Greedy Algorithms

Announcements

- Assignment 1 out Mon (9/16), due Mon (9/23)
    - Question 2 hint explanation
    - “Reducing” to another problem
- Gradescope shows due date as including late day
- Assignment 0 solns will be shared through GLOW
- Wed office hours available for sign up on GLOW

Story So Far

- Review of basic graph algorithms
- Traversals and applications of traversals
- Today: new design paradigm
  - Greedy algorithms
  - Easy to design, hard to prove correctness
  - Greedy is not always optimal (hardly ever!)
  - A greedy algorithm is one that makes short-sighted, locally optimal decisions

Greedy: Examples

- We already saw a greedy algorithm that works!
  - Which one?
- Cashier's algorithm to return change in coins?
  - Greedy! To make change for \$$r$, start with biggest denomination less than r, and so on
  - Optimal for US coins!
  - (Not in general)

Greedy: Locally Optimal

Greedy algorithms build solutions by making locally optimal choices

Surprisingly, sometimes this also leads to globally optimal solutions!

We start with greedy algorithms greedy algorithms as the first design paradigm because

- They are natural and intuitive
- Proving they are actually is the hard part

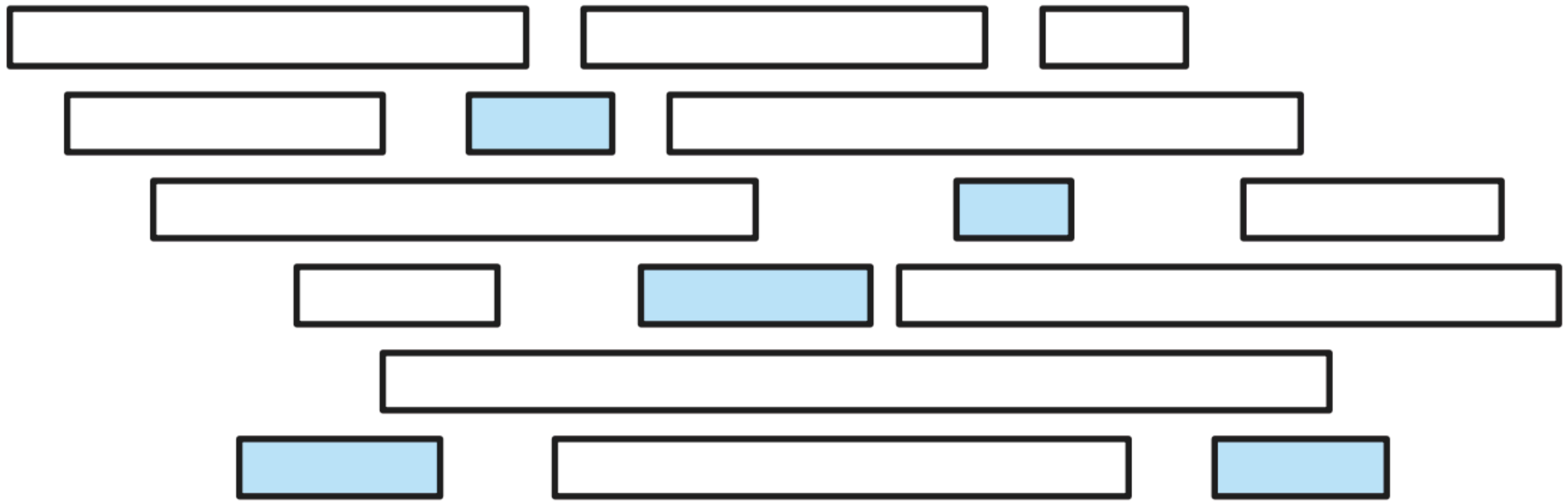
Greedy: Proof Techniques

Two fundamental approaches to proving correctness of greedy algorithms

- **Greedy stays ahead**: Partial greedy solution is, at all times, as good as an "equivalent" portion of any other solution
- **Exchange Property**: An optimal solution can be transformed into a greedy solution without sacrificing optimality.

