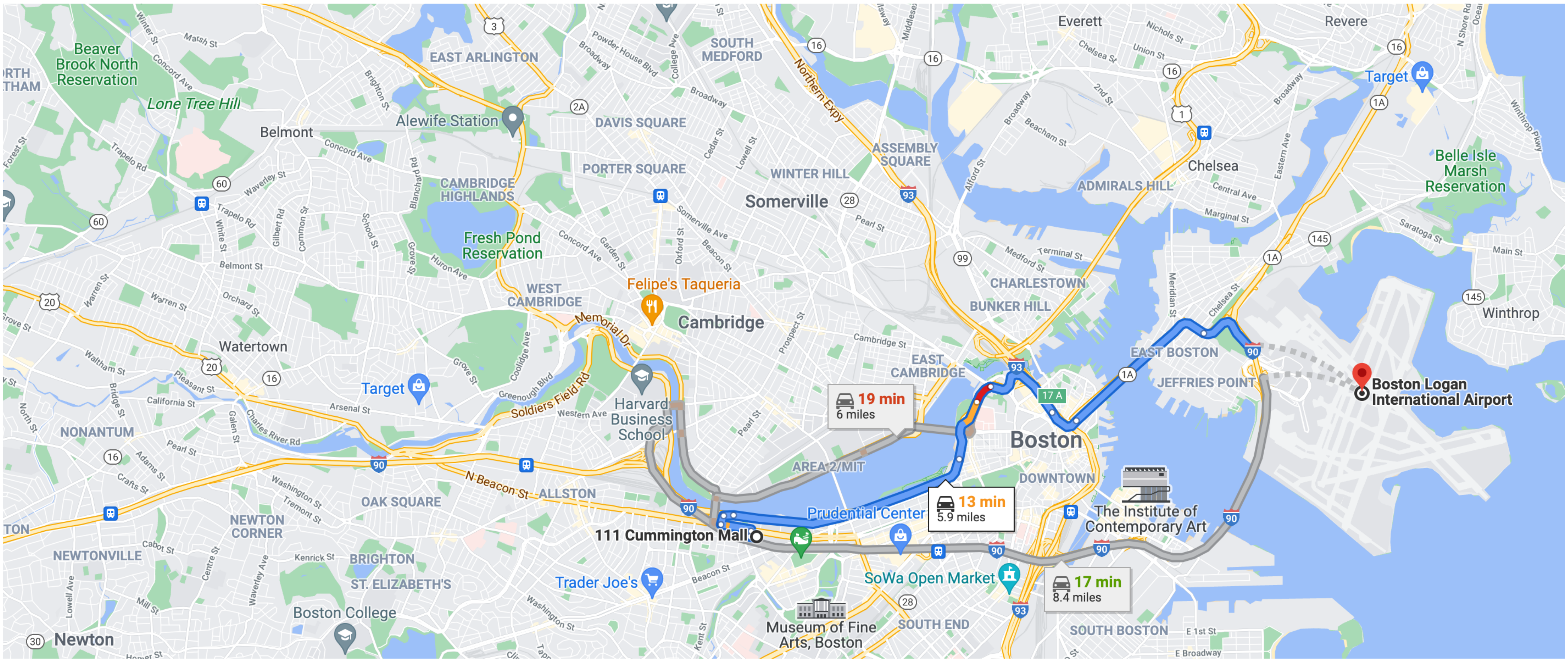
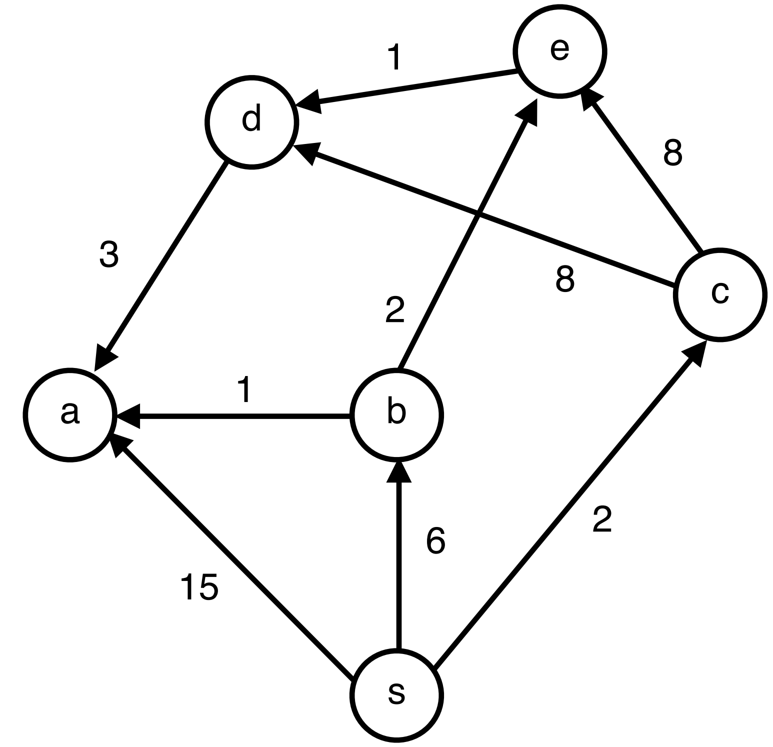
Drive from BU CS to Logan Airport

Single-destination shortest paths problem.

Google Maps 111 Cummington Mall, Boston, MA 02215 to Logan Airport (BOS), Boston, MA Drive 5.9 miles, 13 min



Shortest paths in weighted graphs



Input.

