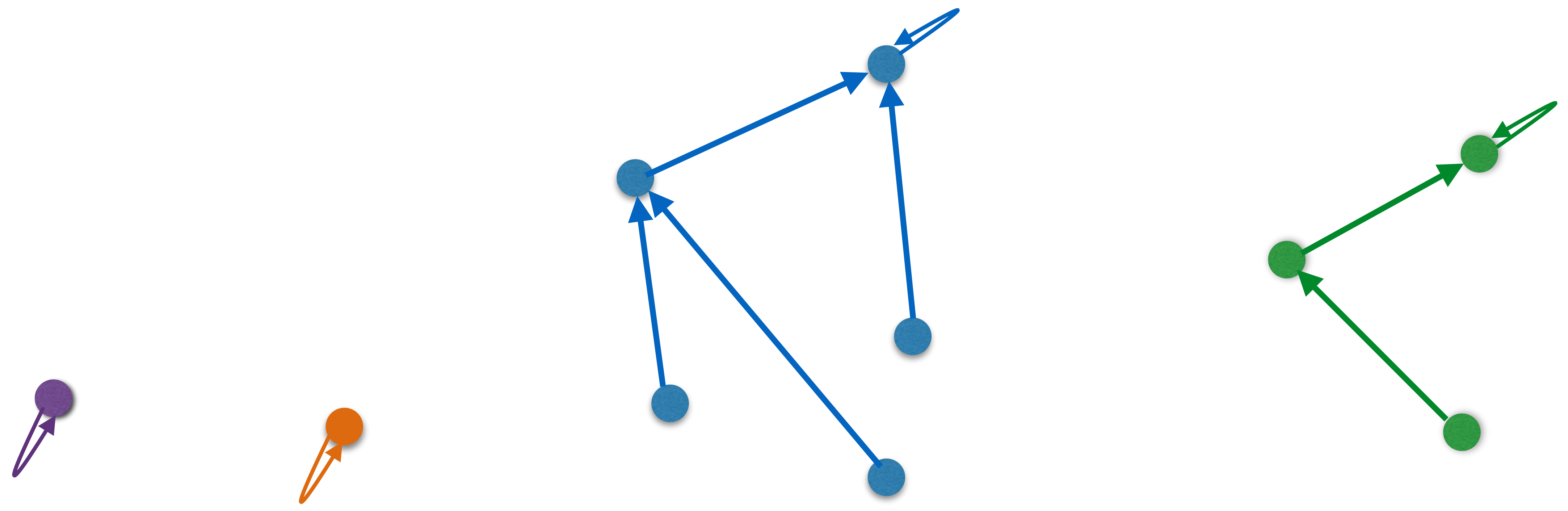
Union-Find, MST, and Shortest Path

Admin

- Assignment 3 out right after class
- Assignment 1 extra credit not included in current grades (will update later today)
- Schedule updated later today assuming Mountain Day is next week
- Previous assignment solutions/questions?
    - Come to me or a TA during office hours!

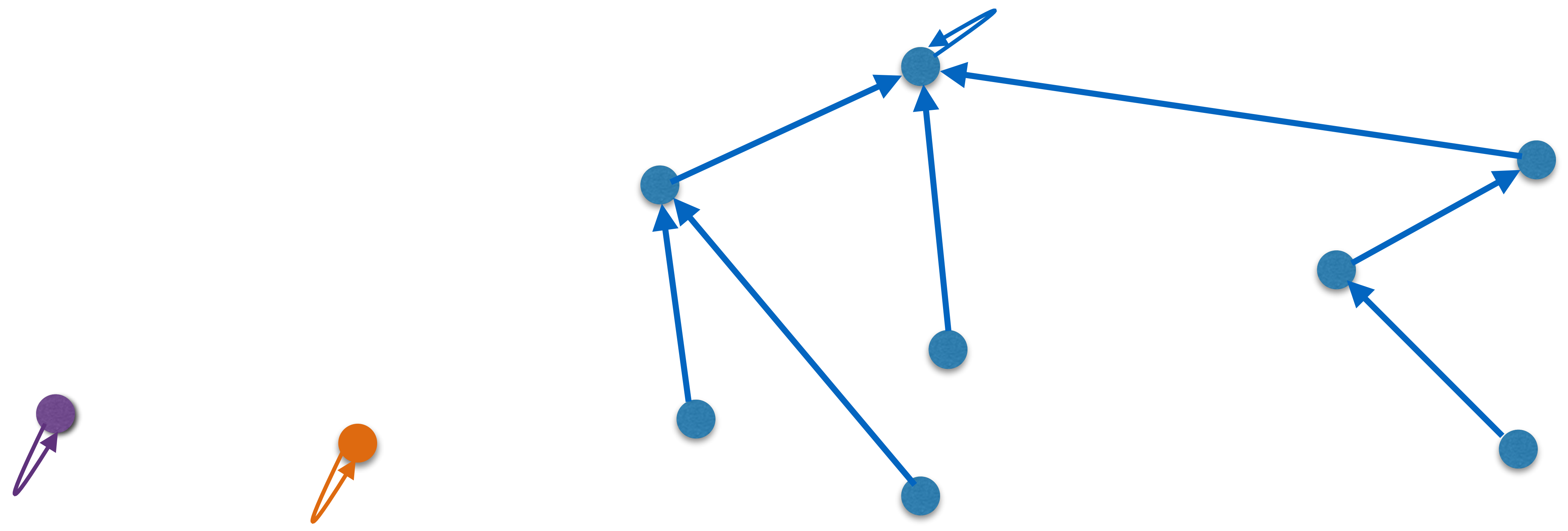
Fast Union with "Trees"

- Let's keep a head node as before
- Now, let's have our pointers act like a tree, but pointing up
- How can we Union?



Fast Union with “Trees”

- Let’s keep a head node as before
- Now, let’s have our pointers act like a tree, but pointing up
- How can we Union? Point lower-height tree to higher



What do we get?

- “Up tree” method:
  - $O(1)$ Union, $O(\log n)$ Find
- “Point to head” method:
  - $O(\log n)$ amortized Union, $O(1)$ Find

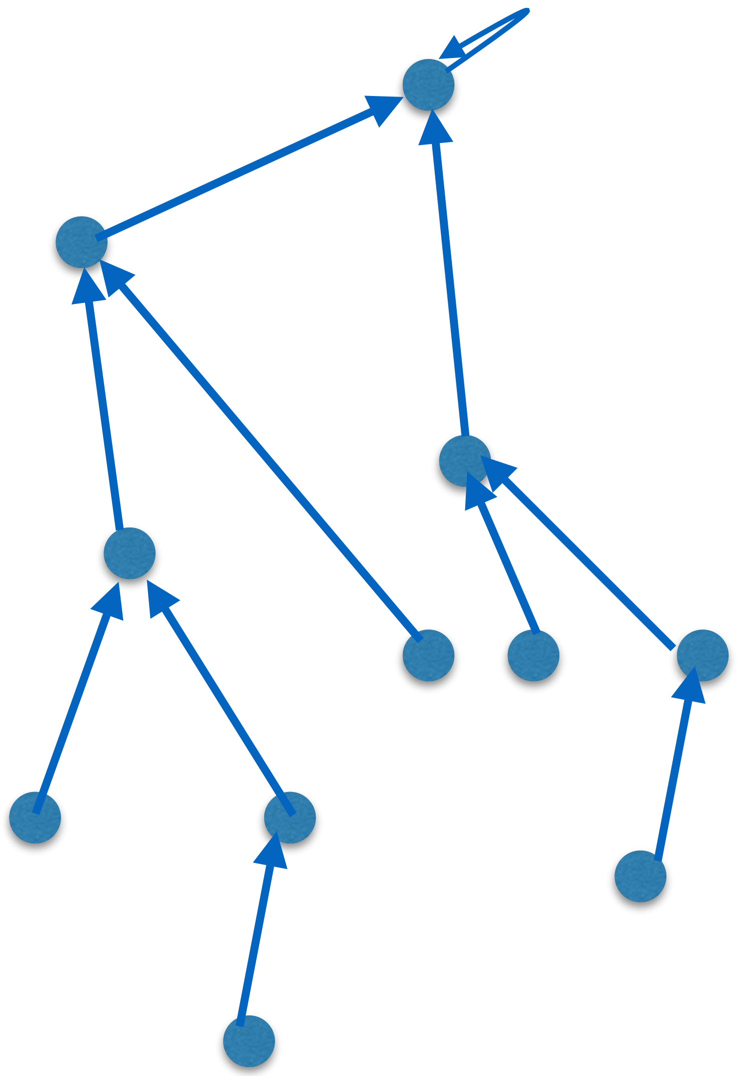
Class poll!

Do you think we can do better? Which of the following do you think is the case?

