MST: Kruskal's Algorithm

Sam McCauley

March 6, 2025

Welcome Back!

- Both homeworks back! (Shout out to TA Charlotte for the fast turnaround.)
- Homework due today: solution posted (including Kruskal's)
- Practice midterm: I realized it has no longer proof-style questions. The problem sets are good practice for those. Practice midterm solutions are posted
- I'll get Problem Sets 2 and 3 graded as soon as I can
- Plan for today: first half of the class we'll do Kruskal's algorithm; I'll set an alarm and second half of class we'll do review.
- Any questions before we start??

Prim’s Wrapup

Prim's Algorithm

- Let's do an example on the board
- Recall: maintain a set S of vertices, all connected by a set of edges T. At each time, find the *cut edge* (edge with one vertex in S, one vertex not in S) with minimum weight; add it to T.
- Recall: we assume for simplicity that all edge weights are positive and distinct. (Easy to generalize!)

Recall: Cut Property of MST

A *cut* is a partition of the vertices V into two subsets: S, and $V \setminus S$. A *cut edge* is an edge with one endpoint in S and the other in $V \setminus S$.

Lemma

Let G be a graph where all edge weights are distinct. For any cut S, let $e = (u, v)$ be the minimum weight cut edge. Then e is in every minimum spanning tree of G.

Side note: this lemma implies that there is actually only one minimum spanning tree if all edge weights are distinct.

Proving Prim's Correct

- **In pairs:** How can we use the cut property to prove Prim's algorithm correct?

- Answer: Every edge we add is the smallest cut edge between S and $V \setminus S$; by the cut property it is in every MST.

Implementing Prim's Algorithm

What do we need to be able to do?

- Maintain all cut edges!
- Must be able to insert new edges when adding a vertex to S
- Must be able to find minimum-weight cut edge (i.e. minimum-weight edge in the data structure) and remove it
- Priority Queue!
- Note that: we will (again!) wind up with some edges from S to S in the data structure (why?). If we remove such an edge we'll just skip it.
    - Reason: when we add v to S, there could be some extra edges to v already in the priority queue

Prim's Algorithm (Jarník's Algorithm)

First, choose a starting vertex u. Create a set of vertices, starting with $S \leftarrow \{u\}$ and a tree starting with $T \leftarrow \emptyset$. Add all edges adjacent to u to a priority queue PQ.

While $|T| \leq n - 1$, find the min-cost edge $e = (u, v)$ such that one end $u \in S$ and $v \in V \setminus S$. Set $T \leftarrow T \cup \{e\}$ and $S \leftarrow S \cup \{v\}$.

To implement: each time we add a vertex to S, add its incident edges to PQ. To find the minimum cut edge, remove edges from PQ until we find a cut edge. Cost?

Need to do $\leq 2m$ inserts, and $\leq 2m$ extract mins (why?).

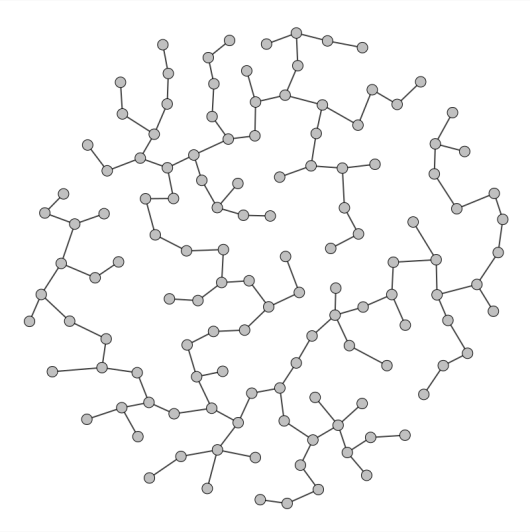
Running time: $O(m \log m)$.

As with Dijkstra's, can use Fibonacci heap to improve to $O(m + n \log n)$.

