Greedy algorithms
(stay ahead) - shortest paths

Reading: Kleinberg & Tardos
Ch. 4.3
Additional resource: CLRS Ch. 16

Shortest paths

Taking the scenic route

Suppose you have decided to take the summer to explore Europe. You pick a city to be your "home base" and plan to ride the rails each weekend to a different destination.

How do you pick the route that is the…

fastest? cheapest? least annoying (fewest connections)?

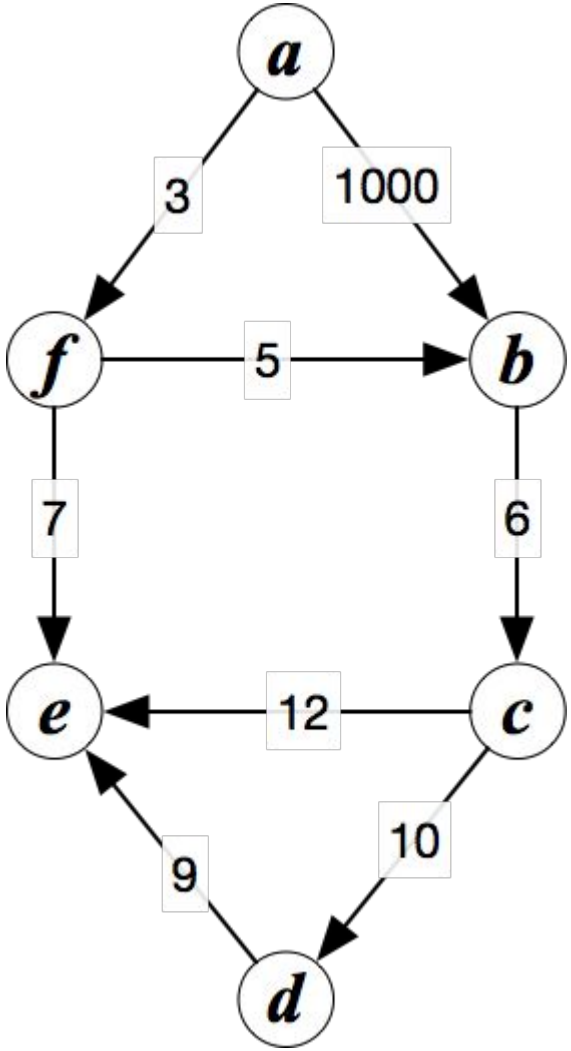
Formulating the problem precisely: shortest paths

Input:

- directed graph $G = (V, E)$
- nonnegative edge lengths $\ell(e)$ for each $e \in E$
- source vertex s

Definitions

- path **length**: sum of edge lengths



Formulating the problem precisely: shortest paths

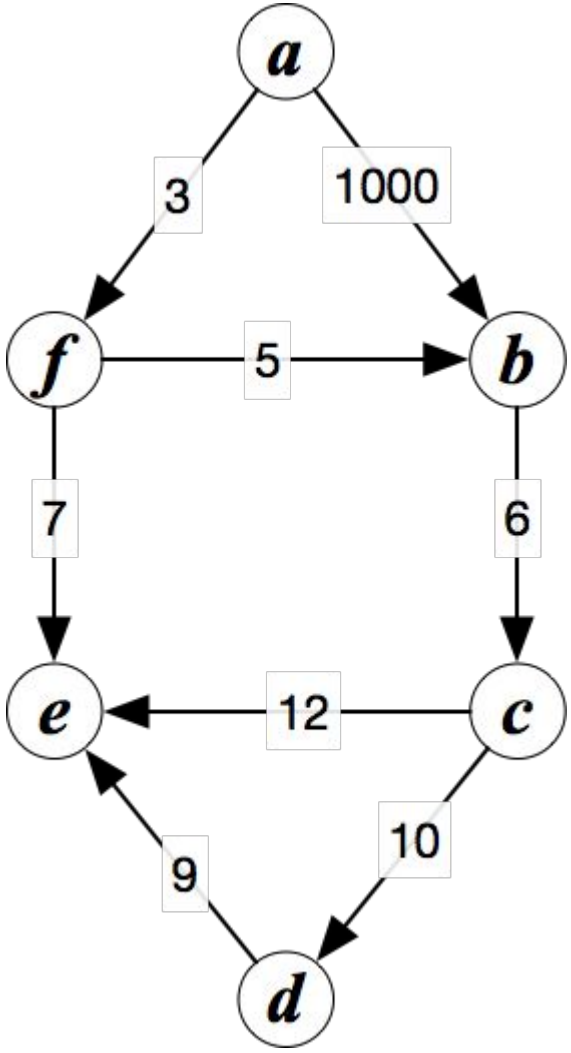
Input:

- directed graph $G = (V, E)$
- nonnegative edge lengths $\ell(e)$ for each $e \in E$
- source vertex s

Definitions

- path **length**: sum of edge lengths $\sum_{i=1}^{p} \ell(e_p)$ $v_0, v_1, \ldots, v_p$ with edges $e_1, e_2, \ldots, e_p$
- **distance** $d(v)$: length of shortest path $s \rightarrow v$

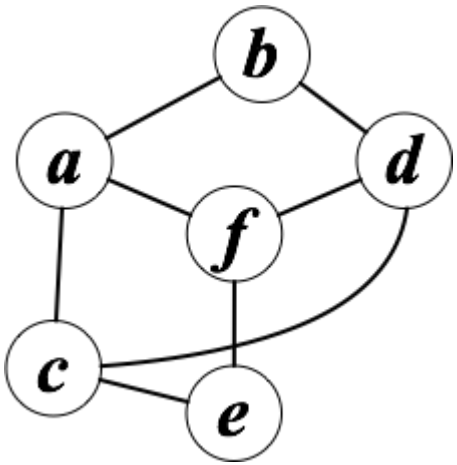
Problem statement: *Can we find d(v) for all v ∈ V?*

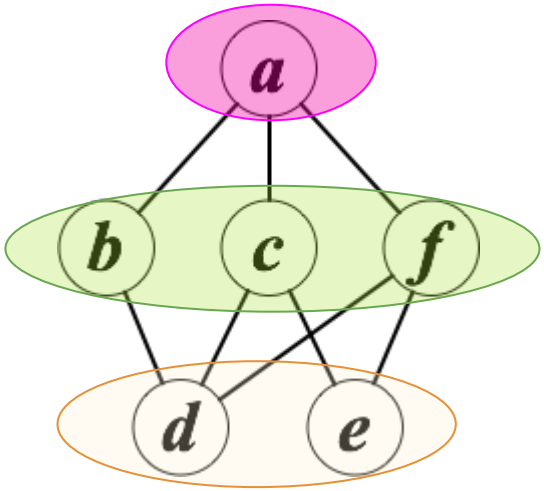


When edge lengths are all the same…

Input:

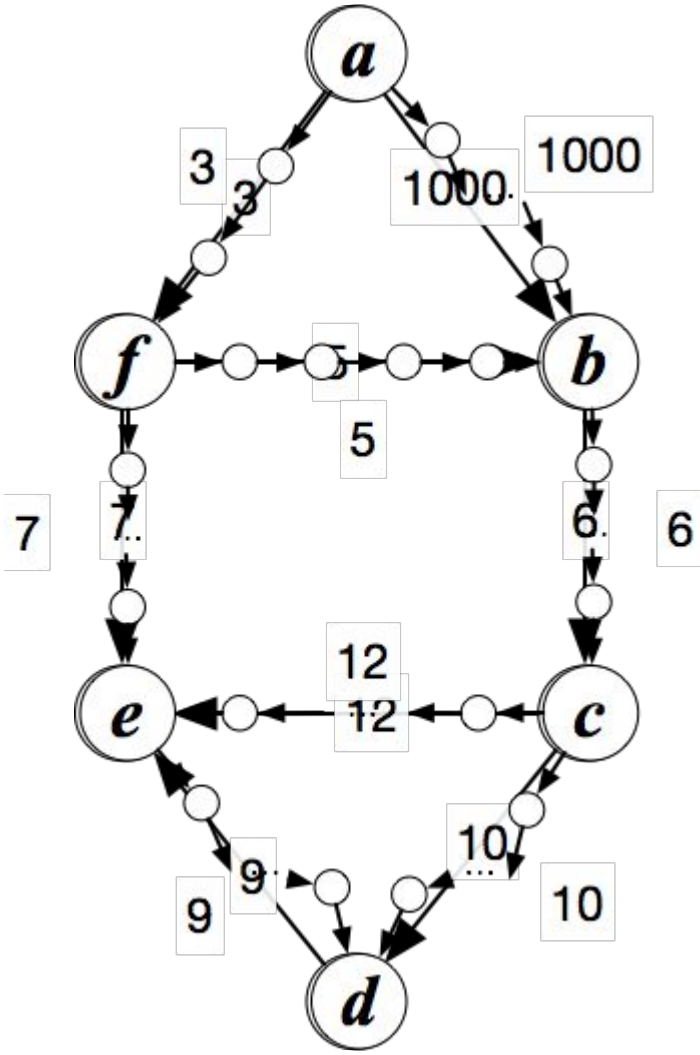
- directed graph $G = (V, E)$
- nonnegative edge lengths $\ell(e)$ for each $e \in E$
- source vertex s





Can we adapt BFS?

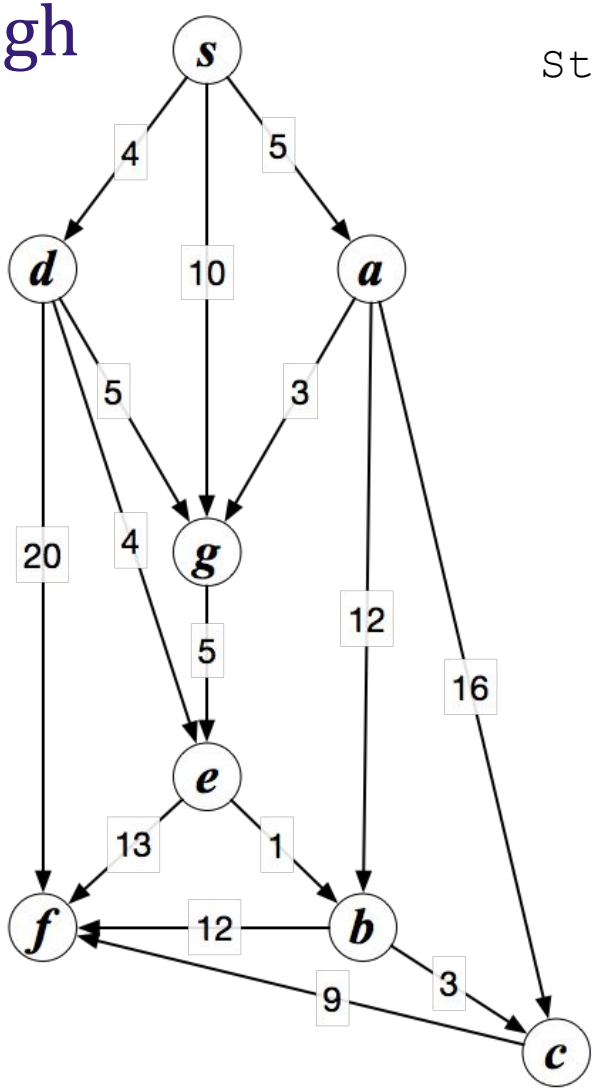
- If edge lengths are integers…



What about non-integer edge lengths?

- BFS: expanding “wavefront”
- Generalize idea of recording expanding “wavefront”
  - want $d(v)$: arrival time at v
- How?
  - **greedily** find next vertex v to be hit by wave
    (unexplored vertex **closest** to s)
  - explore v: update tentative arrival time at each neighbor

Example walkthrough

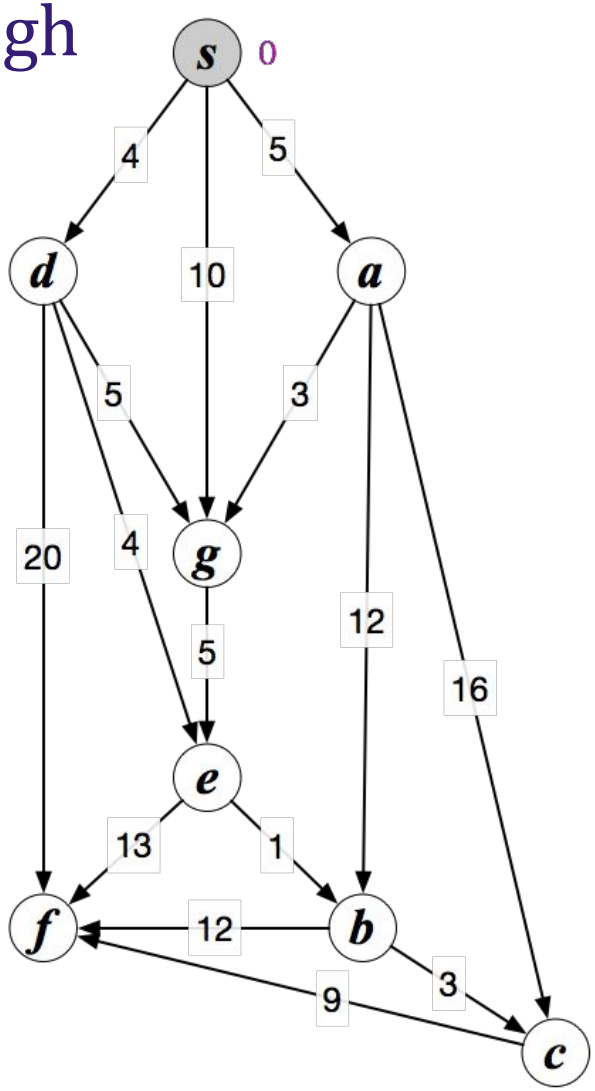


Start at source

Example walkthrough

Update my neighbor distances
(if smaller)

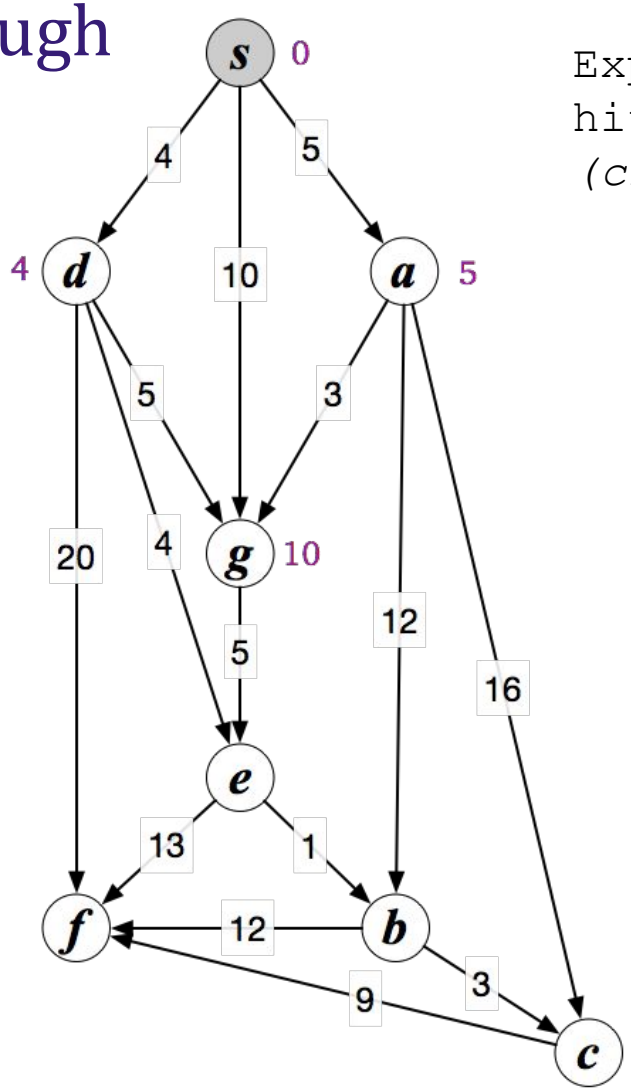
- my d + edge length



Example walkthrough

Update my neighbor distances
(if smaller)