- Either Union or Find take $\Omega(\log n)$
- If you multiply Union and Find, the product of their times must be $\Omega(\log n)$
- Both can be $O(1)$
- Something in the middle

Let’s make things work a little faster in practice

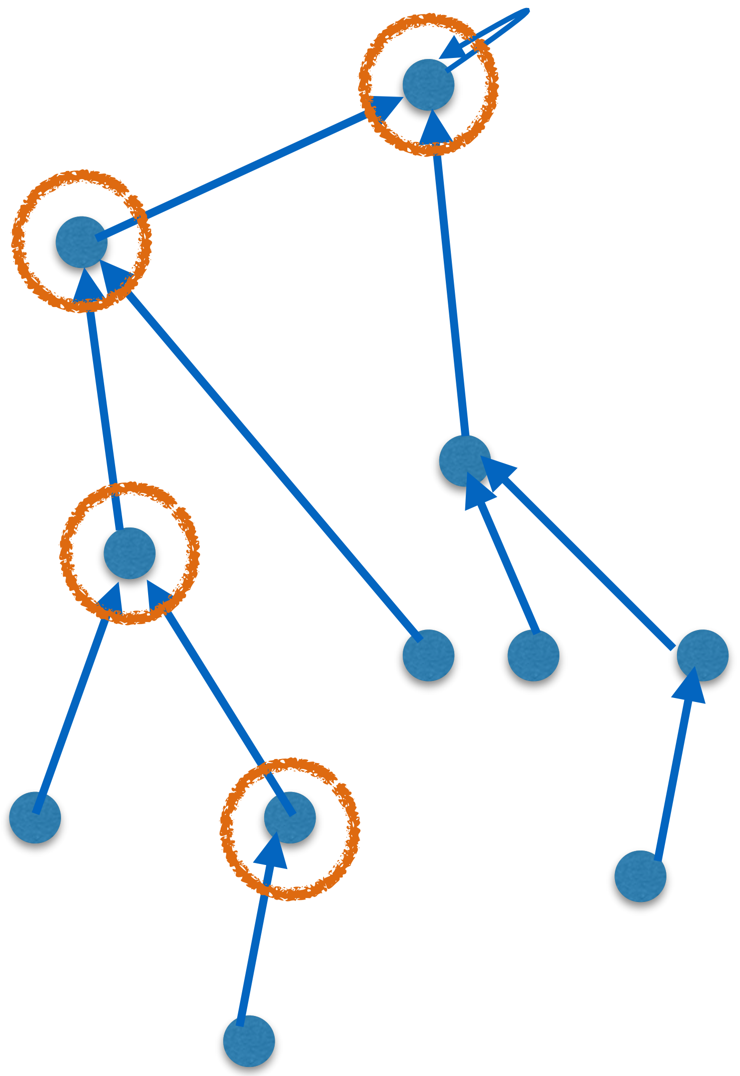
- Think about the “up trees”
- When we’re doing a Find, is there work we can do to make future finds faster?



Let’s make things work a little faster in practice

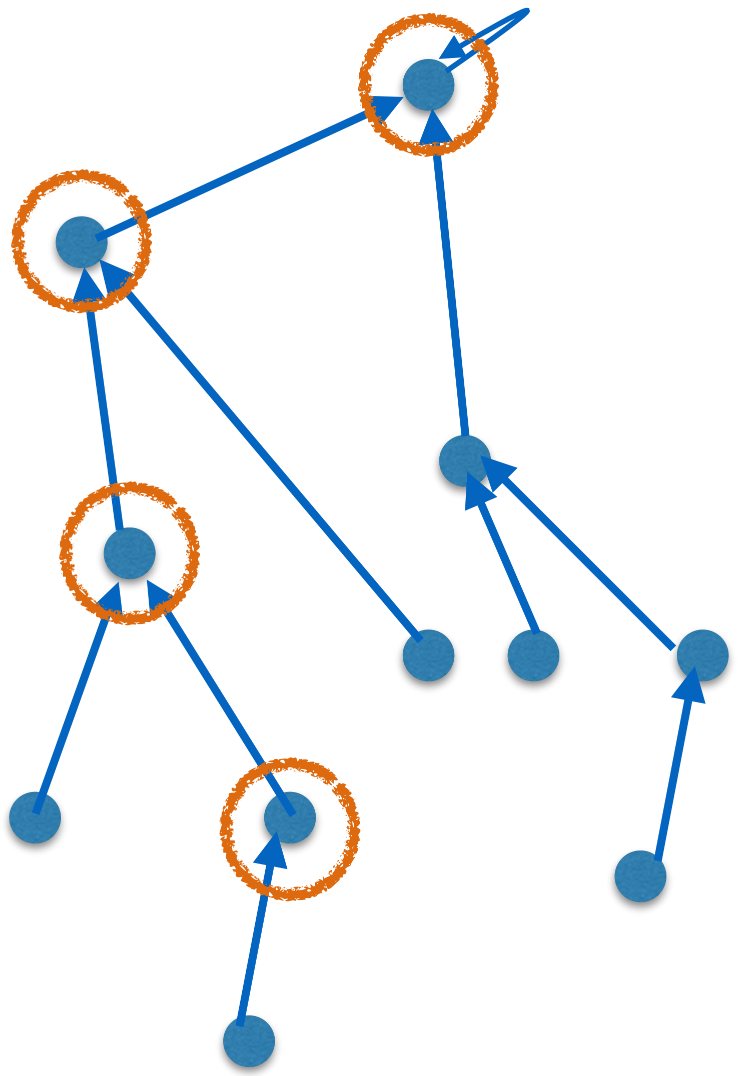
- Think about the “up trees”

- When we’re doing a Find, is there work we can do to make future finds faster?



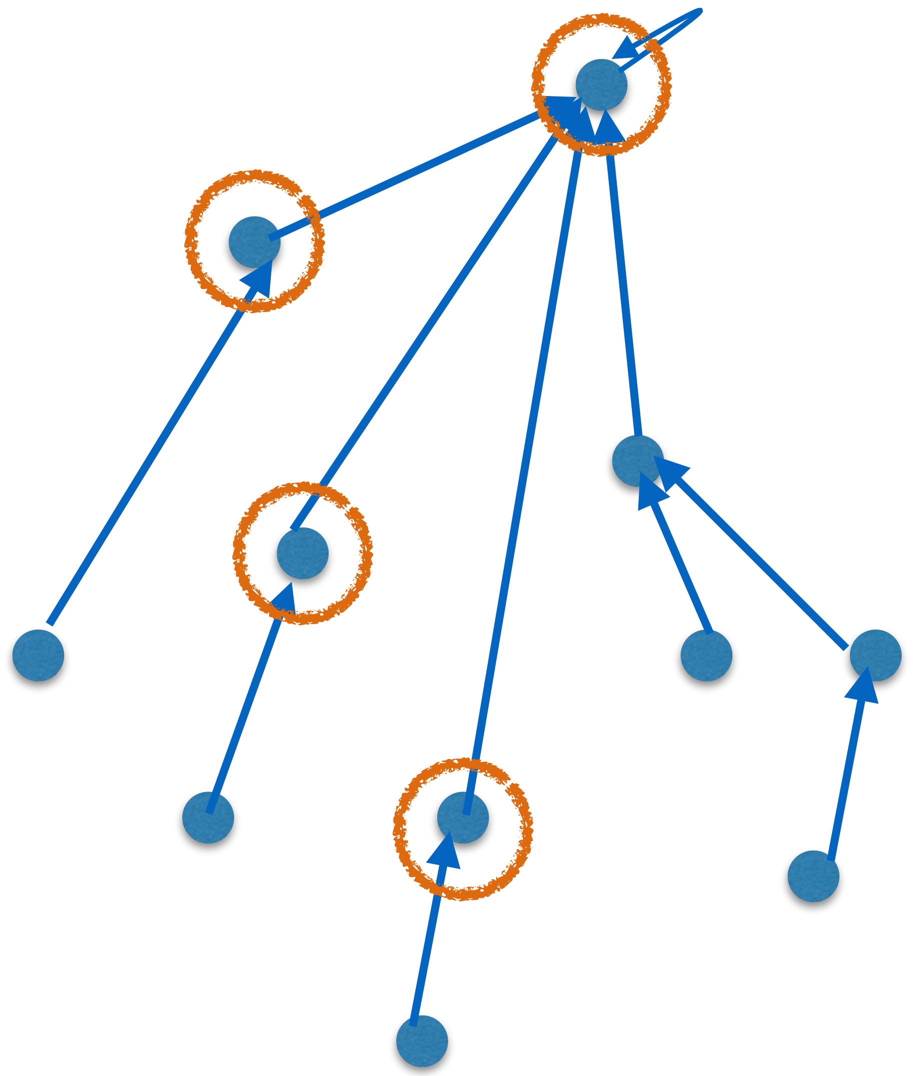
Let's make things work a little faster in practice

- When we're doing a Find, is there work we can do to make future finds faster?
- We really want all of these to point right to the head
- So…let's do that!



Let's make things work a little faster in practice

- When we're doing a Find, is there work we can do to make future finds faster?
- We really want all of these to point right to the head
- So…let's do that!
- Wait, I've broken the data structure!
    - I can't maintain "height"



Maintaining “Height”

- We can’t maintain the exact height. What if we pretend we can? Just do the same bookkeeping:
- Keep a “rank” (starting at 0)
- Always point the head of smaller rank to the head of larger rank; keep rank the same
- If both ranks are the same, point one to the other, and increment the rank

What do we get?