Prim's Algorithm Pseudocode

```
1  Prims(G):
2      pick a vertex v as the starting vertex
3      let T and S be empty sets and pq be an empty priority
           queue
4      add v to S
5      for each edge e′ adjacent to v:
6          pq.insert(e′)
7
8      while pq is not empty:
9          e = pq.removeMin()
10         if e has an endpoint w ∉ S:
11             add e to T; add w to S
12             for each edge e′ adjacent to w:
13                 pq.insert(e′)
```

MST



Kruskal’s Algorithm

Another MST Approach

- In Prim's: we grew from a starting vertex s
- Idea: the cheapest neighbor of s must be in the tree
- Let's look at the whole graph. (No starting vertex.) Is there an edge that we know must be in the graph?
    - Either intuitively or using the cut property
- Answer: The *smallest* edge in the whole graph must be in the minimum spanning tree
- Why does this follow from the cut property?

Building to Kruskal's Algorithm

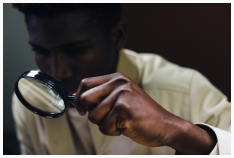
- Repeatedly: add the smallest remaining edge in the graph(?)
- Can you come up with an example of when you don't want to add the smallest remaining edge?
- Answer: we don't want to add the edge if it is between two vertices that are already connected by the minimum spanning tree [Blackboard]
- With this exception we have Kruskal's algorithm

Kruskal's Algorithm

```
1    Let T be an empty set of edges
2    while |T| < n − 1:
3        let e be the lowest-weight edge in G
4        remove e from G
5        if adding e to T does not create a cycle:
6            add e to T
```

- That's it! Let's do an example [Blackboard]
- Why does the cut property imply that Kruskal's is correct? [Blackboard]
- What operations do we need to make this work?
  - Find smallest edge in G
  - Find out of an edge e creates a cycle

Finding the Smallest Edge in G

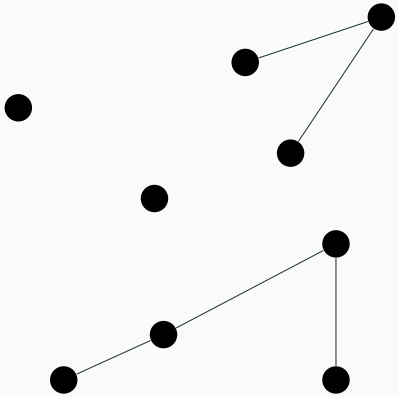


- **In pairs:** can you come up with a simple data structure that allows us to repeatedly find the smallest remaining edge in G?
- A priority queue is one option
- Another option (how it's usually implemented): put all edges into an array and sort them by weight. Keep track of which one was chosen most recently
- Time to sort all edges is $O(m \log m)$. So if we can find if an edge creates a cycle in $O(\log m)$ time, Kruskal's takes $O(m \log m)$ total time.
  - We will. Kruskal's takes $O(m \log m)$ total time.
- Let's talk about how to detect a cycle. Everything from here on out is good practice for the midterm. But: *you do not need to know* for the midterm.

Determining if an Edge Creates a Cycle

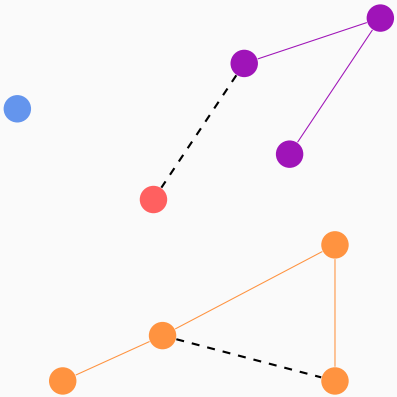
- Any thoughts? Not an easy problem to solve!
- Can run DFS on T each time to see if it contains a cycle in $O(n)$ time
- Can we do better?

What it Means for an Edge to Create a Cycle



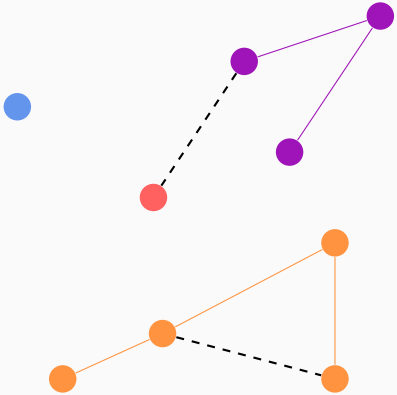
- At all times, T is a *forest*: a collection of trees
- A new edge e is either between two different trees (no cycle), or between two vertices in the same tree (does create a cycle)
- So: we want to find out if the endpoints in e are in the same (connected) tree of T

What it Means for an Edge to Create a Cycle



- We want to find out if the endpoints of e are in the same subtree of T
- What if we could *label* the trees, and quickly find out the label for each vertex?
- We'd be done: an edge (u, v) creates a cycle if and only if u and v have the same label

Labels on Trees



- Maintain *labels* on all vertices so that two vertices have the same label if they are in the same tree
- Query: given a vertex v, what is its label?
  - We'll call this a **find**() operation
- What happens when we insert a new edge?
- Must merge the two trees
  - We'll call this a **union**() operation
- Notice: we don't really need to keep track of the tree structure...