Class Scheduling

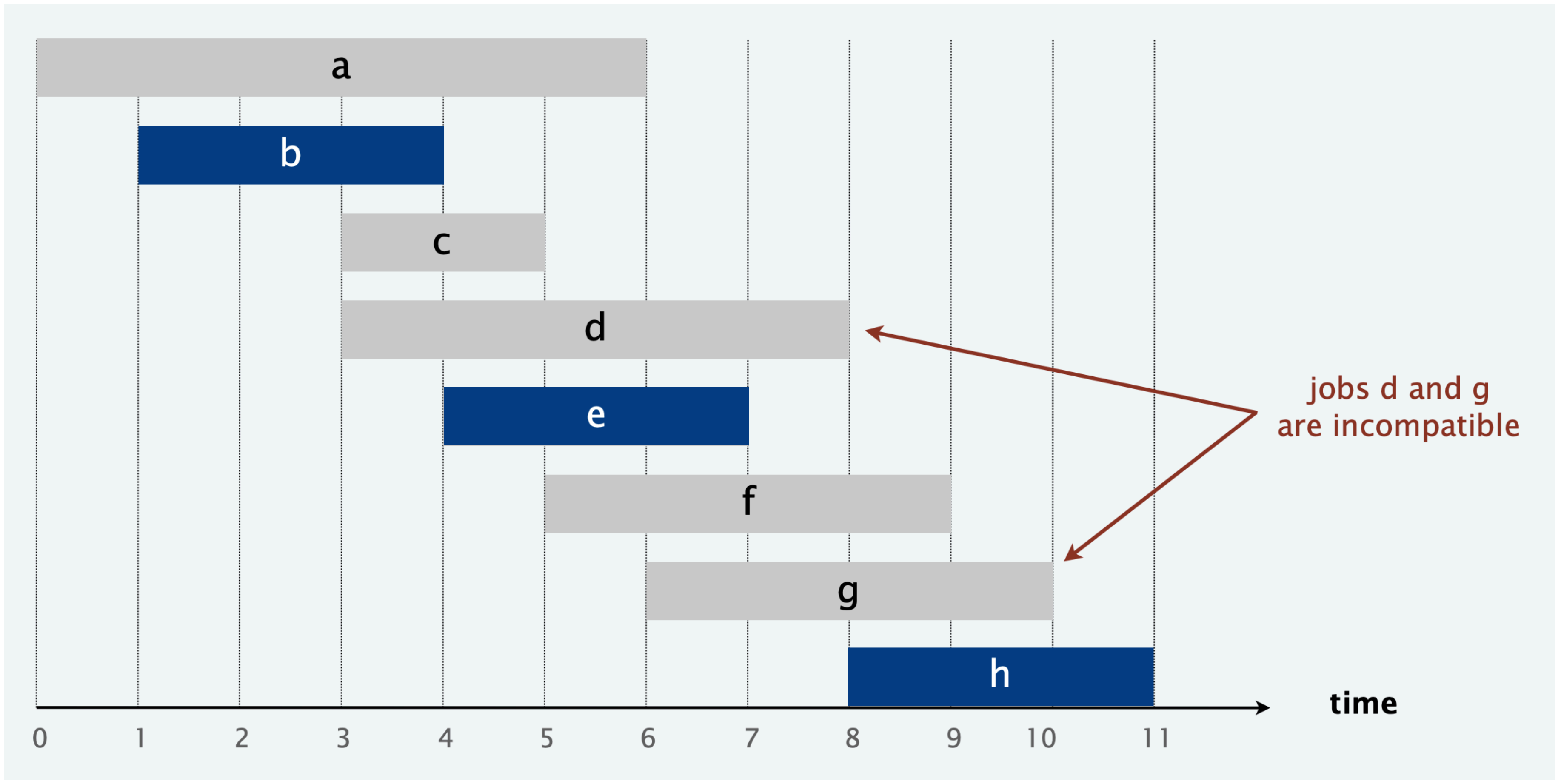
Problem. Given the list of start times $s_1, \ldots, s_n$ and finish times $f_1, \ldots, f_n$ of n classes (labeled $1, \ldots, n$), what is the maximum number of non-conflicting classes you can schedule?


A maximum conflict-free schedule for a set of classes.

From Erickson's Algorithms Book

Interval Scheduling

Job scheduling. This is a general job scheduling problem. Suppose you have a machine that can run one job at a time and n job requests with start and finish times: $s_1, \ldots, s_n$ and $f_1, \ldots, f_n$. How to determine the most number of compatible requests?



Many Ways to be Greedy

- Algorithmic idea: Pick a criterion to be greedy about. Keep choosing compatible jobs based on it
- Possible greedy criteria:
  - Schedule jobs with **earliest start time** first
  - Schedule jobs with **shortest interval** first
  - Schedule jobs with **earliest finish time** first

Many Ways to be Greedy

- Algorithmic idea: Pick a criterion to be greedy about. Keep choosing compatible jobs based on it
- Possible greedy criteria:
  - Schedule jobs with **earliest start time** first
    - Does't work. First jobs may take too long
  - Schedule jobs with **shortest interval** first
    - Doesn't work. First job may start too late
  - Schedule jobs with **earliest finish time** first
    - Surprisingly works. Idea: Free your resource as soon as possible!