- Every time I have an expensive Find, I get a lot of great work done for the future by shrinking the tree
  - Called “path compression” [Galler, Fischer ’64]
- Now I have an inaccurate “rank” instead of an actual “height”
- First: did this make things worse? Union is still $O(1)$, is Find $O(\log n)$?
  - We did not make things worse, Find is $O(\log n)$
- Can we show that we made things better?

Surprising Result: Hopcroft Ulman'73

- Amortized complexity of union find with path compression improves significantly!
- Time complexity for n union and find operations on n elements is $O(n \log^* n)$
- $\log^* n$ is the number of times you need to apply the log function before you get to a number <= 1
- **Very small! Less than 5 for all reasonable values**

$$\boxed{\log^*(n) = \begin{cases} 0 & \text{if } n \le 1 \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}}$$

n	1	2	$4 = 2^2$	$16 = 2^4$	$65,536 = 2^{16}$	$2^{65,536}$
$\log^*(n)$	0	1	2	3	4	5

Digging Deeper

Surprising Result: Tarjan '75

- Improved bound on amortized complexity of union-find with path compression
- Time complexity for n union and find operations on n elements is $O(n\alpha(n))$, where
  - $\alpha(n)$ is extremely slow-growing, **inverse-Ackermann function**
  - Essentially a constant
- Grows much **muuchch morrree** slowly than $\log^*$
- $\alpha(n) \leq 4$ for all values in practice
- **Result.** Union and Find become (essentially) amortized constant time in practice (just short of $O(1)$ in theory) !

Digging Deeper

Inverse Ackermann

- **Inverse Ackerman:** The function $\alpha(n)$ grows much more slowly than $\log^{*^c} n$ for **any fixed c**

- With $\log*$, you count how many times does applying $\log$ over and over gets the result to become small
- With the inverse Ackermann, essentially you count how many times you iterate $\log*$ (not log!) over and over to get the result to become small

- $\alpha(n) = \min\{k \mid \log^{\overbrace{****\cdots*}^{k}}(n) \leq 2\}$

- $\alpha(n) = 4$ for $n = 2^{2^{2^{2^{16}}}}$

Digging Deeper

Can we do better?

- OK, so that's "basically constant". Can we get constant?
- No. *Any data structure* for union find requires $\Omega(\alpha(n))$ amortized time (Fredman, Saks '89)
- So up trees with path compression are optimal(!)

Many Applications of Union-Find

- Good for applications in need of clustering
    - cities connected by roads
    - cities belonging to the same country
    - connected components of a graph
- Maintaining equivalence classes
- Maze creation!

Digging Deeper

Back to MST

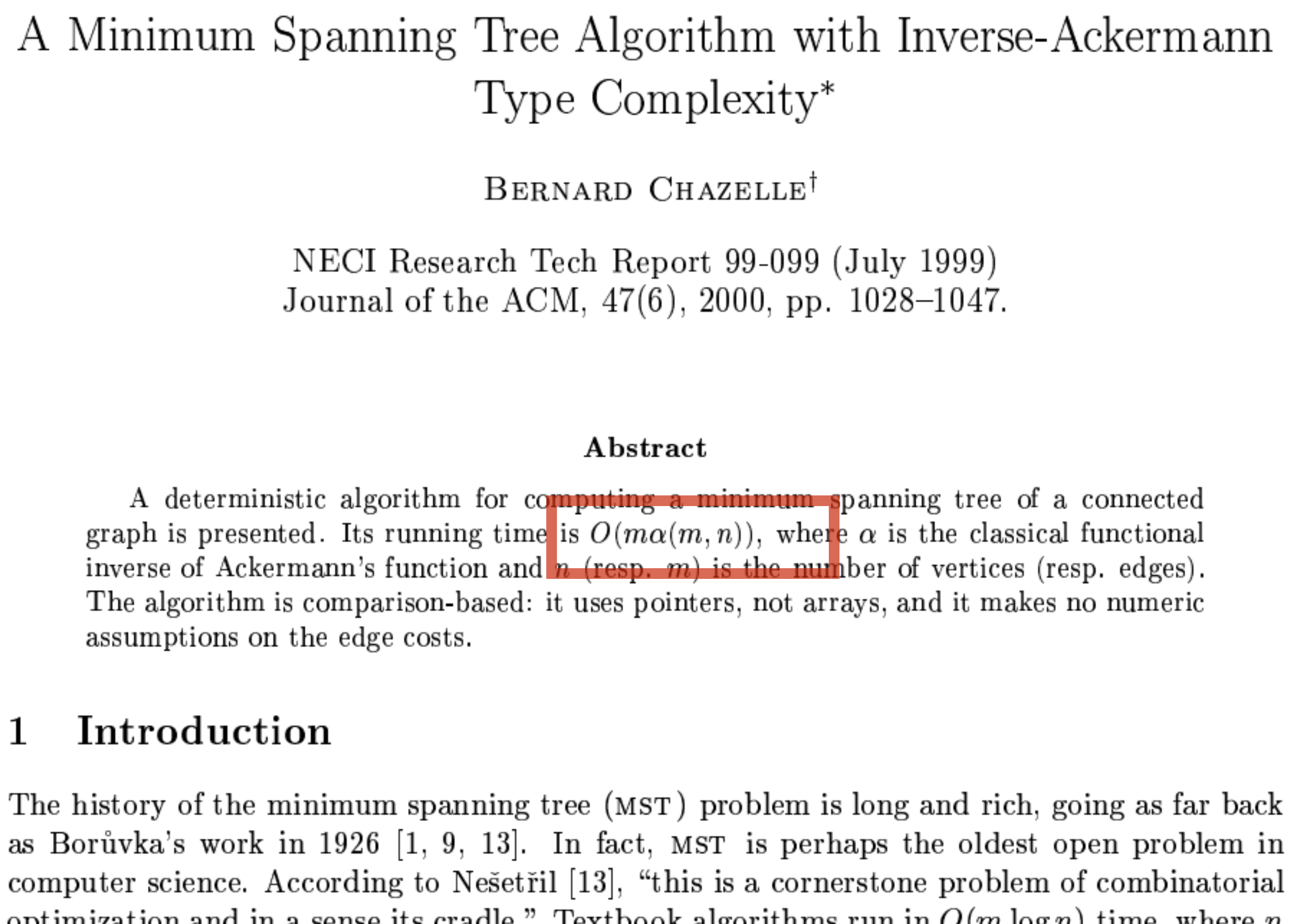
- Prim's algorithm is $O(m + n \log n)$ if using a Fibonnacci tree
- Kruskal's algorithm is $O(m \log m)$
- Which is better in practice?
- Is sorting time required?

MST Algorithms History

- **Borůvka's Algorithm** (1926)
  - The Borvka / Choquet / Florek-ukaziewicz-Perkal-Steinhaus-Zubrzycki / Prim / Sollin / Brosh algorithm
  - Oldest, most-ignored MST algorithm, but actually very good
- **Jarník's Algorithm** ("Prims Algorithm", 1929)
  - Published by Jarník, independently discovered by Kruskal in 1956, by Prims in 1957
- **Kruskal's Algorithm** (1956)
  - Kruskal designed this because he found Borůvka's algorithm "unnecessarily complicated"

Can we do better?

Best known algorithm by Chazelle (1999)