- my d + edge length

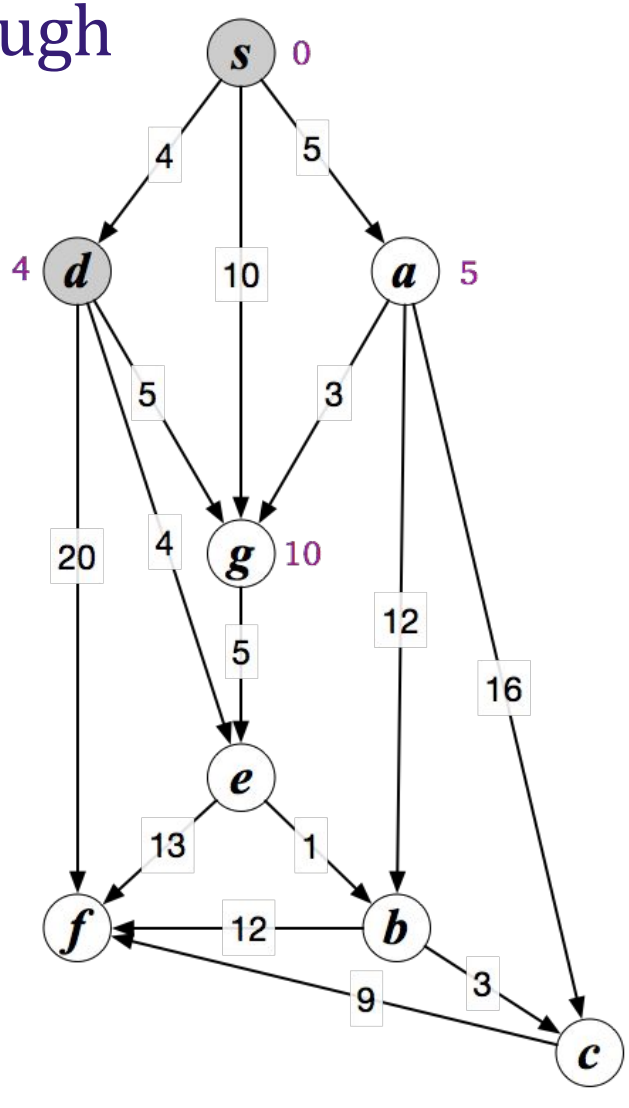


Explore next vertex hit by wavefront
(closest)

Example walkthrough

Update my neighbor distances *(if smaller)*

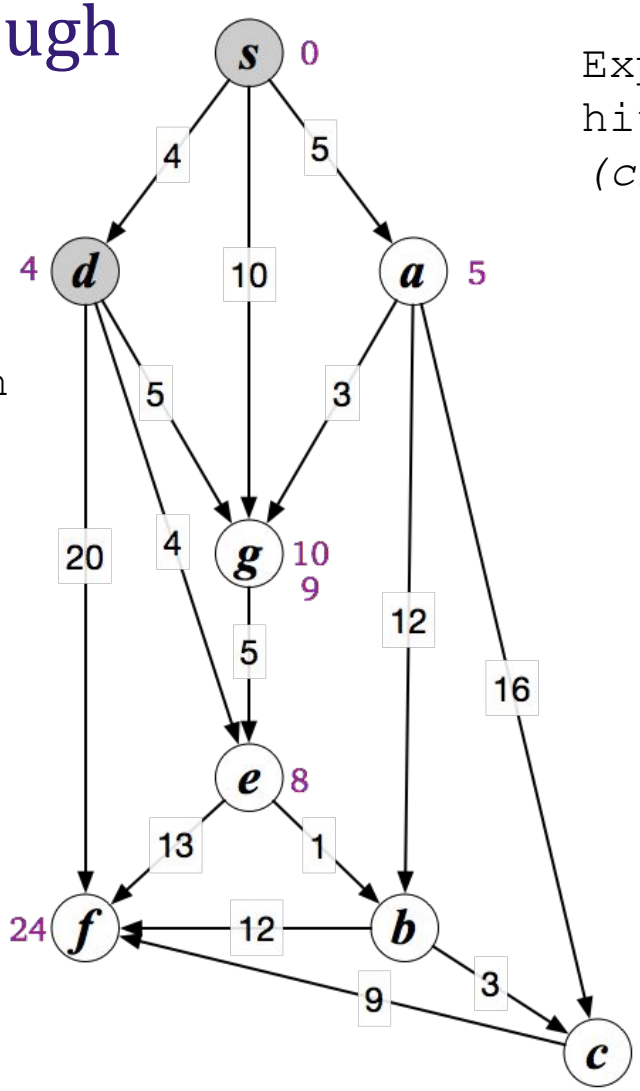
- my d + edge length



Example walkthrough

Update my neighbor distances
(if smaller)

- my d + edge length

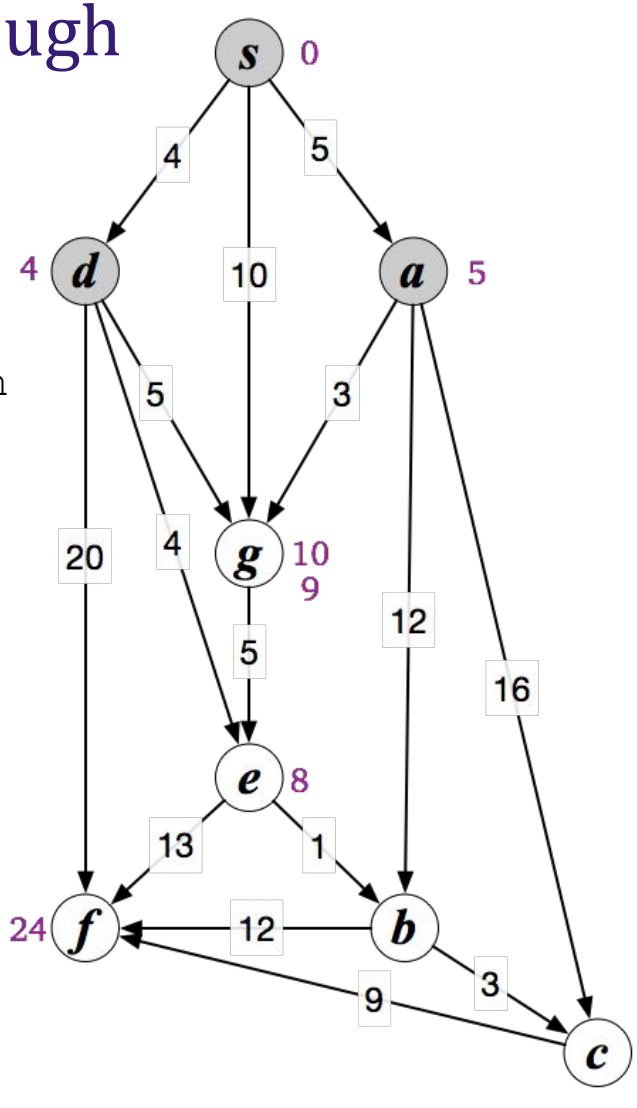


Explore next vertex hit by wavefront
(closest)