Earliest-Finish-Time-First Algorithm

```
EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
$S \leftarrow \varnothing$.   ← set of jobs selected
FOR $j = 1$ TO $n$
    IF job $j$ is compatible with $S$
        $S \leftarrow S \cup \{ j \}$.
RETURN $S$.
```

Proving Greedy is Optimal

- Set S output consists of compatible requests
  - By construction!
- We want to prove our solution S is optimal
- Let $\mathcal{O}$ be an optimal set of jobs
- Want to show $|S| = |\mathcal{O}|$,i.e., greedy also selects the same number of jobs and thus is optimal
- Proof idea: greedy always "stays ahead" (or rather never falls behind OPT)

Get Ahead Stay Ahead Proof

Let $g_1, \ldots, g_k$ be the jobs (in order) selected by the greedy algorithm; let $o_1, \ldots, o_m$ be any other set of compatible jobs, ordered by increasing finish time.

Lemma. For all $i \le k$, we have: $f_{g_i} \le f_{o_i}$.

Proof.

Get Ahead Stay Ahead Proof

Let $g_1, \ldots, g_k$ be the jobs (in order) selected by the greedy algorithm; let $o_1, \ldots, o_m$ be any other set of compatible jobs, ordered by increasing finish time.

Lemma. For all $i \leq k$, we have: $f_{g_i} \leq f_{o_i}$.

Proof. (By induction) Base case: $i = 1$ is true, why?

- Assume holds for $k-1$: $f_{g_{k-1}} \leq f_{o_{k-1}}$
- For kth job, note that $f_{o_{k-1}} \leq s_{o_k}$ (why?)
- Using inductive hypothesis: $f_{g_{k-1}} \leq s_{o_k}$
- Greedy picks earliest finish time among compatible jobs (which includes o_k) thus $f_{g_k} \leq f_{o_k}$

Are We Done? Almost

Lemma. The greedy algorithm returns an optimal set of jobs S, that is, $k = m$.

Proof. (By contradiction)

Suppose S is not optimal, then the optimal set $\mathcal{O}$ must select more jobs, that is, $m \geq k$.

That is, there is a job o_{k+1} that starts after o_k ends

What is the contradiction?

Are We Done? Almost

Lemma. The greedy algorithm returns an optimal set of jobs S, that is, $k = m$.

Proof. (By contradiction)

Suppose S is not optimal, then the optimal set $\mathcal{O}$ must select more jobs, that is, $m \geq k$.

That is, there is a job o_{k+1} that starts after o_k ends

What is the contradiction?

The greedy algorithm keeps selecting jobs until no more compatible jobs left. Since $f_{g_k} \leq f_{o_k}$, greedy would also select compatible job o_{k+1} ($\Rightarrow\Leftarrow$) $\blacksquare$

Implementation & Running Time

Algorithmic steps:

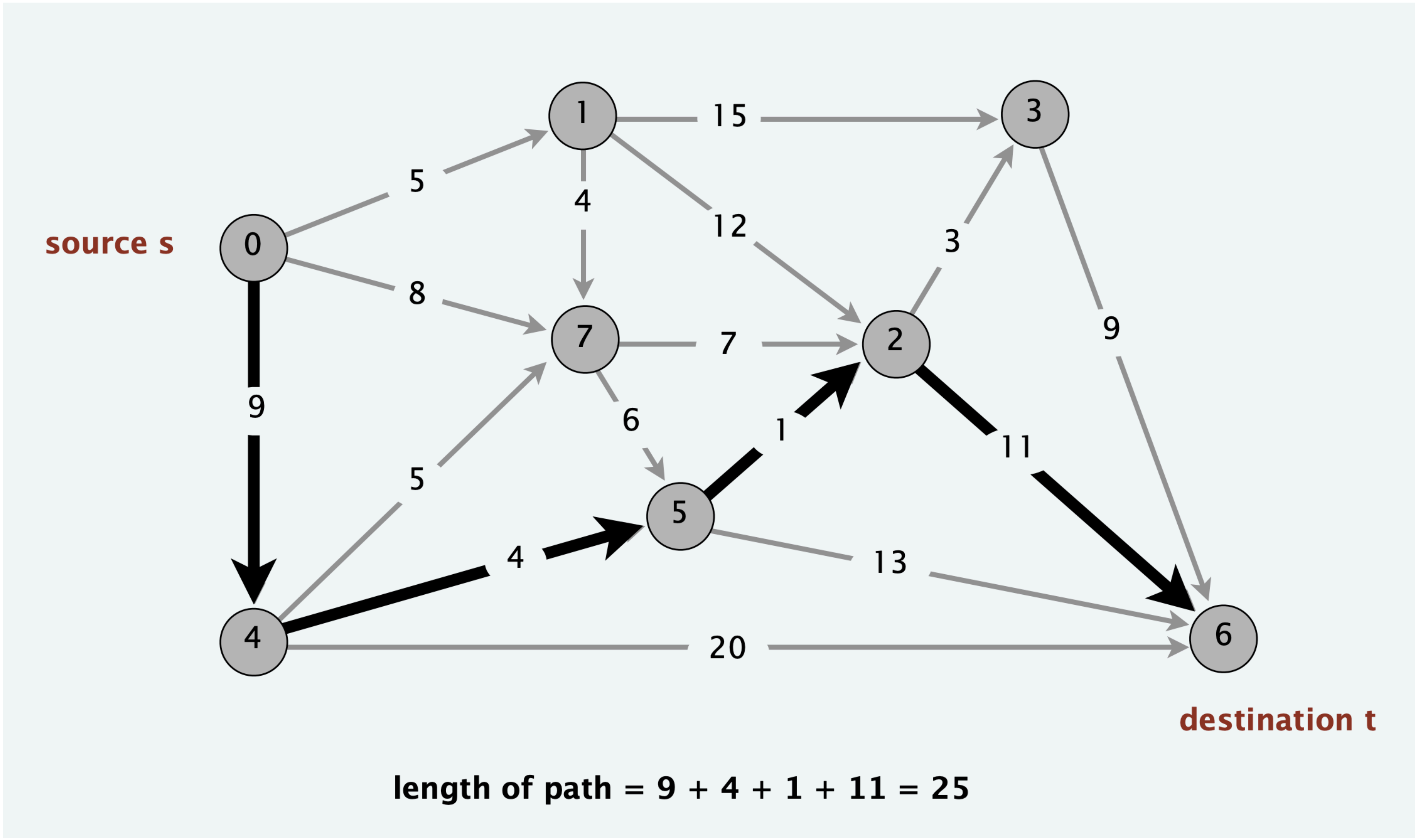
- Sorting jobs by finish times
- Permuting start times in the order of finish times
- For each selected job i, we need to find next job j such that $s_j \geq f_i$
- Overall time