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

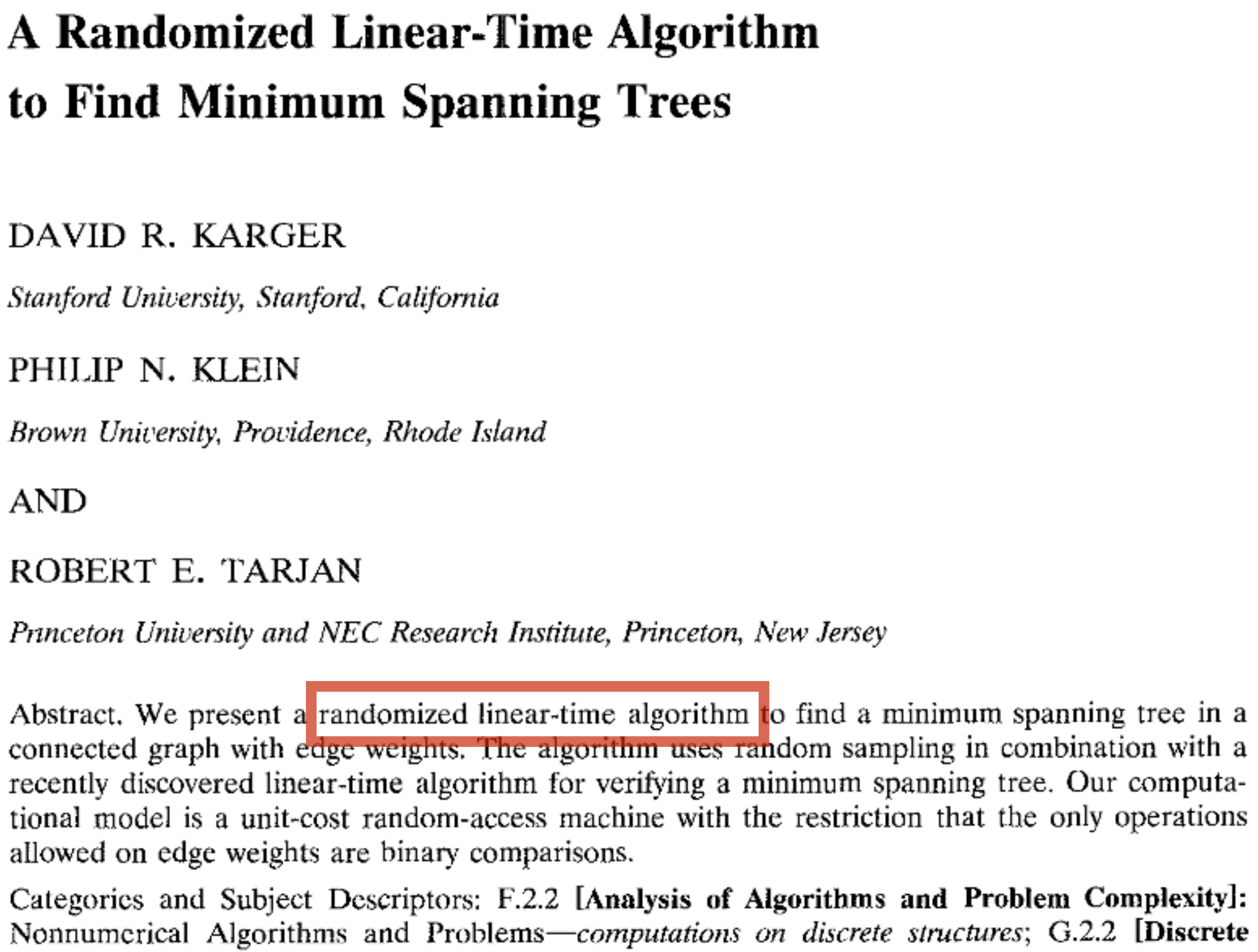
A deterministic algorithm for computing a minimum spanning tree of a connected graph is presented. Its running time is $O(m\alpha(m,n))$, where α is the classical functional inverse of Ackermann's function and n (resp. m) is the number of vertices (resp. edges). The algorithm is comparison-based: it 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

Can we do better?

Using randomness, can get $O(n+m)$ time!



A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees

DAVID R. KARGER

Stanford University, Stanford, California

PHILIP N. KLEIN

Brown University, Providence, Rhode Island

AND

ROBERT E. TARJAN

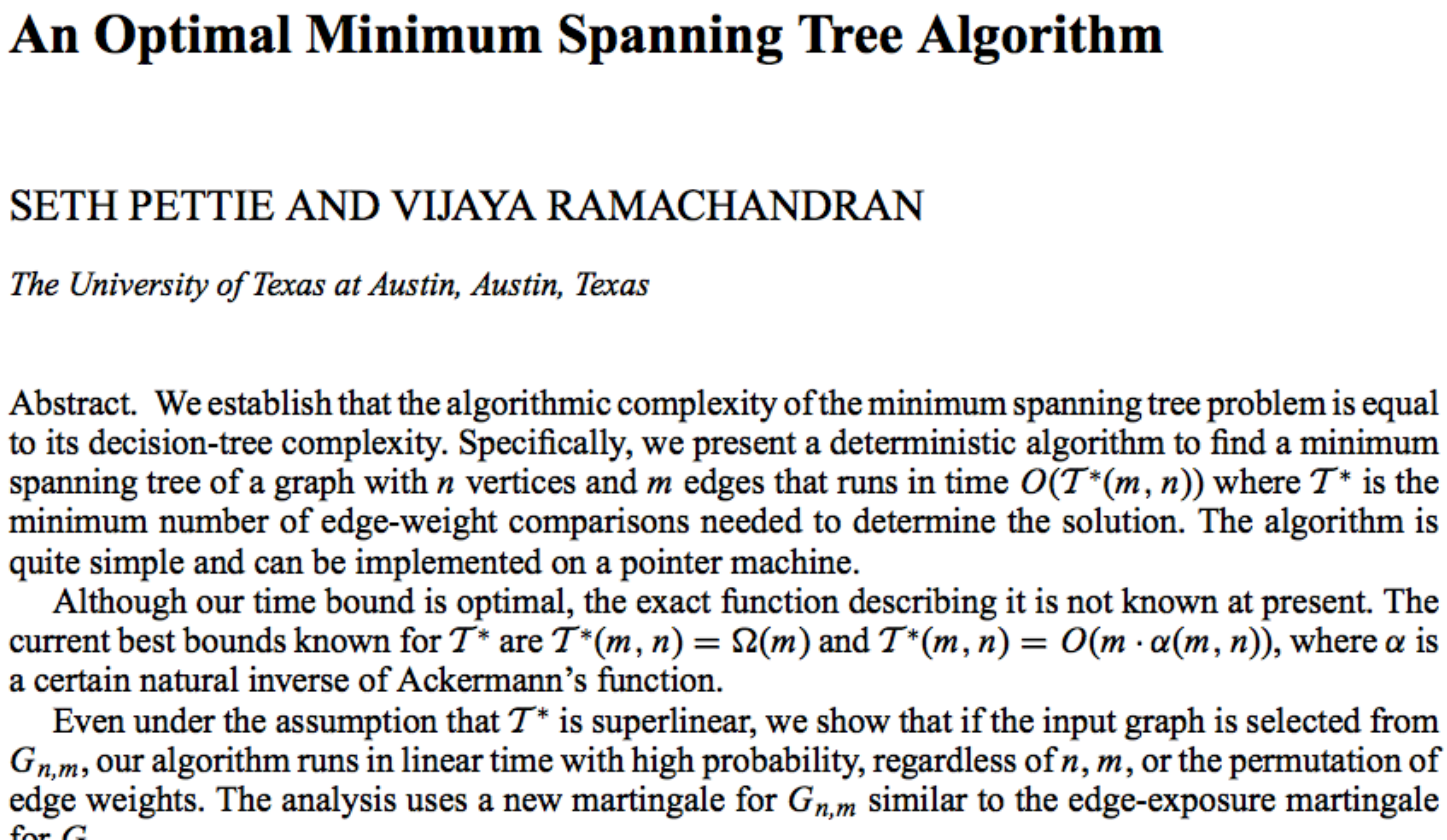
Princeton University and NEC Research Institute, Princeton, New Jersey

Abstract. We present a randomized linear-time algorithm to find a minimum spanning tree in a connected graph with edge weights. The algorithm uses random sampling in combination with a recently discovered linear-time algorithm for verifying a minimum spanning tree. Our computational model is a unit-cost random-access machine with the restriction that the only operations allowed on edge weights are binary comparisons.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G.2.2 [**Discrete**

Optimal MST Algorithm?

Has been discovered but don’t know its running time!



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.

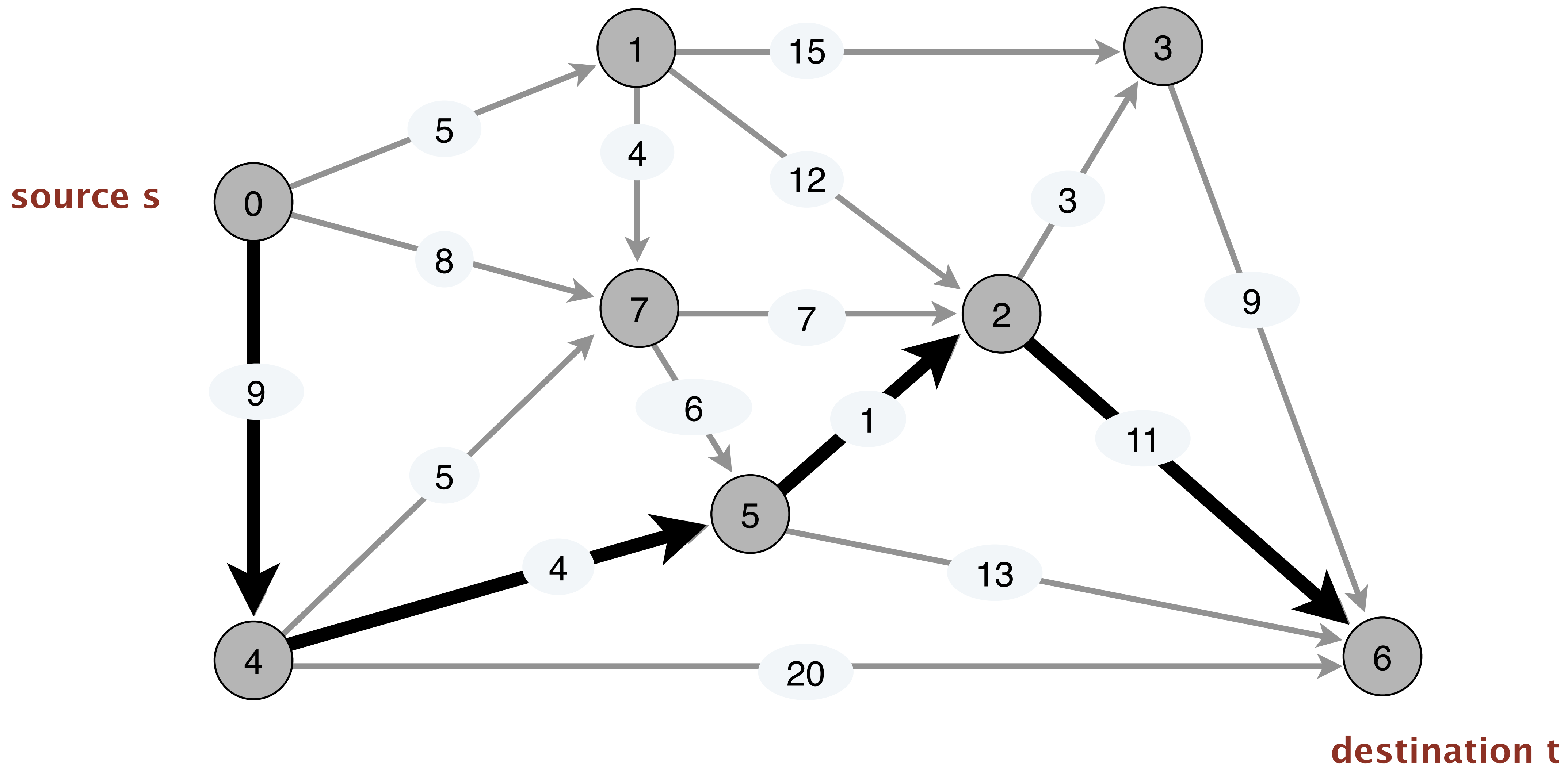
Although our time bound is optimal, the exact function describing it is not known at present. The current best bounds known for $\mathcal{T}^*$ are $\mathcal{T}^*(m, n) = \Omega(m)$ and $\mathcal{T}^*(m, n) = O(m \cdot \alpha(m, n))$, where α is a certain natural inverse of Ackermann’s function.