Example walkthrough

Update my neighbor distances *(if smaller)*

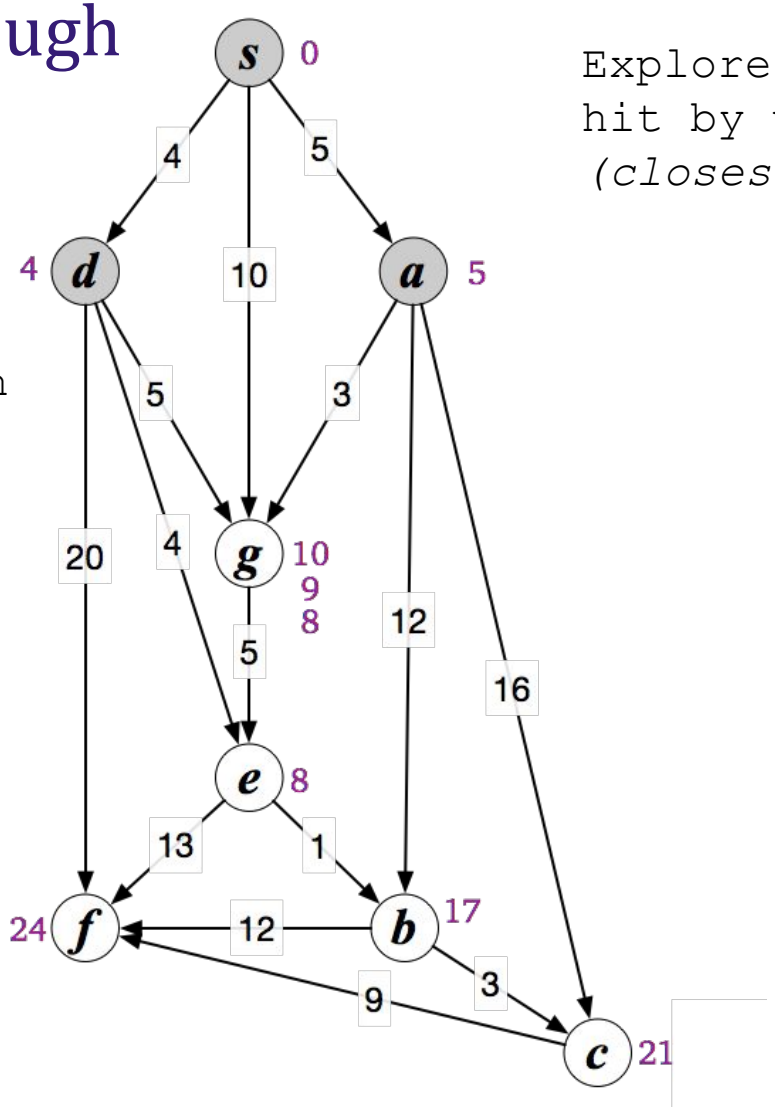
- my d + edge length



Example walkthrough

Update my neighbor distances *(if smaller)*

- my d + edge length

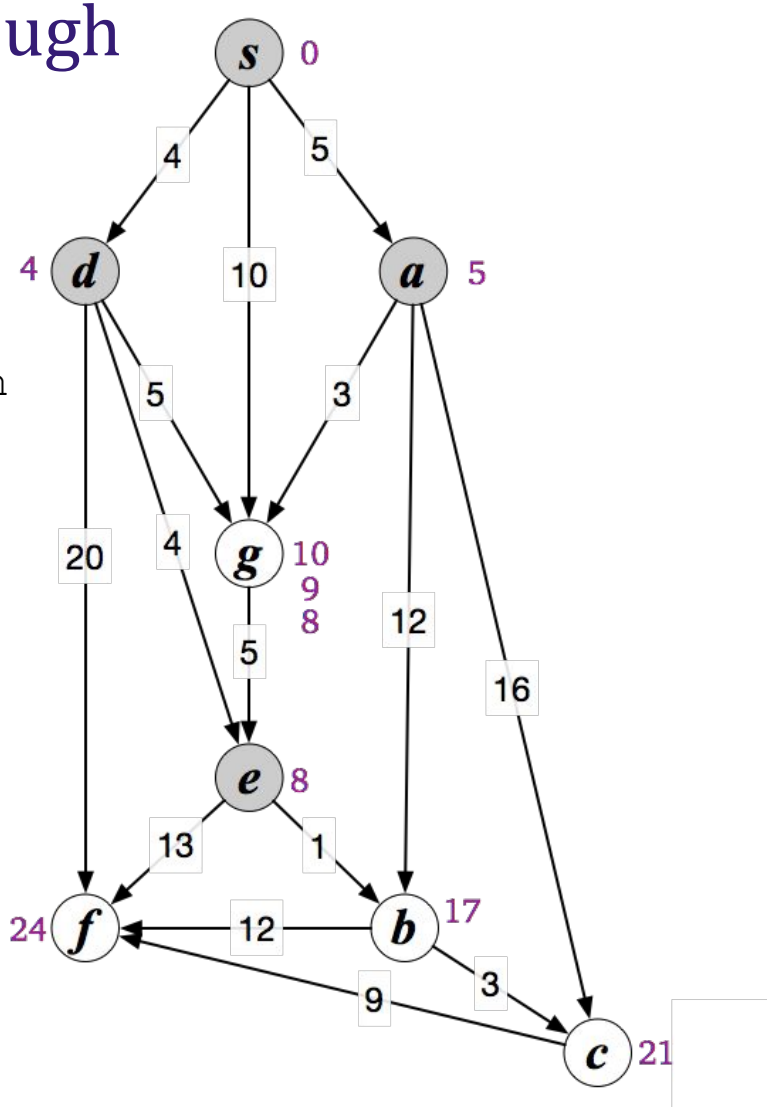


Explore next vertex hit by wavefront *(closest)*

Example walkthrough

Update my neighbor distances *(if smaller)*

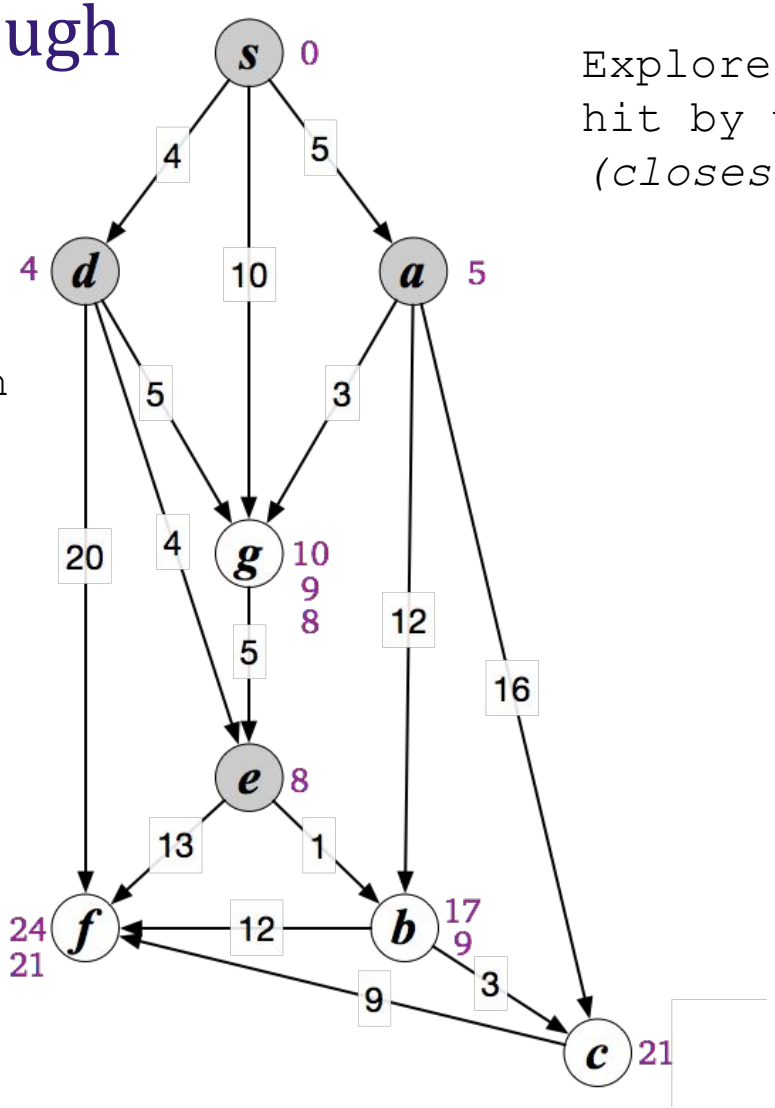
- my d + edge length