Implementation & Running Time

Algorithmic steps:

- Sorting jobs by finish times
    - $O(n \log n)$
- Permuting start times in the order of finish times
    - $O(n)$
- For each selected job i, we need to find next job j such that $s_j \geq f_i$
    - Can do it in one pass through the jobs
- Overall $O(n \log n)$ time

Shortest Paths in Weighted Graph

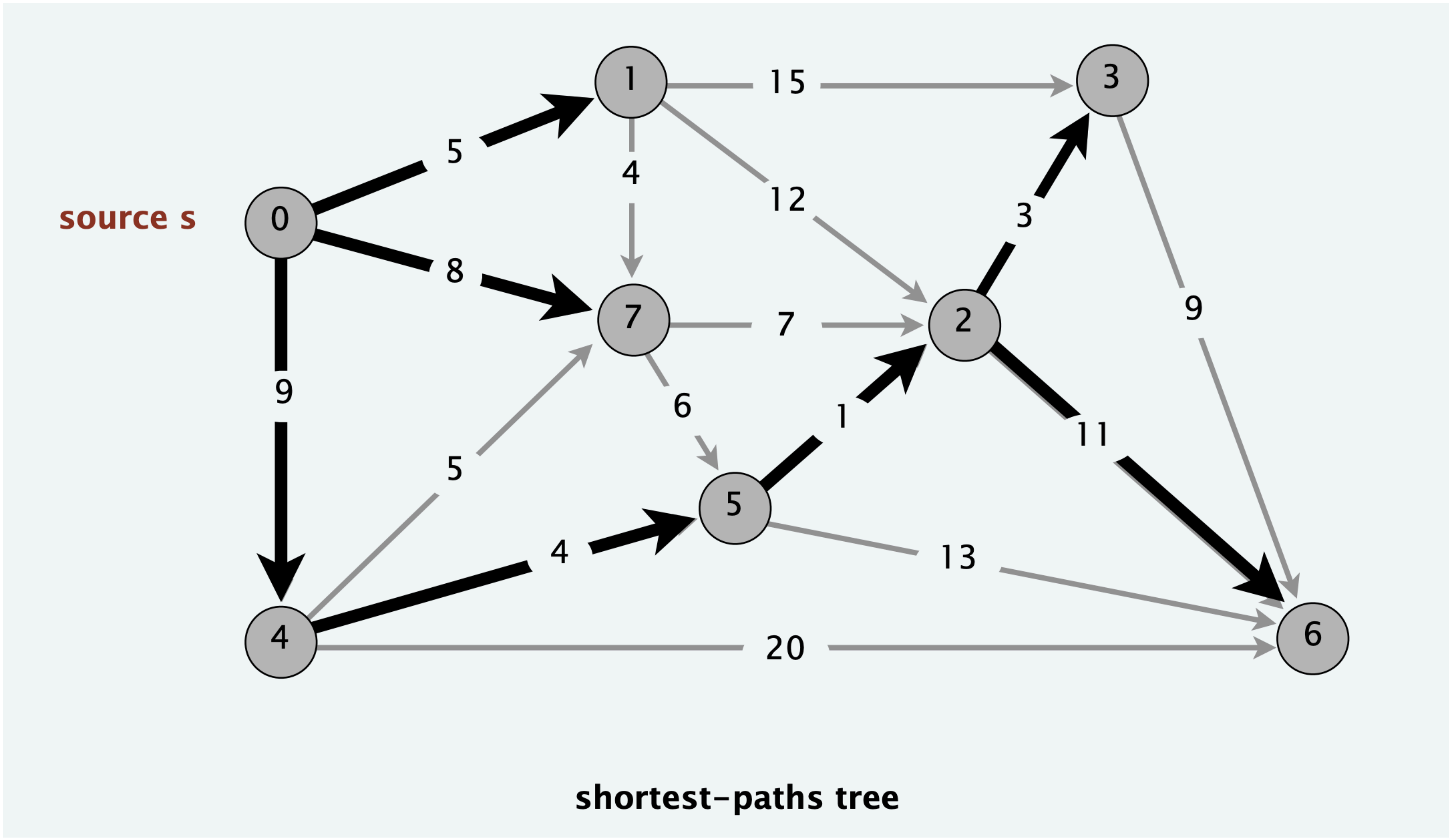


Shortest Paths in Weighted Graph

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 $w(P) = \sum_{e \in P} w(P)$.

Single-Source Shortest Path



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.

Assumption. There exists a path from s to all vertices

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

- Adding 17
- Multiplying 17
- 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!