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

Story So Far

- Graph Traversal algorithms
  - BFS, DFS
  - Properties and applications of traversals
    - Bipartite matching, topological ordering
    - Approximating diameter or graphs
    - Finding bridges, articulation points
- Greedy algorithms
  - Greedy stays ahead and exchange argument proofs
  - Minimum spanning trees
- Last lecture on Greedy:
  - Shortest paths in weighted **directed** graphs

Shortest Paths in Weighted Graph



length of path = 9 + 4 + 1 + 11 = 25

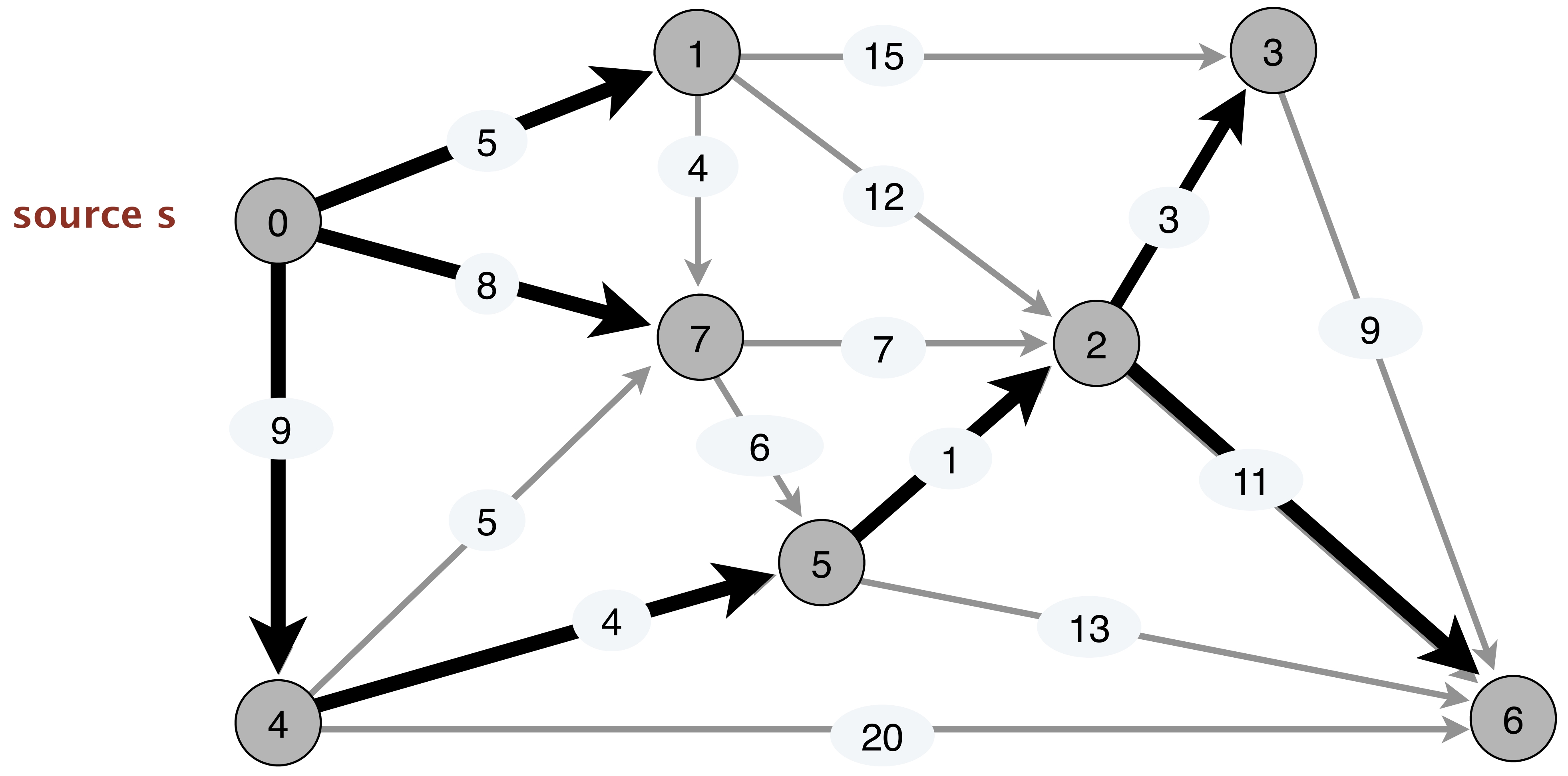
Shortest Paths in Weighted Graph

Problem. Given a directed graph $G = (V, E)$ with positive edge weights: that is, each edge $e \in E$ has a positive weight $w(e)$ and vertices s and t, find the shortest path from s to t.

The shortest path from s to t in a weighted graph is a path P from s to t (or a s-t path) with minimum weight $w(P) = \sum_{e \in P} w(e)$.

Single-Source Shortest Path

Assumption. There exists a path from s to every node in the graph.



shortest-paths tree

Single-Source Shortest Path

Problem. Given a directed graph $G = (V, E)$ with positive edge weights w_e for each $e \in E$ and a source $s \in V$, find a shortest directed path from s to every other vertex in V.

Quick quiz. Which of these changes to edge weights on a graph does not affect the shortest paths?

A. Adding 17

B. Multiplying 17

C. None of the above

Shortest Paths Applications

- Map routing
- Robot navigation
- Texture mapping
- Typesetting in LaTeX.
- Urban traffic planning.
- Scheduling, routing of operators
- Network routing protocols (OSPF, BGP, RIP)
- It is so important that we will revisit shortest paths when we study dynamic programming!

Dijkstra’s Algorithm

Computes the shortest path from s to all vertices

Dijkstra’s algorithm has the following key components

- It evolves a tree, rooted at s, of shortest paths to the vertices closest to s
- It keeps a conservative estimate (that is, over-estimate) $d(u)$ of the shortest path length to vertices u not yet in the tree
- It selects the next vertex to add to the tree based on lowest estimate **(Greedy: choose locally best next move)**

