this problem of connected components with edge additions (what we need for Kruskal's algorithm)

Tarjan 1975: Proved a nearly-constant but not constant amortized time bound for this structure

Tarjan 1979: Found examples for which this structure's time is not constant (showing that Tarjan's earlier analysis was tight)

Fredman and Saks 1989: Proved that this time bound is optimal (no other structure for the same problem has better O-notation)

Equivalence between forests and sets

Each tree of the forest $\Rightarrow$ set of its vertices

Forest of trees $\Rightarrow$ partition of the vertices into disjoint sets
(meaning that each element belongs to only one set)

Add edge to the forest, merging two trees into a single tree $\Rightarrow$ merge two sets into a single set, their union

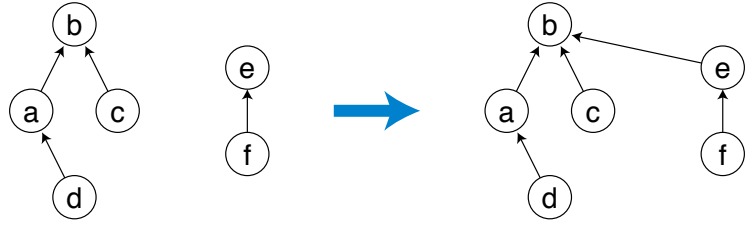
Test which tree contains a given vertex v $\Rightarrow$ find the set containing a given element

This problem and its solution are sometimes called union-find or the disjoint sets problem

Sets as forests

We can represent any family of disjoint sets as a forest
(many different representations are possible; it doesn't have to be the same forest as the one in Kruskal's algorithm)

- Set elements = forest vertices, each with pointer to parent
- Identify each set with its root (the element with no parent)
- Find the set containing an element by following path to root
- Merge two sets by adding an edge from one root to the other



Optimization 1: Union by size

Store at each tree root the number of nodes in that tree

Can be updated in constant time whenever we merge trees

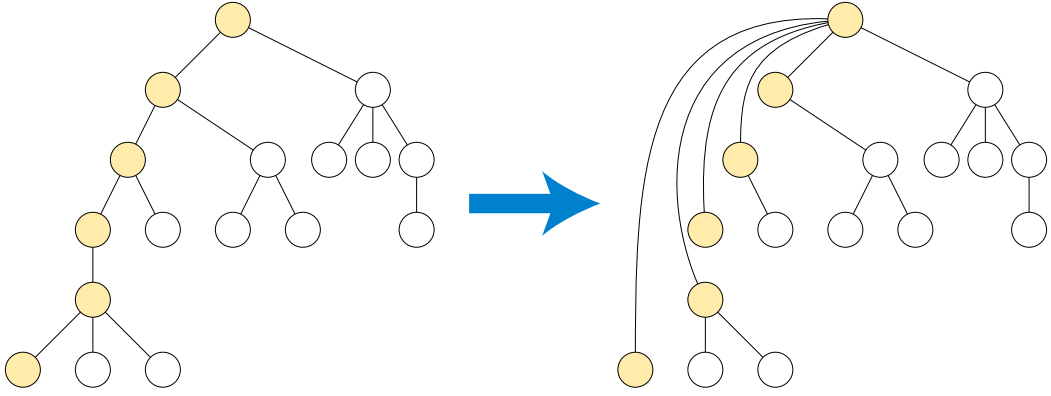
When we merge two trees, we can choose which direction to add the new edge between the two roots; always make the smaller root become a child of the larger one

With this optimization, every node has
size of its subtree $\leq \frac{1}{2}$ size of its parent's subtree
$\Rightarrow$ path to root can be at most $\log_2 n$ steps

Find takes $O(\log n)$ time, union takes $O(1)$ time

Optimization 2: Path compression

When a find operation follows a (possibly long) path to the root of its tree, change tree so all nodes on path point directly to the root



Speeds up any later find operations using any of the same nodes

But how big is the speedup?

Pseudocode for combined optimizations

New structure with given elements, each in their own set:

- size = dictionary mapping each x to 1
- parent = dictionary mapping each x to None

Merge sets with roots s, t:

- If size[s] $<$ size[t]:
  swap s and t
- Add size[t] to size[s]
- Set parent[t] to s

Find root of set containing x:

- If parent[x] is None:
  return x
- y = find(parent[x])
- Set parent[x] to y
- Return y

Union-find analysis

Some quickly-growing functions

$$f_0(x) = 2x$$

values (for $x = 0, 1, 2, 3, \ldots$) are 0, 2, 4, 6, 8, 10, 12, ...

$$f_1(x) = 2^x$$

repeatedly double (apply f_0) x times, starting from 1

values are 1, 2, 4, 8, 16, 32, 64, ...

$$f_2(x) = 2 \uparrow\uparrow x = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{x \text{ times}}$$

values are 1, 2, 4, 16, 65536, $\approx 2 \times 10^{19728}$, ...