Example walkthrough

Update my neighbor distances *(if smaller)*

- my d + edge length

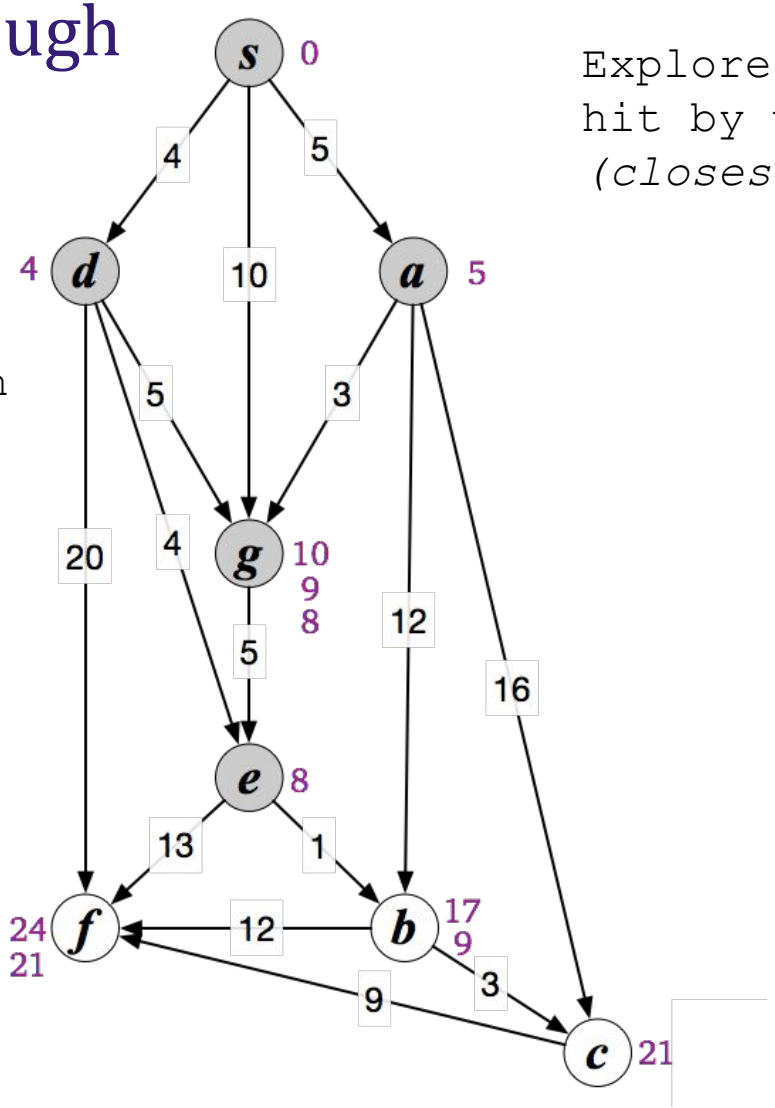


Explore next vertex hit by wavefront *(closest)*

Example walkthrough

Update my neighbor distances *(if smaller)*

- my d + edge length

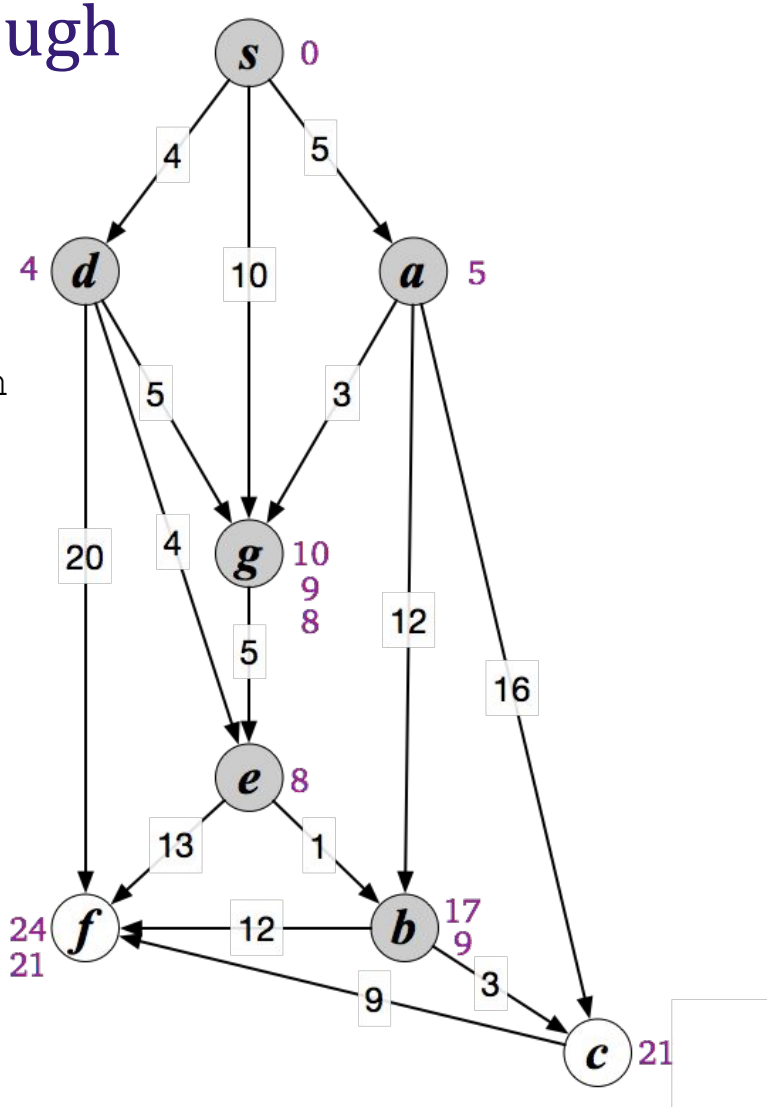


Explore next vertex hit by wavefront *(closest)*

Example walkthrough

Update my neighbor distances
(if smaller)