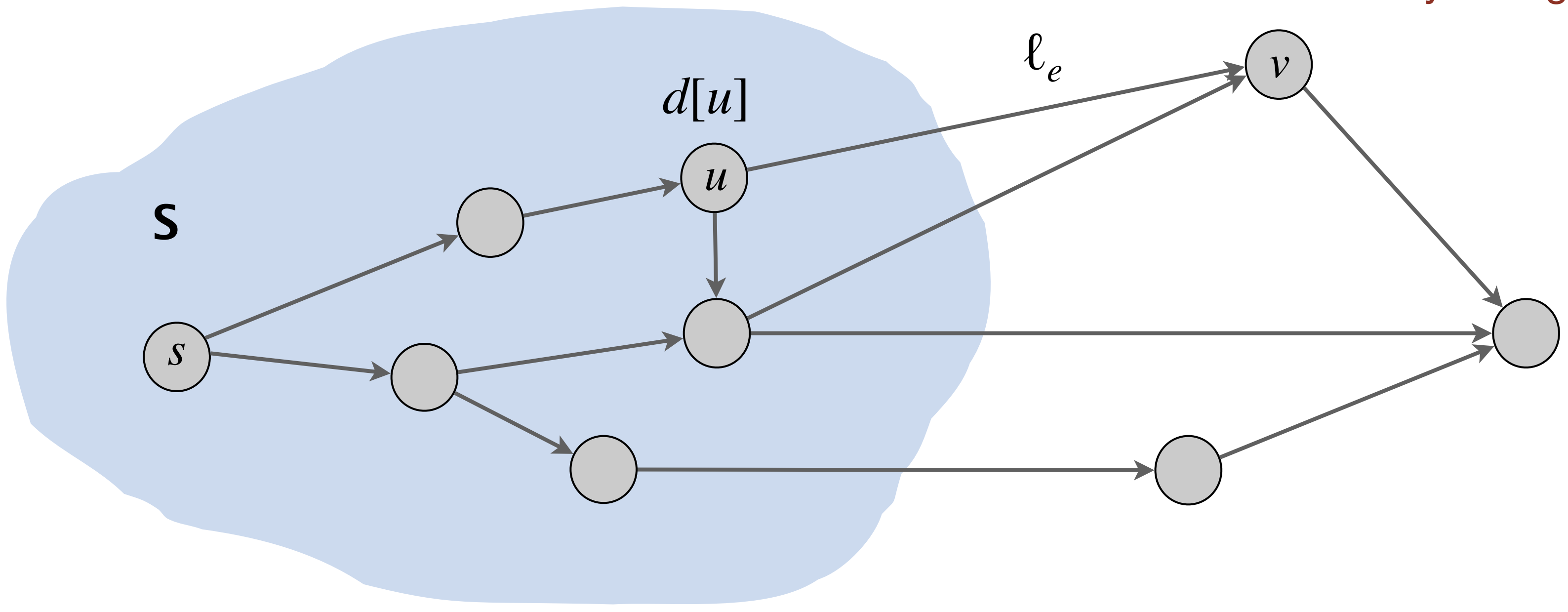
Dijkstra's Algorithm

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \leadsto u$ path.

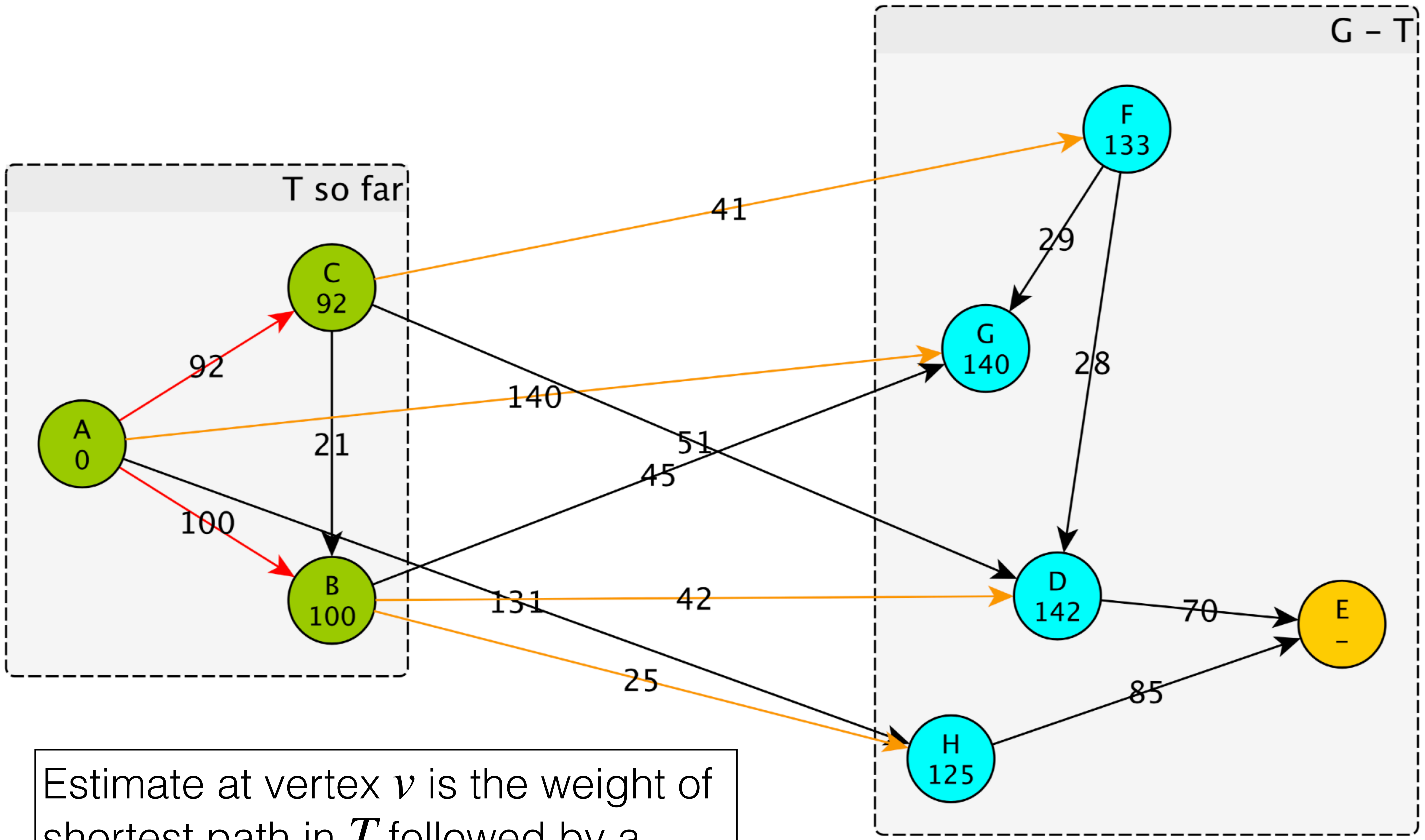
- Initialize $S \leftarrow \{s\}$, $d[s] \leftarrow 0$.
- Repeatedly add unexplored node $v \notin S$ which minimizes

$$\min_{e=(u,v):u \in S} \boxed{d[u] + \ell_e}$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$



Dijkstra's Algorithm



Estimate at vertex v is the weight of shortest path in T followed by a single edge from T to $G - T$

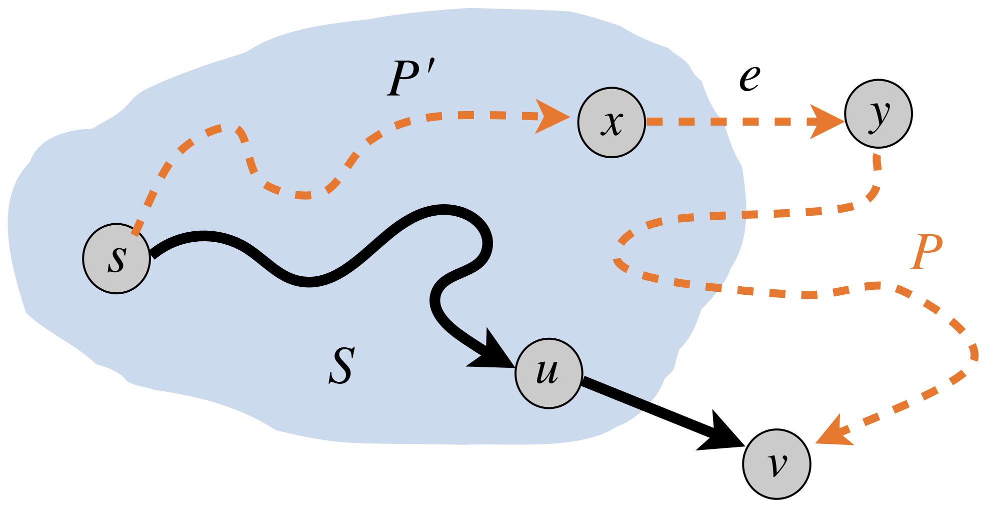
Dijkstra’s Demo

Pseudocode in Textbook

- Use it to test your understanding
- But, high-level idea is most important

Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

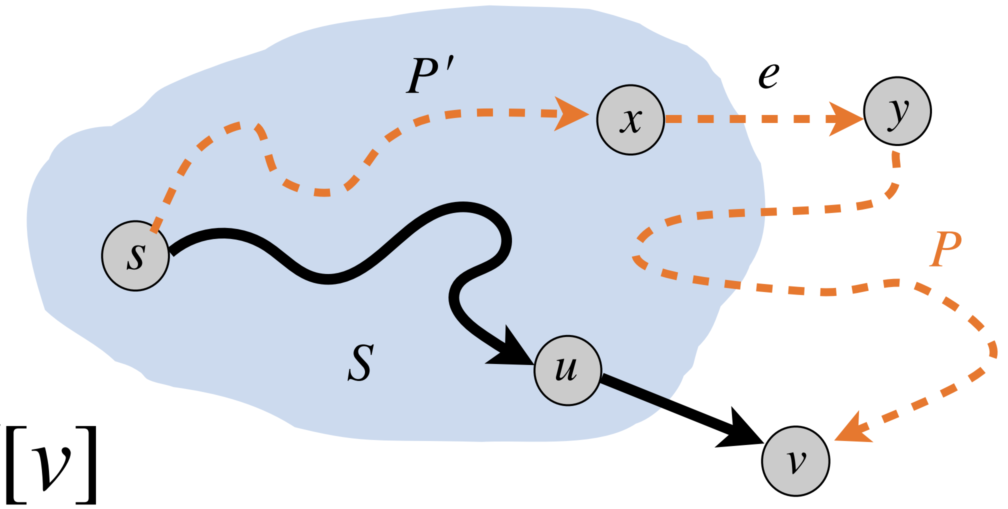


Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$. Let v be next node added to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