...

$$f_i(x) = 2\underbrace{\uparrow\uparrow \ldots \uparrow}_{i \text{ times}}x = \underbrace{f_{i-1}(f_{i-1}(\ldots(f_{i-1}(2))\ldots))}_{x \text{ times}}$$

The Ackermann function

We can put all these functions together into a single function $F(i,x) = f_i(x)$, defined (with appropriate base cases) by the recurrence equation $F(i,x) = F(i-1, F(i,x-1))$

The Ackermann function $A(i,j)$ uses the same recurrence $A(i,j) = A(i-1, A(i,j-1))$ but (for historical reasons) different base cases: $A(0,j) = j+1$ and $A(i,0) = A(i-1,1)$

Some sources use different base cases than either of these

	j=0	1	2	3	4	...	
i=0	1	2	3	4	5	...	$j+1$
1	2	3	4	5	6	...	$j+2$
2	3	5	7	9	11	...	$2(j+3)-3$
3	5	13	29	61	125	...	$2^{j+3}-3$
4	13	65533	$\approx 2 \times 10^{19728}$	...	...	...	$2 \uparrow\uparrow (j+3) - 3$

For $i \geq 2$, $A(i,j) = f_{i-2}(j+3) - 3$, very quickly growing

The one-parameter inverse Ackermann function

Consider the diagonal of the table of values of the Ackermann function:

$$A(i,i) = 1, 3, 7, 61, \mathbf{??}, \ldots, \text{ for } i = 0, 1, 2, 3, 4, \ldots$$

Here the **??** is so big that writing it down in binary notation, using the space of an atom for each bit, would take more volume than the known universe

Define $\alpha(n)$ to be the smallest i for which $A(i,i) \geq n$

Then α is non-constant in theory but ≤ 4 in practice

Claims: for union-find with n elements, all operations take amortized time $O(\alpha(n))$, and no other data structure can do better

The two-parameter inverse Ackermann function

For union-find, it's convenient to analyze the algorithm in terms of two parameters:

- n: the number of elements in the disjoint sets

  In Kruskal's algorithm, the number of vertices
- m: the number of find operations we perform

  In Kruskal's algorithm, $2\times$ the number of edges

More find operations $\Rightarrow$ more paths replaced by edges directly to the roots $\Rightarrow$ operations become faster

Define $\alpha(m, n)$ to be the smallest i for which $A(i, m/n) \geq \log_2 n$

Constant when m grows slightly more quickly than n, e.g. $m = \Theta(\log \log n)$, but non-constant for $m = \Theta(n)$

Claim: Amortized time for union-find is $O(\alpha(m, n))$ per find

Content warning

The rest of this lecture sketches a proof of this time bound

There won't be any course assignments based on it

It's more important to understand what the analysis means
(how fast are the union-find operations) than how to prove it

Shortcut-counting

Time for a find operation
$= O(1) + O(\text{number of times we change someone's parent})$

But any single element's parent can only change $\log_2 n$ times because size of its parent's subtree doubles after each change

So total time for m finds with n elements is $O(m + n \log n)$

This is better than worst-case time per operation, which gives $O(n + m \log n)$, because it gives the more frequent operation (finds) the smaller amortized time $O(1)$ per operation

First-parent tree

If there are multiple disjoint sets at the end of a sequence of union-find operations, add more union operations to merge them into a single set (doesn't significantly change overall analysis)

Define the first-parent tree to be a tree where

- Vertices are the set elements
- The parent of the vertex is the first parent it gets in the union-find data structure, before any shortcuts are made

This tree may not describe any actual state of the data structure!