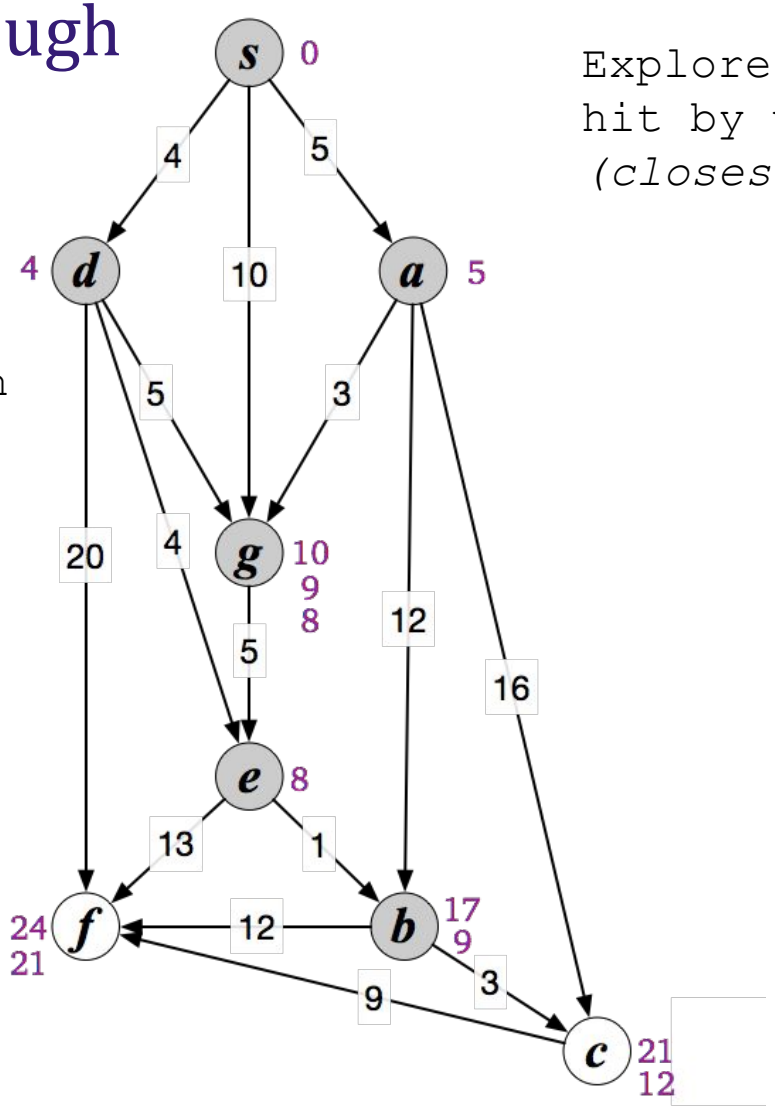
- my d + edge length



Example walkthrough

Update my neighbor distances *(if smaller)*

- my d + edge length

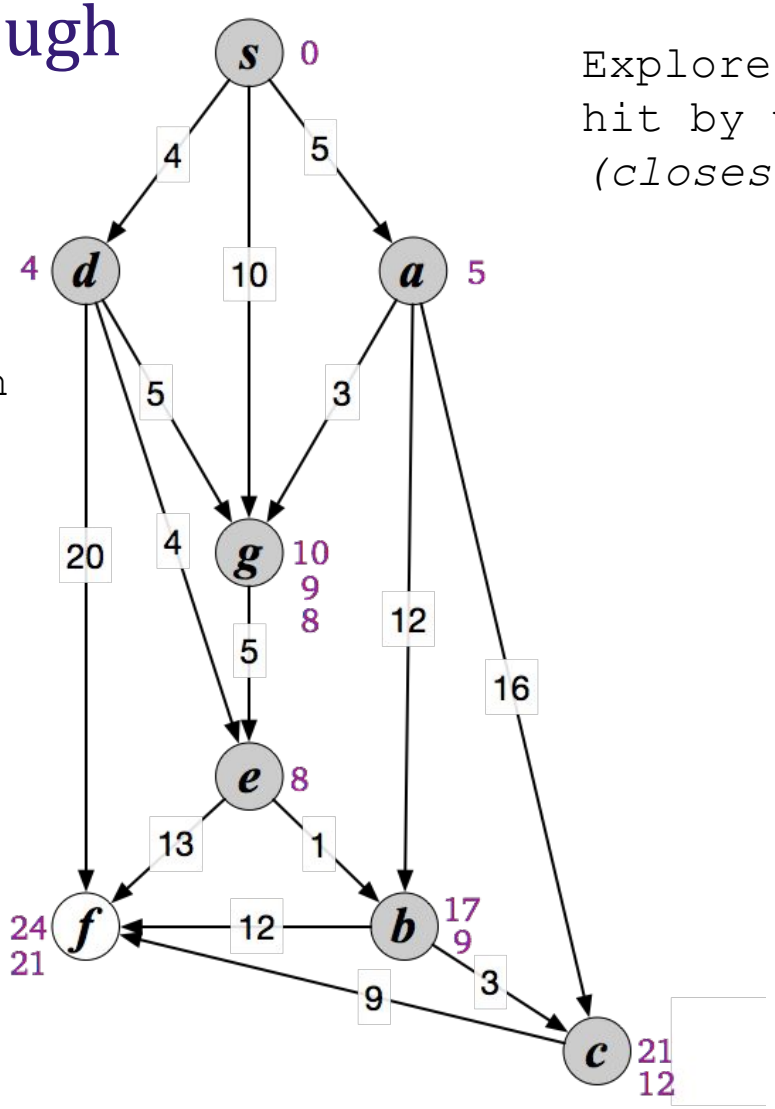


Explore next vertex hit by wavefront *(closest)*

Example walkthrough

Update my neighbor distances *(if smaller)*

- my d + edge length

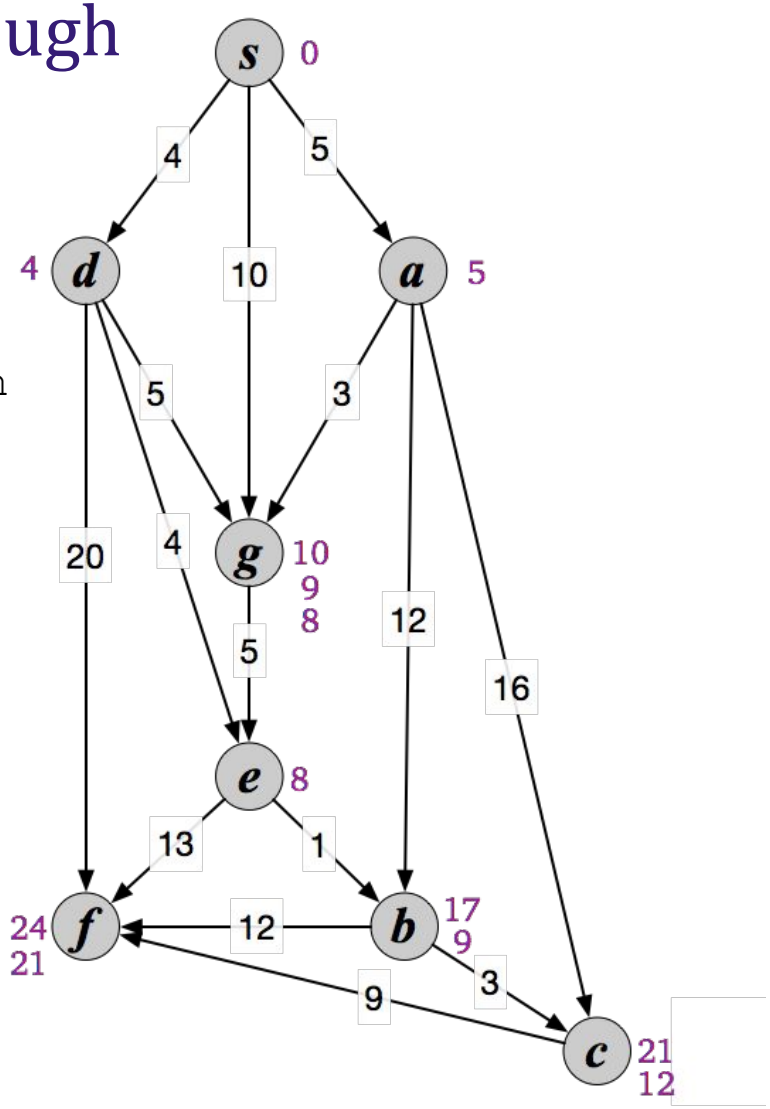


Explore next vertex hit by wavefront *(closest)*

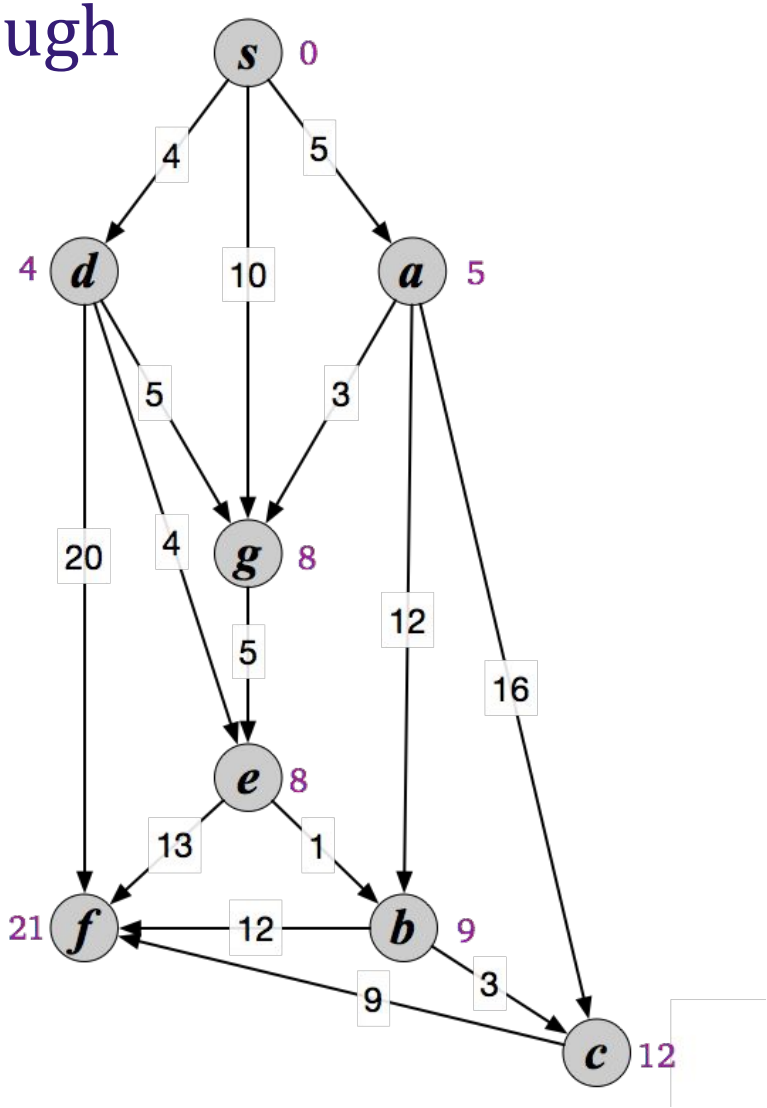
Example walkthrough

Update my neighbor distances *(if smaller)*