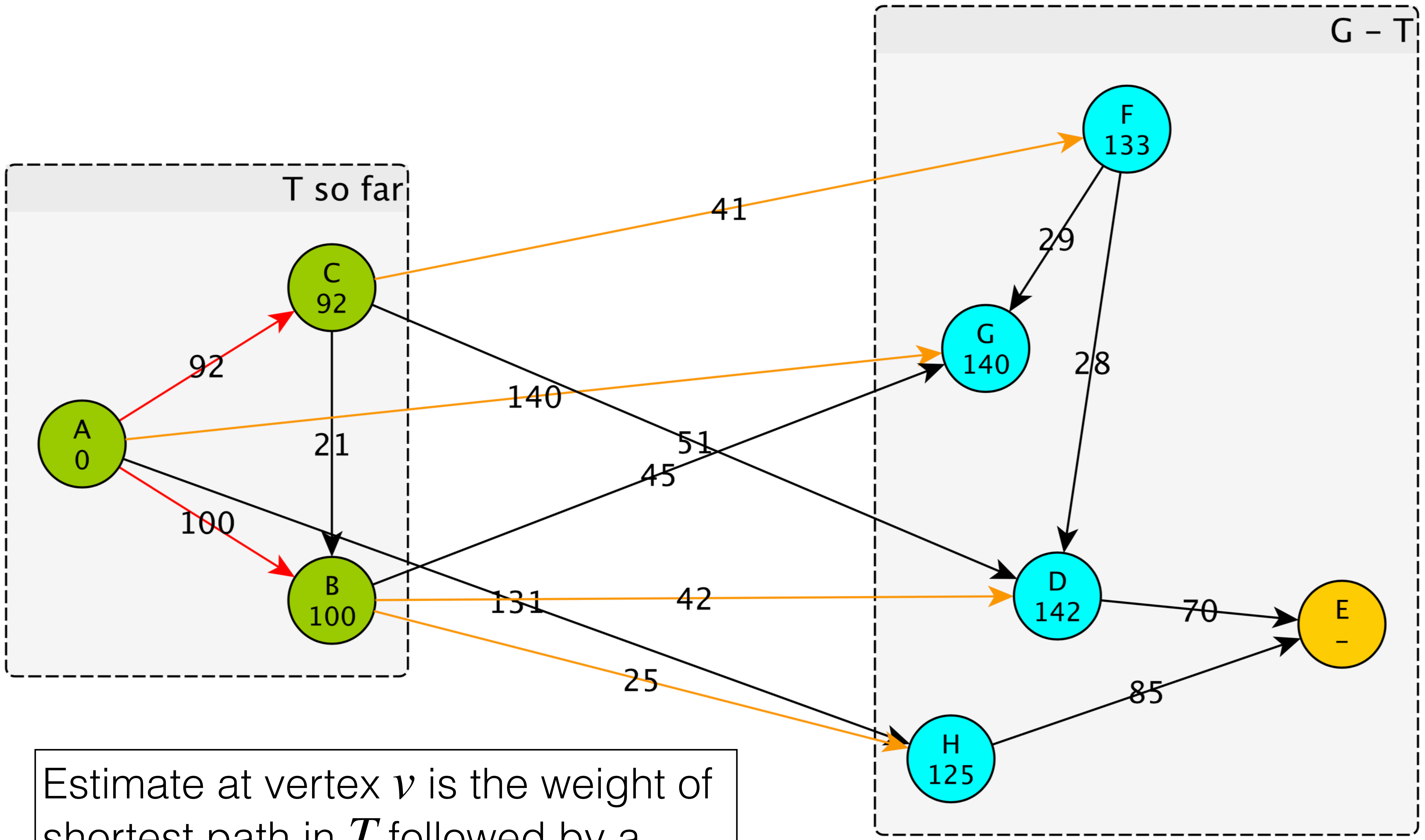
Dijkstra's Algorithm

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



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

Dijkstra's Algorithm

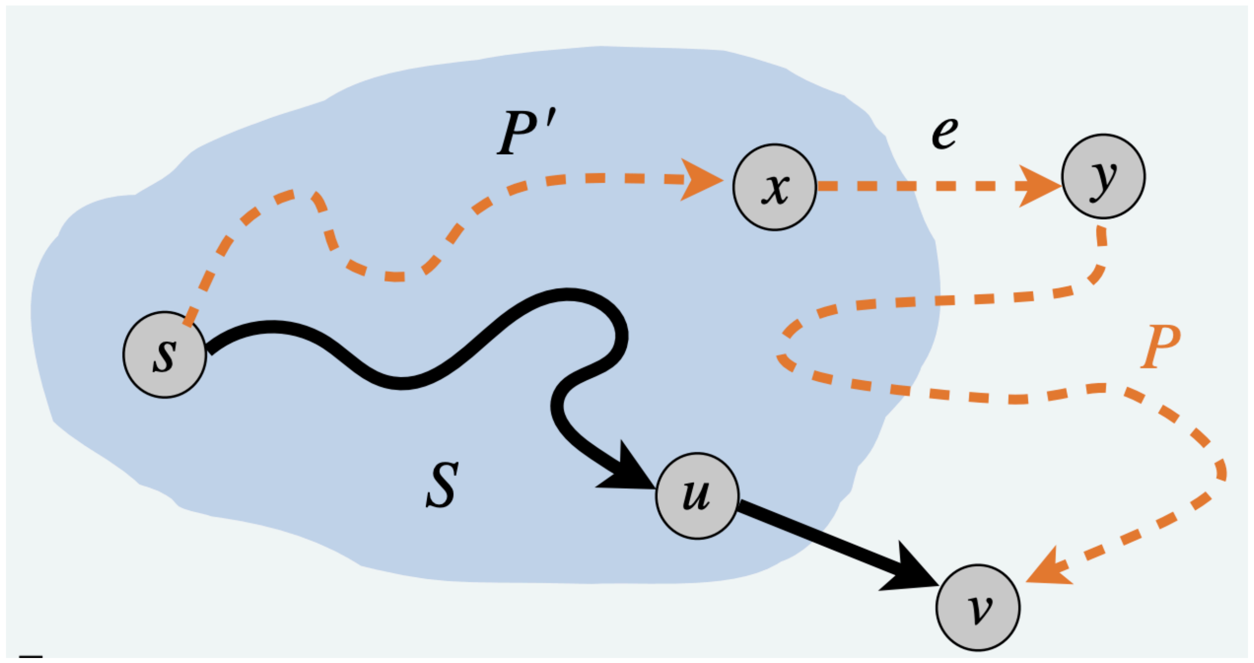
```
procedure Dijkstra(G, s)    # G = (V,E) is connected
    T = ∅; S = {s}; d[s] ← 0      # initialize
    for all neighbors v of s:
        d[v] ← w(s, v); pred[v] ← s
    for all non-neighbors v of s:
        d[v] ← ∞
    while S not equal to V:
        select v from V - S with minimum d[v]
        Add v to S;  add (v, pred[v]) to T
        for each neighbor u of v in V - S:
            if d[v] + w(v, u) < d[u]:   # found a better path
                d[u] = d[v] + w(v, u)
                pred[u] ← 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

- Suppose some other s-v path P in G is shorter