Union-Find Data Structure

- Classic data structure on sets of items. To begin, all v_i are in their own set, each with its own unique label
  - This works for any set, but visually we'll discuss it for vertices
- Can handle three operations:
  - **initialize(n):** build the data structure with all n v_i in their own set
  - **find**(v_i)**:** find which set v_i is in.
  - **union(v_i, v_j):** take the set containing v_i and the set containing v_j, and combine them into a single set
- This is one of my favorite data structures. We'll start by building some not-so-efficient ways to do this. We'll quickly get to a *very* efficient methodology

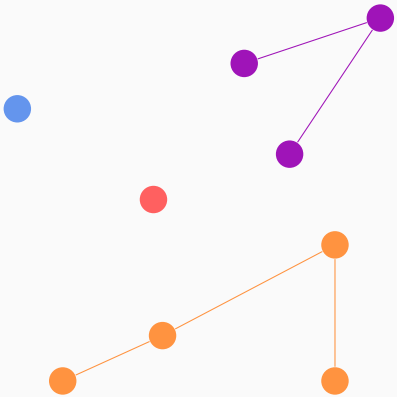
Union-Find: Any Ideas?

- **In pairs:** can you get a data structure with $O(1)$ **find**(), and $O(n)$ **union**()?
  - Hint: we'll be maintaining labels
  - First, come up with intuition for how to do this. Then, try to be specific about exactly how your data structure works—whether you store information in a linked list/an array/etc.
- Answer: maintain a label for each vertex (e.g. in an array of length n). **find**(v_i) can be done by looking up the ith entry of the array. For **union**(v_i, v_j), we first find the labels ℓ_i and ℓ_j. Then, we go through the whole array, replacing all instances of ℓ_j with ℓ_i

Union-Find: Improving Union

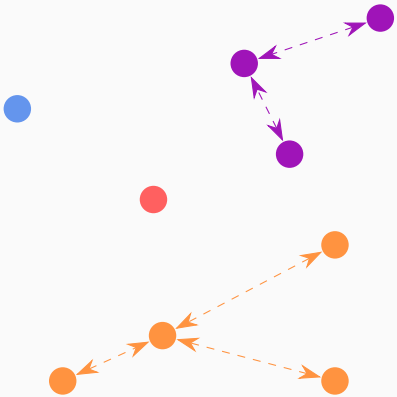
- How can we avoid the $\Theta(n)$ time search for union?
- Can't store an array—don't have the time to update it
  - It takes $\Omega(n)$ time just to *find* the items with a given label using the array approach
  - Need to somehow "link" the items together
- Idea: Let's store a special "head" vertex in each set. Store a pointer from any vertex in the set to the head vertex; from the head vertex to all other vertices in the set.

Union-Find: Improving Union



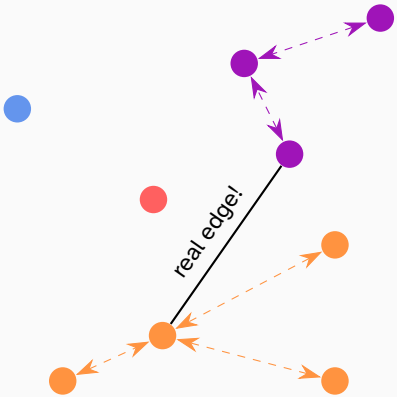
- Idea: Let’s store a special “head” vertex in each set. Store a pointer from any vertex in the set to the head vertex; from the head vertex to all other vertices in the set.