- my d + edge length



Example walkthrough final distances



Shortest paths algorithm

Notation:

- $d'(v)$: current earliest arrival time (tentative)
- $d(v)$: shortest distance (final/actual arrival time)

How to keep track of wavefront?

- Find next arrival: v with smallest $d'(v)$
- Actual arrival time now $\rightarrow$ shortest distance: $d(v)=d'(v)$
- Update $d'(v)$ for neighbors if path through v shorter

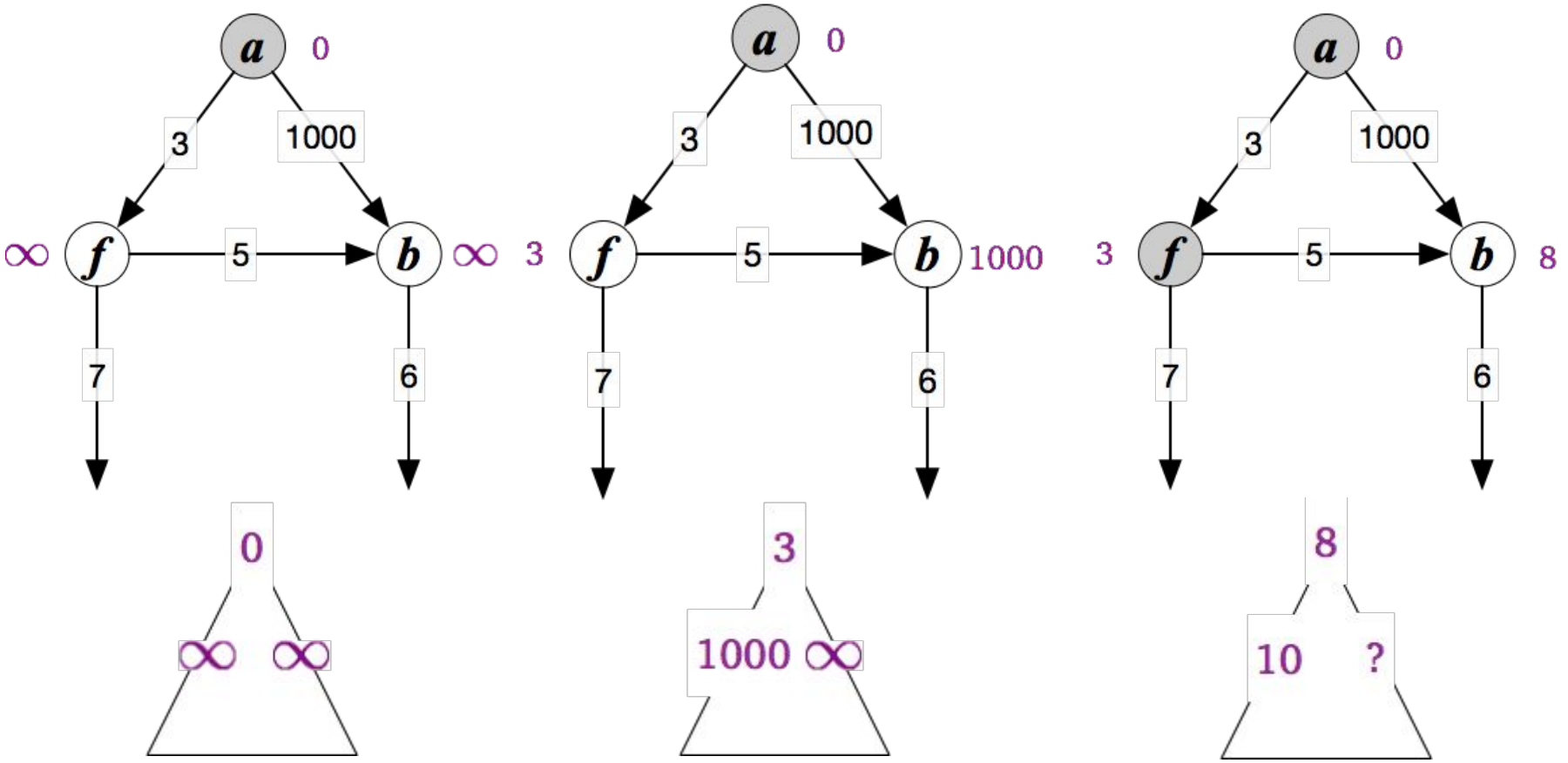
What data structure supports find smallest and update values?