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- Claim: $w(P) \geq d[v]$ as soon as it reaches y



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

- Suppose some other s-v path P in G is shorter
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- Claim: $w(P) \geq d[v]$ as soon as it reaches y

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

Non-negative weights

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

- Suppose some other s-v path P in G is shorter
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- Claim: $w(P) \geq d[v]$ as soon as it reaches y

$$w(P) \geq \underbrace{w(P') + w_e \geq d[x] + w_e}_{\text{Inductive Hypothesis}} \geq d[y] \geq d[v]$$

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

- Suppose some other s-v path P in G is shorter
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- Claim: $w(P) \geq d[v]$ as soon as it reaches y

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

Definition of d[y]

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

- Suppose some other s-v path P in G is shorter
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- Claim: $w(P) \geq d[v]$ as soon as it reaches y

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

Dijkstra chose v instead of y

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 $(v, (\text{pred}[v])$
- Select the unvisited vertex v that minimizes d[v]
- Update $d[v]$ for unvisited vertices

Priority Queue with Delete-Min

Updating the Priority Queue

How to to update priorities 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
- 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 heapify-up 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)

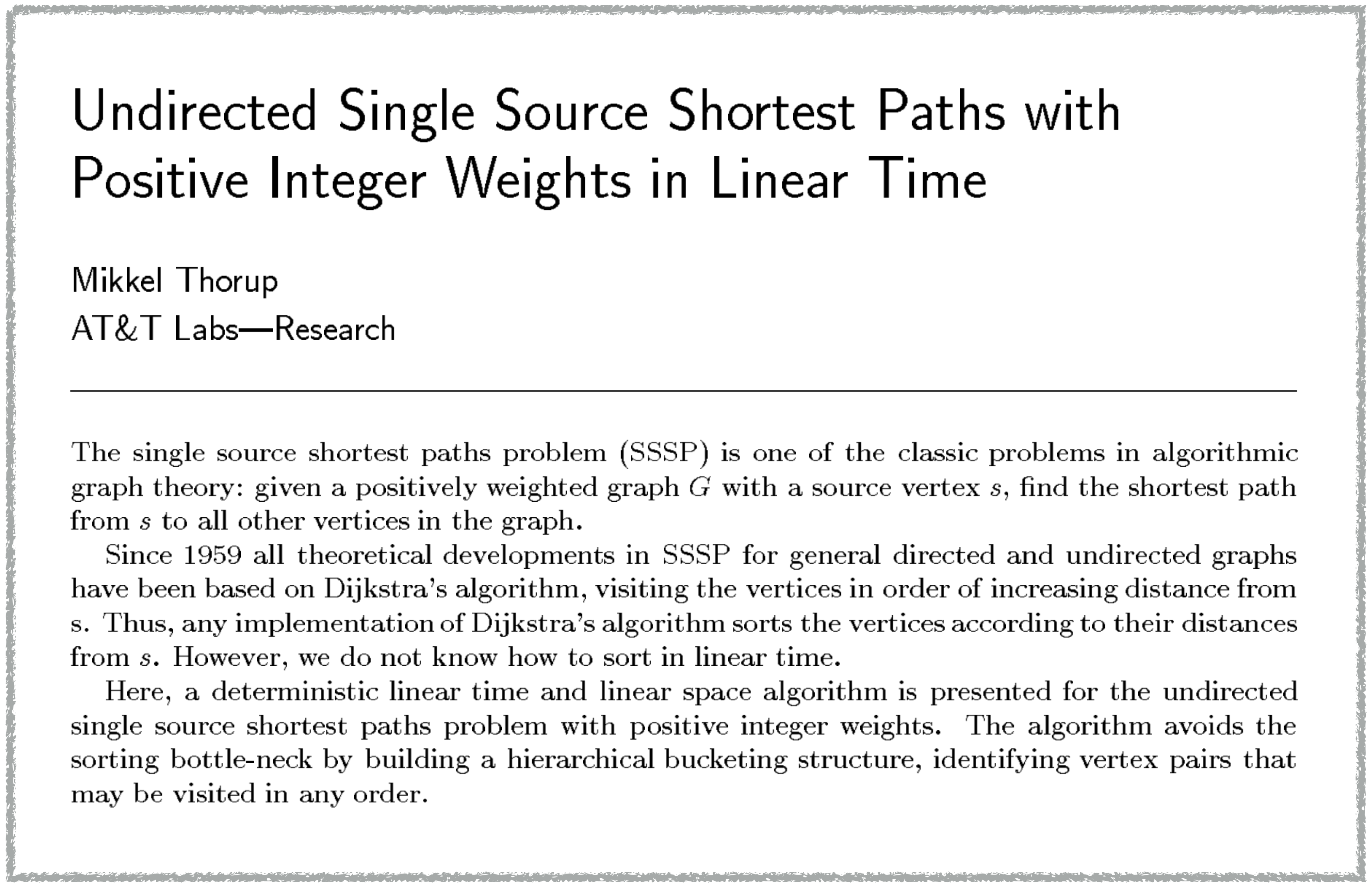
(c) Either A or B

(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

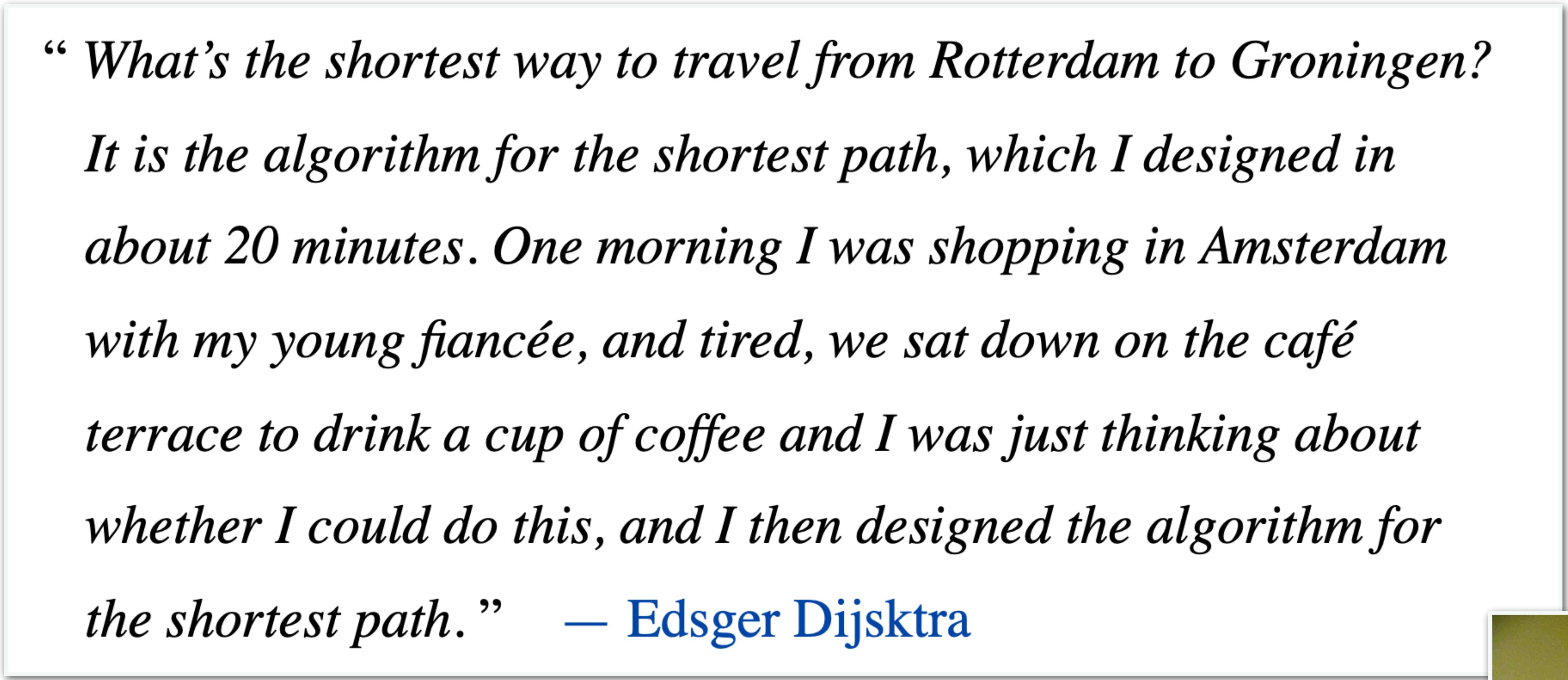
Mikkel Thorup
AT&T Labs—Research

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.

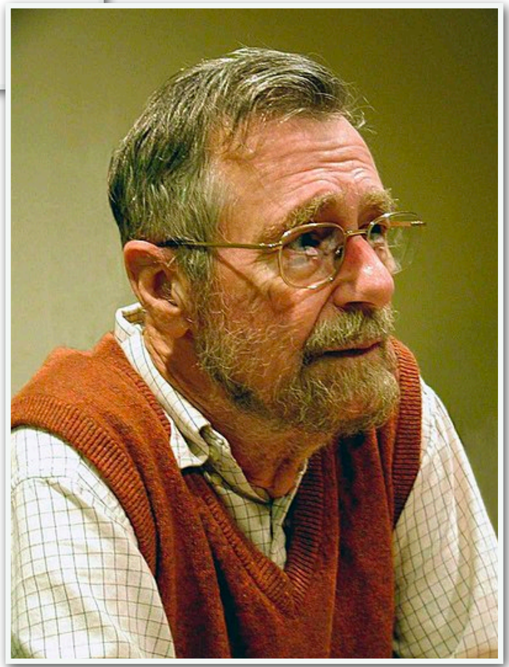
Since 1959 all theoretical developments in SSSP for general directed and undirected graphs have been based on Dijkstra's algorithm, visiting the vertices in order of increasing distance from s. Thus, any implementation of Dijkstra's algorithm sorts the vertices according to their distances from s. However, 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



Acknowledgments

- The pictures 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)