Union-Find: Improving Union



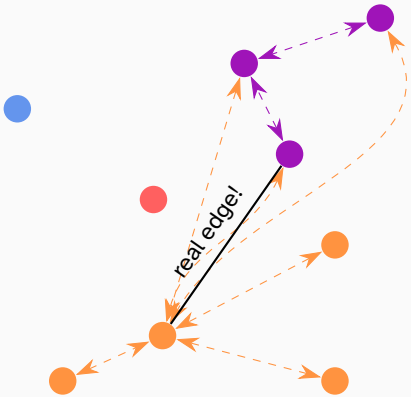
- Idea: Let's store a special "head" vertex in each set. Store a pointer from any vertex in the set to the head vertex; from the head vertex to all other vertices in the set.
- Not necessarily graph edges!! We're only using these to keep track of which vertices are in the same subtree
- How to do **find**(v_i)?
  - Either v_i is the head; or follow pointer to head and return it.
- How to do **union**(v_i)?

Union-Find: Improving Union



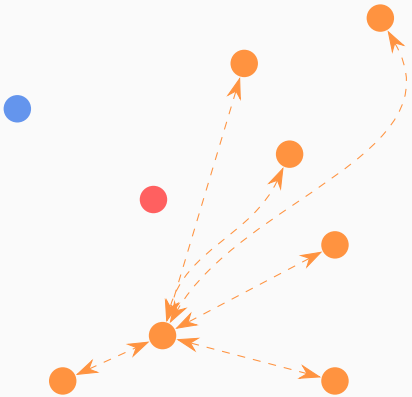
- How to do **union**(v_i)?
- Find the head of both sets. Pick one; make all vertices in the other set point to that head.
- What's the running time?
  - $O(k)$ if smaller set has k elements
  - Could be $\Theta(n)$ in the worst case
- Which head should we pick?
- Idea: Keep track of the *size* of each set. Make the smaller head point to the larger head

Union-Find: Improving Union



- How to do **union**(v_i)?
- Find the head of both sets. Pick one; make all vertices in the other set point to that head.
- What's the running time?
  - $O(k)$ if smaller set has k elements
  - Could be $O(n)$ in the worst case
- Which head should we pick?
- Idea: Keep track of the *size* of each set. Make the smaller head point to the larger head

Union-Find: Improving Union



- How to do **union**(v_i)?
- Find the head of both sets. Pick one; make all vertices in the other set point to that head.
- What's the running time?
  - $O(k)$ if smaller set has k elements
  - Could be $O(n)$ in the worst case
- Which head should we pick?
- Idea: Keep track of the *size* of each set. Make the smaller head point to the larger head

Tighter Analysis

- Some union operations are $\Omega(n)$
- Does this mean that the *total* cost over all operations is $\Omega(n^2)$
- Surprisingly: No—because we always picked the head of the smaller set!
  - I think it's surprising that this choice actually affects asymptotics.
  - Let's look at this in more detail

Tighter Analysis

Course note: You won't be tested on the remainder of the union-find data structure analysis. In particular, you don't need to know or use the "on average" argument we're about to use.

Lemma

Over n union operations using the above method, the total cost is $O(n \log n)$

Proof Sketch: Let's each vertex v_i is reassigned a new head.

If vertex v_i points to a new head, the size of the set containing v_i at least doubled. (Why?) Because we redid the pointers in the smaller set!

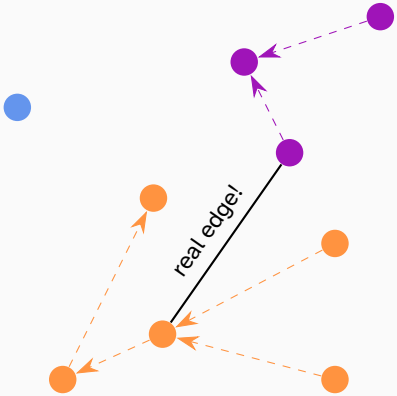
Since v_i starts in a set of size 1, and cannot be in a set of size $> n$, v_i can only be reassinged to a new head $O(\log n)$ times.

Union is $O(k)$ where k is the size of the smallest set. The above argument means that $\sum k = O(n \log n)$. So the total cost of all union operations is $O(n \log n)$.