- Priority queue

Dijkstra's algorithm

```
d′(v) = ∞ for all vertices                    // tentative arrival time
A = V                                         // priority queue
d′(s) = 0
while (A not empty)                           // vertices left to explore
    extract v ∈ A with smallest d′(v)
    d(v) = d′(v)                              // wave arrives at v
    for all edges vw where w ∈ A
        if d(v) + ℓ(vw) < d′(w)               // shorter path to w?
            d′(w) =d(v) + ℓ(vw)               // update tentative arrival
```

Priority queue with binary heap update, extract-min in $O(\log n)$



Dijkstra's algorithm: running time

```
d′(v) = ∞ for all vertices                    // tentative arrival time
A = V                                         // priority queue
d′(s) = 0
while (A not empty)                           // vertices left to explore
    extract v ∈ A with smallest d′(v)         // extract-min: O(log n)
    d(v) = d′(v)                              // wave arrives at v
    for all edges vw where w ∈ A
        if d(v) + ℓ(vw) < d′(w)               // shorter path to w?
            d′(w) =d(v) + ℓ(vw)               // update tentative arrival
```

update: O(log *n*)

Dijkstra's algorithm: running time

- extract-min for each vertex: $O(n \log n)$
- update for each edge: $O(m \log n)$
- Total: $O((m + n)\log n)$

```
d'(v) = ∞ for all vertices
A = V
d'(s) = 0
while (A not empty)                          // vertices left to explore
    extract v ∈ A with smallest d'(v)        // extract-min: O(log n)
    d(v) = d'(v)                             // wave arrives at v
    for all edges vw where w ∈ A
        if d(v) + ℓ(vw) < d'(w)              // shorter path to w?
            d'(w) =d(v) + ℓ(vw)              // update tentative arrival
```

update: $O(\log n)$

Proof of correctness - greedy stays ahead (inductively)

Observations

- "**optimal substructure**":
  if u on a shortest path $s \rightarrow u \rightarrow v$, then $s \rightarrow u$ is a shortest path.
- vertices explored in increasing order of distance from s
- at each step, two sets:
  - A (vertices still to explore)
  - $S = V \setminus A$ (explored vertices; $d(v)$ set)

Claim (invariants)

(1) for $v \in A$, $d'(v)$ is length of
shortest $s \rightarrow v$ path with all vertices in S except v

(2) for $v \in S$, $d(v)$ is length of shortest $s \rightarrow v$ path

Proof: invariants

(1) for $v \in A$, $d'(v)$ is length of shortest $s \to v$ path with all vertices in S except v

(2) for $v \in S$, $d(v)$ is length of shortest $s \to v$ path

- at each step, two sets:
  - A (vertices still to explore)
  - $S = V \setminus A$ (explored vertices; $d(v)$ set)

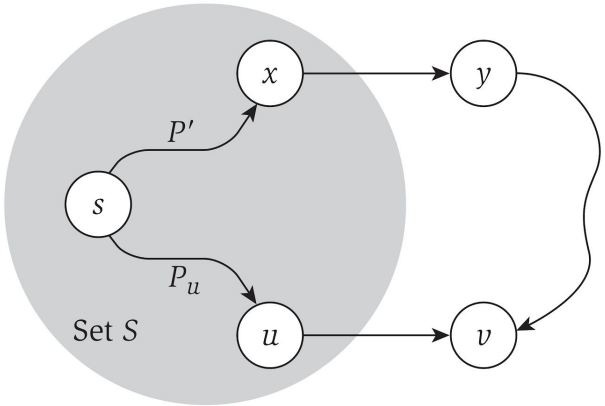


Figure 4.8 The shortest path P_v and an alter

Proof

(1) for $v \in A$, $d'(v)$ is length of shortest $s \rightarrow v$ path with all vertices in S except v

(2) for $v \in S$, $d(v)$ is length of shortest $s \rightarrow v$ path

Base case: $|S| = 0$, when initially $S = \{\}$.

Inductive hypothesis (IH): Assume invariants hold for $|S| = k \geq 0$

Inductive case: $|S| = k + 1$.

Let $v = (k+1)^{st}$ node added to S; then $d'(v) = \min_{w \in A} d'(w)$.

(1) Only neighbors of v had $d'(v)$ updated if shorter path; by IH and construction, (1) holds.

Proof of (2)

(1) for $v \in A$, $d'(v)$ is length of shortest $s \rightarrow v$ path with all vertices in S except v

(2) for $v \in S$, $d(v)$ is length of shortest $s \rightarrow v$ path

- Let u be predecessor of v
- Consider previous iteration (k)
  - S does not contain v
- Consider any other s → v path
- Must leave S via $s \rightarrow x \rightarrow y \rightarrow v$
- Show $|P'| + |x \rightarrow y \rightarrow v| \geq |s \rightarrow u \rightarrow v|$
  - $P_u = s \rightarrow u$ is shortest path by IH
    - w/all vertices (but v) in S
  - Since v chosen over y, $|P'|+\ell(xy) \geq |P_u| + \ell(uv)$
  - Then adding path $y \rightarrow v$ certainly maintains inequality:

    $|P'|+\ell(xy) + |y \rightarrow v| \geq |P_u| + \ell(uv)$
- Thus, $P_u \rightarrow v$ is shortest $s \rightarrow v$ path, so $d(v)$ is correct

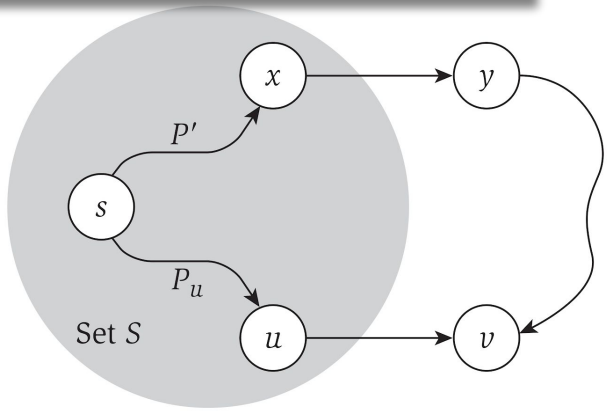


Figure 4.8 The shortest path P_v and an alter

Tracking paths

```
d′(v) = ∞ for all vertices                    // tentative arrival time
pre(v) = null for all vertices                // track predecessor for path
A = V                                         // priority queue
d′(s) = 0
while (A not empty)                           // vertices left to explore
    extract v ∈ A with smallest d′(v)
    d(v) = d′(v)                              // wave arrives at v
    for all edges vw where w ∈ A
        if d(v) + ℓ(vw) < d′(w)               // shorter path to w?
            d′(w) =d(v) + ℓ(vw)               // update tentative arrival
            pre(w) = v                        // remember predecessor
```

Dijkstra's algorithm: which priority queue?

Performance. Depends on PQ: n INSERT, n DELETE-MIN, $\leq m$ DECREASE-KEY.

- Array implementation optimal for dense graphs. ← $\Theta(n^2)$ edges
- Binary heap much faster for sparse graphs. ← $\Theta(n)$ edges
- 4-way heap worth the trouble in performance-critical situations.

priority queue	INSERT	DELETE-MIN	DECREASE-KEY	total
unordered array	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman-Tarjan 1984)	$O(1)$	$O(\log n)$ †	$O(1)$ †	$O(m + n \log n)$
integer priority queue (Thorup 2004)	$O(1)$	$O(\log \log n)$	$O(1)$	$O(m + n \log \log n)$

† amortized

Can we do better?

Thorup 1999: Single-source shortest paths in undirected graphs with positive integer edge lengths in $O(m)$ time

- Does not explore nodes by increasing distance from s

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

MIKKEL THORUP

AT&T Labs Research, Florham Park, New Jersey

Abstract. 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 bottleneck by building a hierarchical bucketing structure, identifying vertex pairs that may be visited in any order.