Non-negative weights

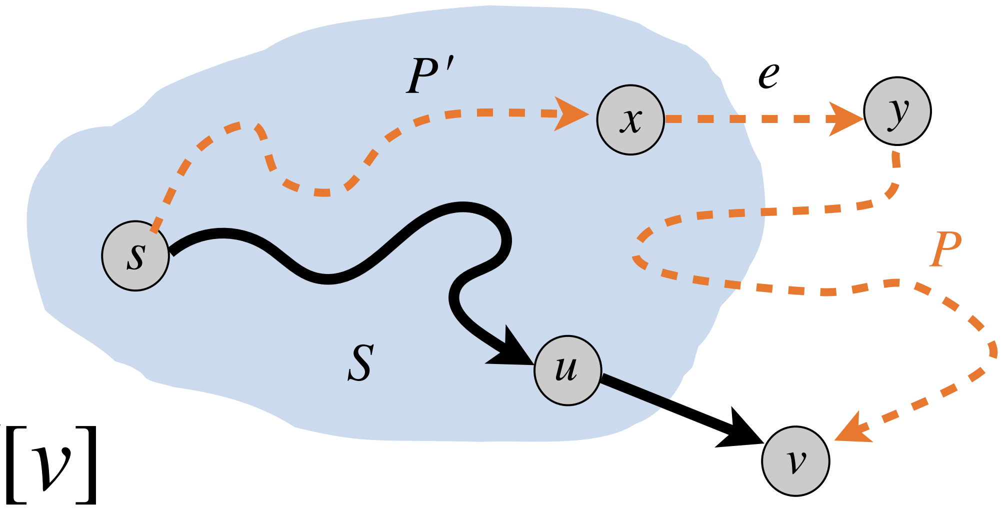
$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v]$$

Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$. Let v be next node added to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



Inductive Hypothesis

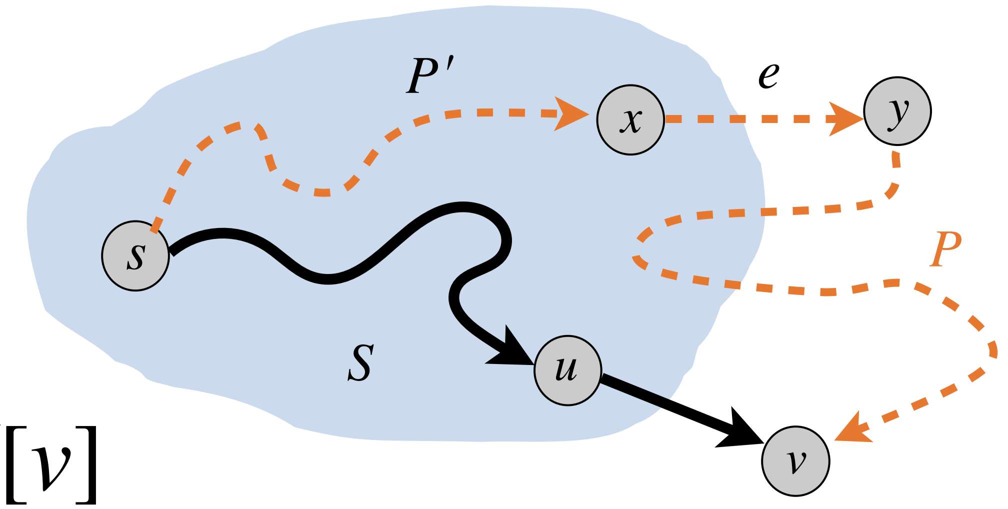
$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v]$$

Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$. Let v be next node added to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



When x was added to S, $d[y]$ was updated

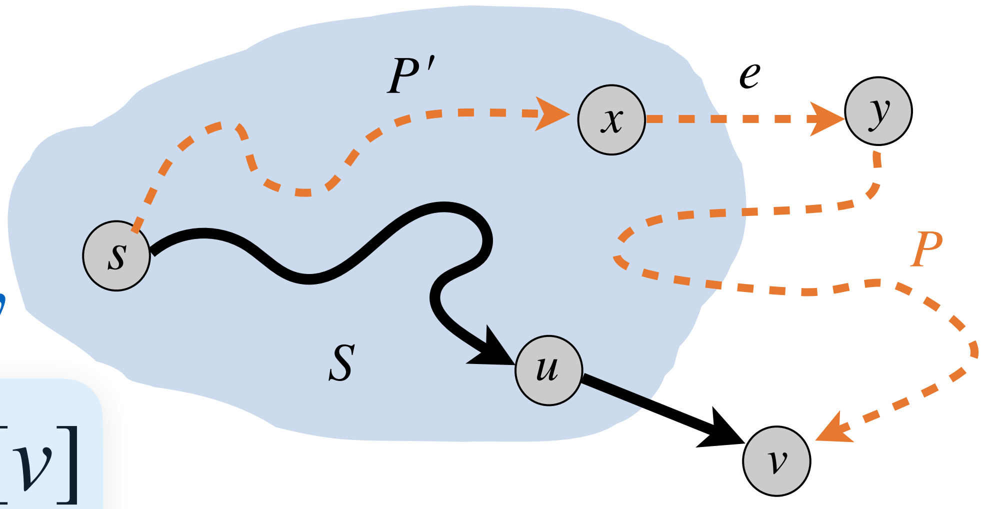
$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v]$$

Dijkstra’s Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$. Let v be next node added to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



Dijkstra chose to add v instead of y

$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v]$$

Implementation & Running Time

How can we efficiently implement Dijkstra's algorithm? We need to be able to

- Visit every neighbor of a vertex
- Maintain set of visited S and unvisited vertices $V - S$
- Maintain a tree of edges $(v, (\text{pred}[v])$
- (Delete-min) Select & delete unvisited vertex v with min d[v]
- (Decrease-key) Update $d[v]$ for unvisited vertices

Priority Queue: Delete-Min & Decrease-Key

Updating the Priority Queue

How to to update priorities (perform decrease-key) in the priority queue efficiently?

- Recall vertices are represented by $1, \ldots, n$
- Maintain an array `PQIndex[1..n]` that holds the index of each vertex v in the priority queue
- **(Decrease-min)** If we update $d[u]$ for some u, we then heapify-up from u's location in the PQ to restore heap property
- Every time we swap two heap elements, we update `PQIndex` for the two vertices

Time and Space Analysis

Space: $O(n+m)$; Running Time:

- Traversal of S (each edge visited at most once)
  - $O(n \log n + m)$
  - Why the $O(\log n)$?
  - n deleteMin operations from PQ to select next vertex $O(n \log n)$
- Construction of T: time proportional to its size:$O(n)$
- Creation of priority queue: $O(n)$
- At most one decrease-key for each edge: $O(m \log n)$

Total time: $O((n+m)\log n)$. This is $O(m \log n)$ if G is connected (Why?)

What About Undirected Graphs

How to solve the single-source shortest paths problem in undirected graphs with positive edge lengths?

(a) Replace each undirected edge with two antiparallel edges of same length and run Dijkstra's algorithm on the resulting digraph

(b) Modify Dijkstra's algorithms so that when it processes node u, it consider all edges incident to u (instead of edges leaving u)