Improved Union-Find

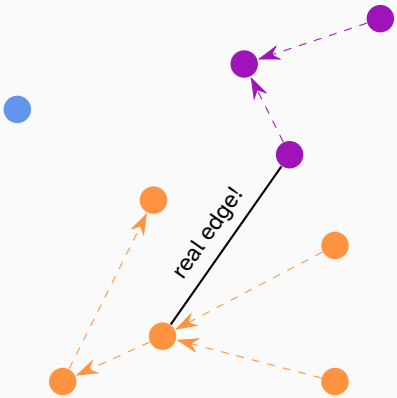
- We have $O(1)$ **find**() per operation, and all union operations cost $O(n \log n)$ total
- Essentially $O(\log n)$ “on average”; this is called the “amortized” cost. You will not be tested on analyzing amortized algorithms.
- Can we improve even further? What if we want $O(\log n)$ find and $O(1)$ union? (Detail: we’ll run **find**() on each vertex before taking the union.)

Fast Union; Slower Find



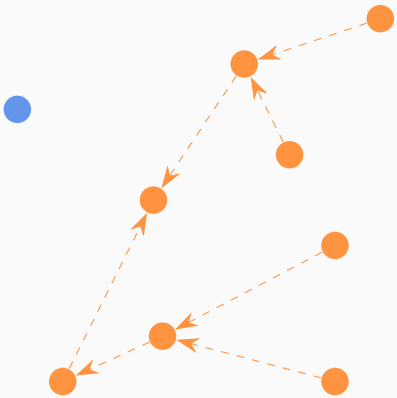
- Keep a head node as before
- Nodes don't point straight to the head; keep them in an "up tree"
- How does **find**() work for this data structure?
  - Follow pointers! Running time?
  - $O(h)$ if height is h
- How does **union**() work for this data structure?

Fast Union; Slower Find



- How does **union**() work for this data structure?
- Point one head node to the other and we're done!
  - No longer need back pointers
- Which one should become the new head?
  - The tree of smaller height
- Time? $O(1)$ after we do a **find**()
- Exercise at home: show that if we always point the smaller-height node to the larger-height node, all trees have height $O(\log n)$

Fast Union; Slower Find



- How does **union**() work for this data structure?
- Point one head node to the other and we're done!
  - No longer need back pointers
- Which one should become the new head?
  - The tree of smaller height
- Time? $O(1)$ after we do a **find**()
- Exercise at home: show that if we always point the smaller-height node to the larger-height node, all trees have height $O(\log n)$

Where do you think we're going?

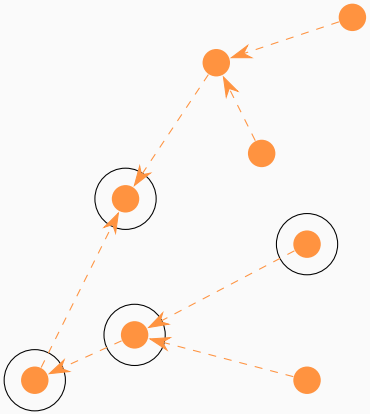
- (This is a just-for-fun topic.)
- Do you think we can do better? Which do you think is the case?

1. Either **union**() or **find**() take $\Omega(\log n)$
2. If you multiply the time to do **union**() and **find**() the product must be $\Omega(\log n)$
3. Both can be $O(1)$
4. Something in the middle

Union-Find: The True Data Structure

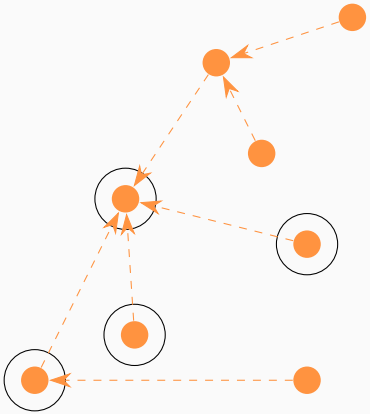
- Let’s start with some practical improvements
- Then I’ll talk about running time

Saving Work for the Future: Path Compression



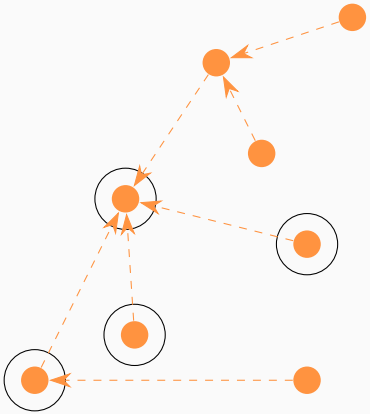
- When we're doing a **find**(), is there work we can do that makes future **find**() operations faster?
  - The running time of **find**() is proportional to the height of the tree. Degree doesn't really matter!
  - We can take any nodes visited during the **find**() and point them to the head of the tree directly
  - Called path compression