We will use it to split the problem into smaller subproblems

Each change to parent of x moves to a higher ancestor in this tree

Big and small elements

Choose a parameter s (to be determined later)

Define an element x to be small if it has fewer than s descendants (including itself) in the first-parent tree, and big if it is not small

Claim: at most $2n/s$ elements are big

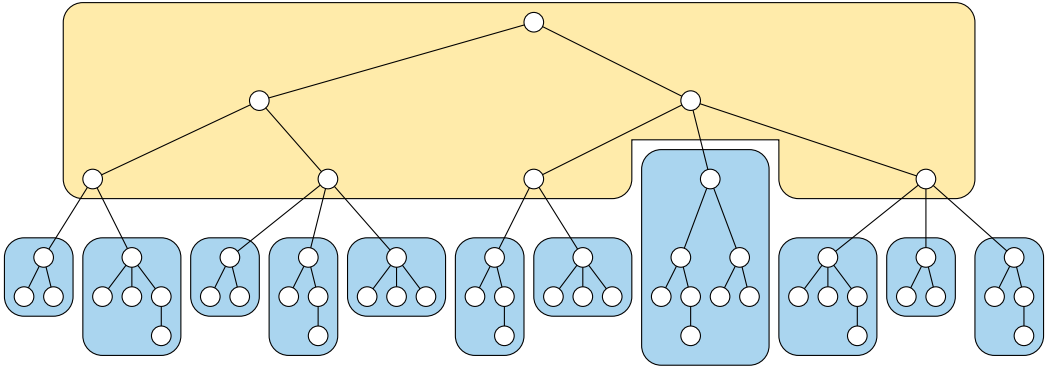
Proof idea:

- If x is big, and h steps above its farthest big descendant y, then y's subtree has size $\geq s$, so x's subtree has size $\geq 2^h s$
- Nodes that are h steps above their farthest big descendant have disjoint subtrees, so there are at most $n/(2^h s)$ of them.
- Summing over all choices of h gives at most $n/s + n/2s + n/4s + \cdots = 2n/s$ big elements.

Big and small subtrees

Let T_0 be the subtree of the first-parent tree consisting only of the big elements

For each x that is small but has a big parent in the first-parent tree, make a tree T_x consisting of x and all its descendants



Then T_0 has size $O(n/s)$ and all small trees T_x have size $O(s)$

Partition into smaller union-find problems

Each time we perform find(x) for some x, we follow a path through some of the ancestors of x

- If the topmost element of the path is still inside a small tree T_y, charge the operation to T_y
- If the parent of x at the start of the find belongs to the big tree T_0, charge the operation to T_0
- In the remaining case, charge part to T_y and part to T_0

So m finds in a single union-find data structure of n elements are equivalent to:

- Some number m_0 of finds in the big tree T_0 of size $O(m/s)$
- Some number m_x of finds in the small trees T_x of size $O(s)$
- $m_0 + \sum_x m_x \leq m + n$, where the $+n$ is from the finds that cross over from small to big

Reduction in growth rate

Suppose we already know that $\text{time}(m, n) = O(m + ng_i(n))$,
where g_i is the inverse function to $f_i(n)$
(for example if $i = 1$, $f_i = 2^n$ and $g_i = \log n$, and we do know this).

Choose $s = g_i(n)$ and look at steps in the big subtree:
$\text{time}(m_0, n/s) = O(m_0 + (n/s)g_i(n) = O(m_0 + n)$
$\Rightarrow$ For this s, time for operations in the big tree is linear

Plugging this into the total time gives
$\text{time}(m, n) = O(m_0 + n) + \sum_x \text{time}(m_x, \text{size}(T_x))$
$\Rightarrow$ we have eliminated T_0 and reduced all trees to size $\leq s$
but in exchange, we added an extra $+O(n)$ to the time bound

Repeat in the small subtrees with $s = g_i(g_i(n))$, in the small subtrees of small subtrees with $s = g_i(g_i(g_i(n)))$, etc

gives an additional $+O(n)$ at each level of expansion

$\Rightarrow$ time is $O(m)$ + number of levels $\times O(n) = O(m + ng_{i+1}(n))$

Completing the analysis

Previous slide: $O(m + ng_i(n)) \Rightarrow O(m + ng_{i+1}(n))$

So for every constant i, we have $O(m + ng_i(n))$

This is already enough to tell us that the time is very close to linear, because $g_2(n)$ (the smallest height of a tower of powers of two that is $\geq n$, also called $\log^* n$) is already very slowly growing

But what if we do the analysis without assuming that i is constant?

If i is not constant, we cannot remove functions of i from O-notation, so time becomes $O\big(f(i)(m + ng_i(n))\big)$ for some f

We can choose i to be any natural number (depending somehow on n and m), getting more slowly-growing functions g_i for larger i, but also getting larger factors $f(i)$ in the analysis.

Choose i to balance increase versus decrease $\Rightarrow O(m\alpha(m, n))$

Summary

Summary

- Suffix arrays and their construction
- Tries and compressed tries
- Suffix tree = compressed trie of suffixes
- Using suffix trees to find substrings and solve other problems on strings
- Union-find and its application in Kruskal's algorithm
- Union-find analysis
- Overview of dynamic graph algorithms