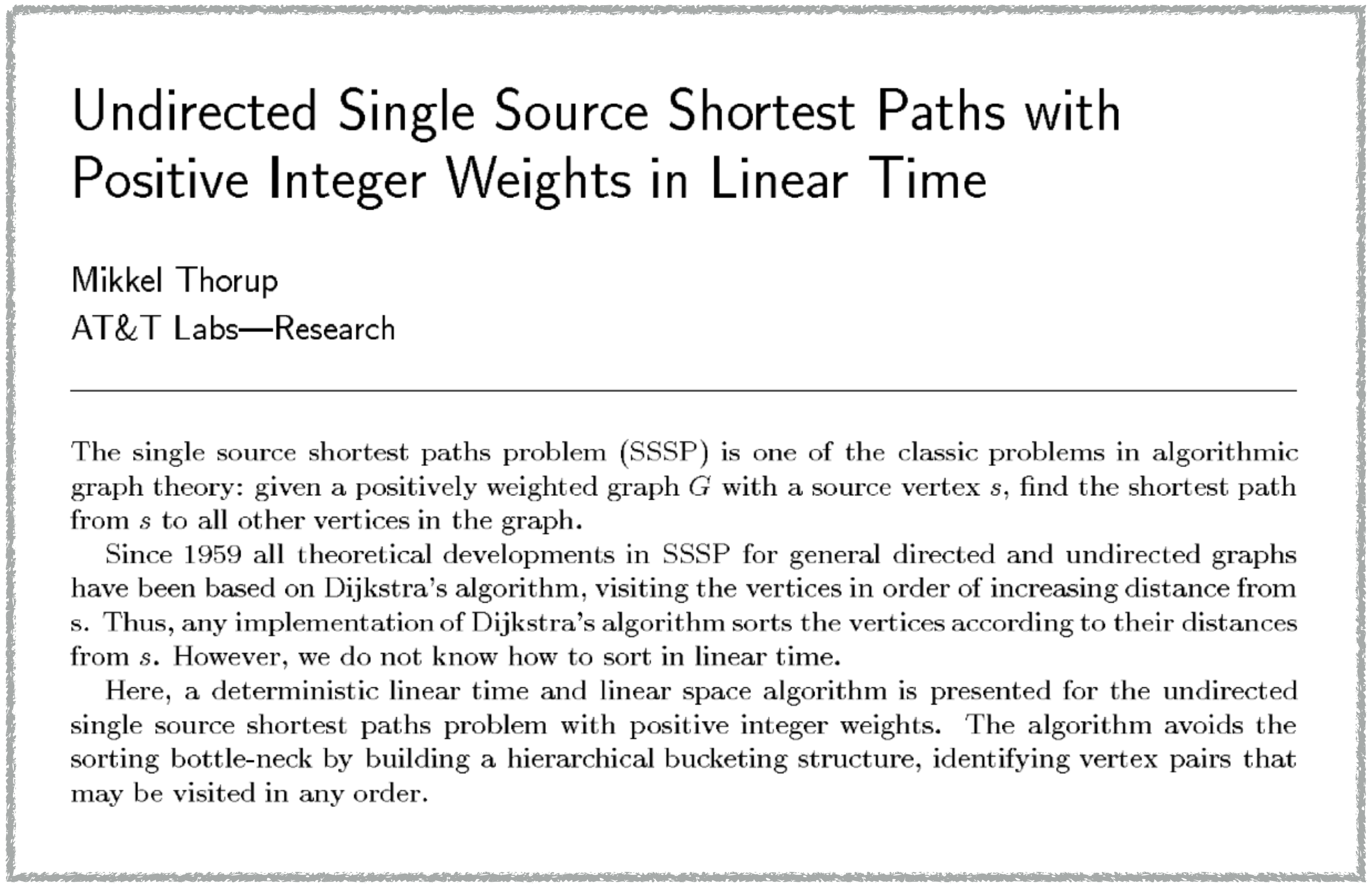
(c) Either A or B

(d) Neither A nor B

Shortest Path in Linear Time

[Thorup 1999] Can solve single-source shortest paths problem in undirected graphs with positive **integer** edge lengths in $O(m)$ time.

Remark. Does not explore vertices in increasing distance from s



Undirected Single Source Shortest Paths with Positive Integer Weights in Linear Time

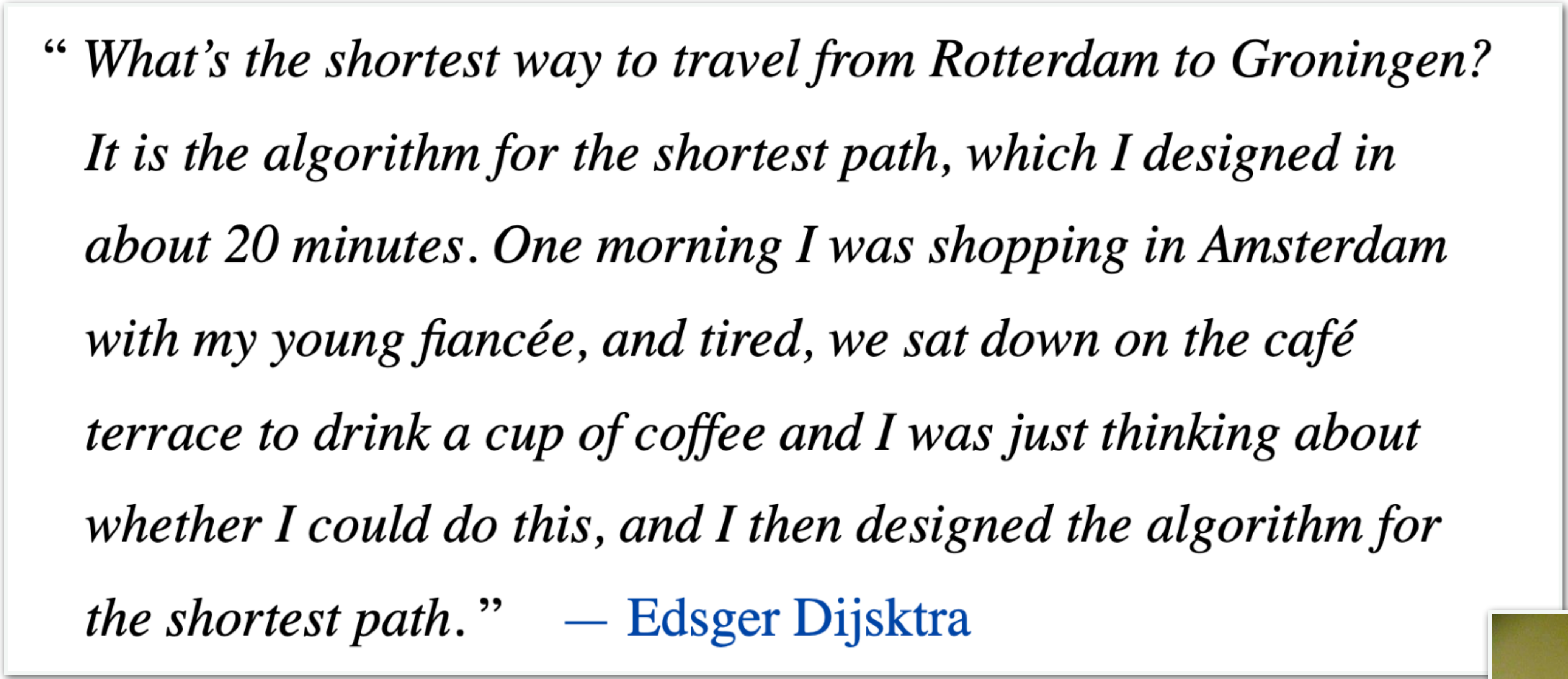
Mikkel Thorup
AT&T Labs—Research

The single source shortest paths problem (SSSP) is one of the classic problems in algorithmic graph theory: given a positively weighted graph G with a source vertex s, find the shortest path from s to all other vertices in the graph.

Since 1959 all theoretical developments in SSSP for general directed and undirected graphs have been based on Dijkstra's algorithm, visiting the vertices in order of increasing distance from s. Thus, any implementation of Dijkstra's algorithm sorts the vertices according to their distances from s. However, we do not know how to sort in linear time.

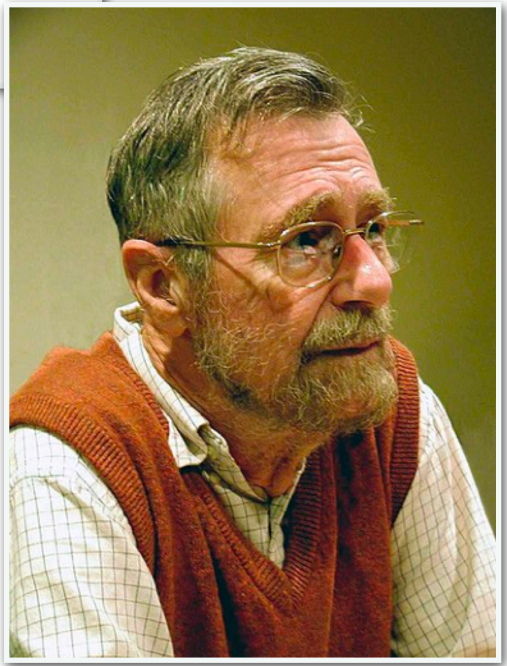
Here, a deterministic linear time and linear space algorithm is presented for the undirected single source shortest paths problem with positive integer weights. The algorithm avoids the sorting bottle-neck by building a hierarchical bucketing structure, identifying vertex pairs that may be visited in any order.

Edsger Dijkstra (1930-2002)



“ *What’s the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path.* ” — Edsger Dijsktra

- Shortest-path algorithm was actually discovered independently (around 1956) by a bunch of different people (read Jeff Erickson’s description and Strigler’s law in CS). *“Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-SeitzDantzig-Dijkstra-Minty-Whiting-Hillier algorithm”*



Recap: Greedy Algorithms

- Scheduling non-conflicting jobs (intervals)
    - Earliest finish-times first
    - Greedy stays ahead to prove correctness
- Scheduling with deadlines to maximize lateness of jobs
    - Earliest-deadline first
    - Exchange argument to prove correctness
- Minimum spanning trees: greedily pick edges
    - Cut property: essentially a non-local exchange argument
    - Boruvka's, Prims, Kruskals: correctness from cut property
    - Union find data structure
- Djisktra's shortest path: greedily find paths

Acknowledgments

- Some of the material in these slides are taken from
  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)