Saving Work for the Future: Path Compression



- When we're doing a **find**(), is there work we can do that makes future **find**() operations faster?
  - The running time of **find**() is proportional to the height of the tree. Degree doesn't really matter!
  - We can take any nodes visited during the **find**() and point them to the head of the tree directly
  - Called path compression

Saving Work for the Future: Path Compression

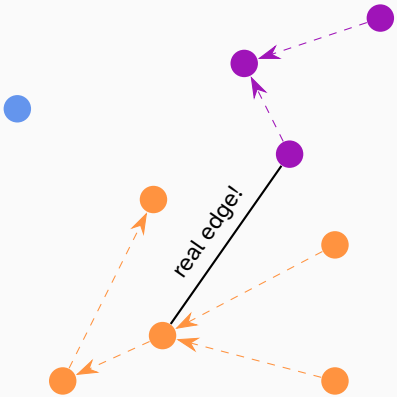


- Any subsequent call to **find()** on any of these vertices—before the head changes—takes $O(1)$ time
- Any *downsides* to doing this?
- Recall: for the union, we pointed the smaller-height node to the larger-height node
- These shortcuts change the height! Can we resolve this??

Keeping Track of the Height

- No, we *can't* resolve this. We can't keep track of the height quickly.
- What can we do instead?
- Let's just....not fix it
- Rather than keeping the height, we'll call it the "rank"
- All ranks start at 0
- If we point a smaller-rank tree to a larger-rank tree, its rank stays the same
- If two trees have the same rank, point one to the other, increment the rank

Keeping Track of the Height



- If we point a smaller-rank tree to a larger-rank tree, its rank stays the same
- If two trees have the same rank, point one to the other, increment the rank

Have we changed the asymptotics?

- We've made some **find**() operations faster—though it's no longer clear that the height really stays $O(\log n)$. **union**() is still fast
- Surprisingly: yes it's better than $O(\log n)$
- The running time for any n union and find operations is $O(n \log^* n)$ [Hopcroft Ullman 1973]
  - $\log^* n$ is the number of times you need to apply the $\log_2$ function to get to a number ≤ 1

n	2	4	$2^4 = 16$	$65536 = 2^{16}$	2^{65536}
$\log^* n$	1	2	3	4	5

This is $O(1)$ for all intents and purposes. Is this analysis tight? Is it actually $O(1)$?

Union-Find True Running Time

- [Tarjan 1975]: n union and find operations require $\Theta(n \cdot \alpha(n))$ time using this data structure
  - $\alpha(n)$ is the *inverse Ackermann function*
  - $\alpha(n)$ is (essentially) the number of times you need to apply $\log^*$ to get to 2
  - For example: $\alpha(2^{2^{2^{2^{16}}}}) = 4$
  - Ridiculously slowly growing
- This is *optimal*; any data structure for union-find requires $\Omega(\alpha(n))$ time

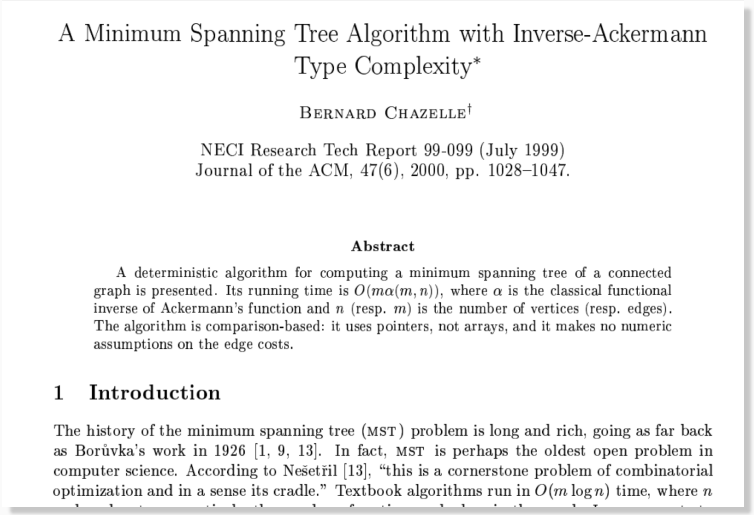
Takeaways

- A couple simple heuristics on top of a simple data structure leads to outstanding performance, and some really cool math
- Kruskal's running time?
  - $O(m \log m)$ to sort all the edges
  - $O(m \cdot \alpha(m))$ for all union-find operations to detect cycles
  - $O(m \log m)$ overall

MST Algorithms

- Prims: $O(m \log m)$ using a binary heap; $O(m + n \log n)$ using a Fibonnacci heap
- Kruskal's: $O(m \log m)$
- Kruskal's is usually better in practice (sorts are easy to optimize)
- Is it possible to do better than $m \log m$ time?

Best Known MST Algorithm



A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity*

BERNARD CHAZELLE†

NECI Research Tech Report 99-099 (July 1999)
Journal of the ACM, 47(6), 2000, pp. 1028–1047.

Abstract

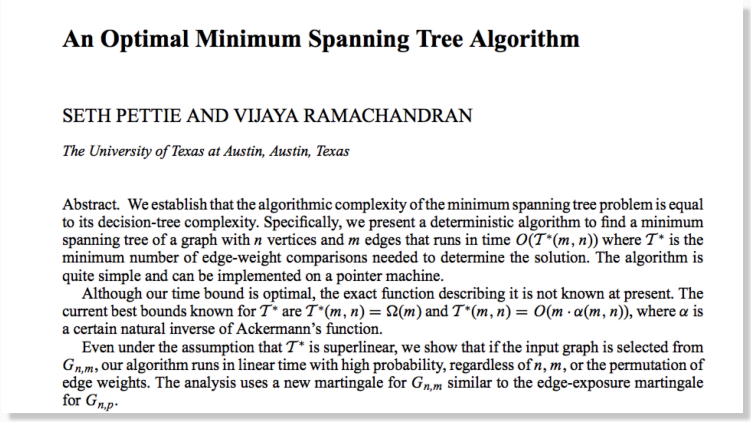
A deterministic algorithm for computing a minimum spanning tree of a connected graph is presented. Its running time is $O(m\alpha(m,n))$, where α is the classical functional inverse of Ackermann's function and n (resp. m) is the number of vertices (resp. edges). The algorithm is comparison-based: it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.

1 Introduction

The history of the minimum spanning tree (MST) problem is long and rich, going as far back as Borůvka's work in 1926 [1, 9, 13]. In fact, MST is perhaps the oldest open problem in computer science. According to Nešetřil [13], "this is a cornerstone problem of combinatorial optimization and in a sense its cradle." Textbook algorithms run in $O(m \log n)$ time, where n

[Chazelle 1999]: Minimum Spanning Tree in $O(m \cdot \alpha(m)$ time

Optimal MST Algorithm



An Optimal Minimum Spanning Tree Algorithm

SETH PETTIE AND VIJAYA RAMACHANDRAN

The University of Texas at Austin, Austin, Texas

Abstract. We establish that the algorithmic complexity of the minimum spanning tree problem is equal to its decision-tree complexity. Specifically, we present a deterministic algorithm to find a minimum spanning tree of a graph with n vertices and m edges that runs in time $O(\mathcal{T}^*(m, n))$ where $\mathcal{T}^*$ is the minimum number of edge-weight comparisons needed to determine the solution. The algorithm is quite simple and can be implemented on a pointer machine.

Although our time bound is optimal, the exact function describing it is not known at present. The current best bounds known for $\mathcal{T}^*$ are $\mathcal{T}^*(m, n) = \Omega(m)$ and $\mathcal{T}^*(m, n) = O(m \cdot \alpha(m, n))$, where α is a certain natural inverse of Ackermann's function.

Even under the assumption that $\mathcal{T}^*$ is superlinear, we show that if the input graph is selected from $G_{n,m}$, our algorithm runs in linear time with high probability, regardless of n, m, or the permutation of edge weights. The analysis uses a new martingale for $G_{n,m}$ similar to the edge-exposure martingale for $G_{n,p}$.

- [Pettie Ramachandran 2002]: An optimal algorithm for minimum spanning tree (as fast as any other algorithm)
- Must be $O(m \cdot \alpha(m))$ because it's as least as fast as Chazelle's algorithm. But could be better!

Review!