- Directed or undirected graph *G(V,E)* (today: directed)
- (optional) source node *s*, (optional) destination node *t*
- Weight *l(u,v)* on every edge *(u,v)*
  - *l(u,v)* can take any value, e.g. positive or negative, integer or real
  - referred to as "length", "weight", "cost" depending on the application
  - (we're sometimes going to be sloppy and use these words interchangable)
- Length, weight or cost of a path $(v_1, v_2, \ldots, v_k)$ is the sum of values $l(v_i, v_{i+1})$ along the edges.

$$\text{length} = \sum_{i=1}^{k-1} l(v_i, v_{i+1})$$

Shortest paths problem: Find a path between two nodes of minimum total weight

Seam Carving example

original - Broadway tower, Cotswolds, England

https://en.wikipedia.org/wiki/Seam_carving

Seam Carving example

original

scaled

https://en.wikipedia.org/wiki/Seam_carving

Seam Carving example

original

scaled

cropped

https://en.wikipedia.org/wiki/Seam_carving

Seam Carving example

original

paths of least significant pixels

https://en.wikipedia.org/wiki/Seam_carving

Seam Carving example

original

final

paths of least significant pixels

https://en.wikipedia.org/wiki/Seam_carving

Single source shortest paths – Dijkstra's algorithm

Input.

- Directed graph G(V,E)
- Edge lengths $l(u,v) \geq 0$
- source s

Return.

- Distance from s to every node
- Parent table - shortest paths tree from s to each node

Dijkstra's only works with *non-negative* edge weights.

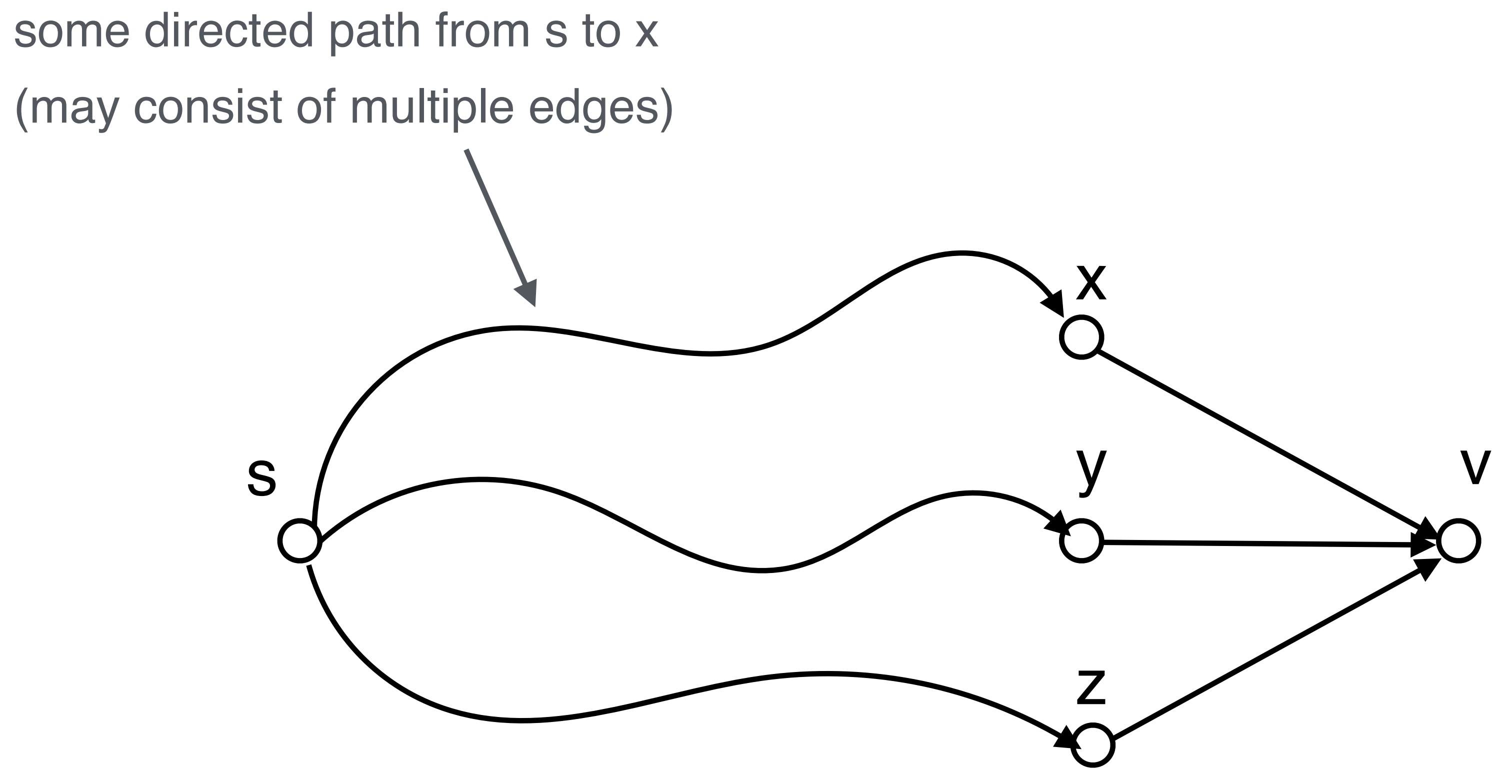
Subpaths of a shortest paths

proposition. Suppose that there is a shortest paths from u to v. Then any subpath between nodes x and y on this path is a shortest path from x to y.

proof:

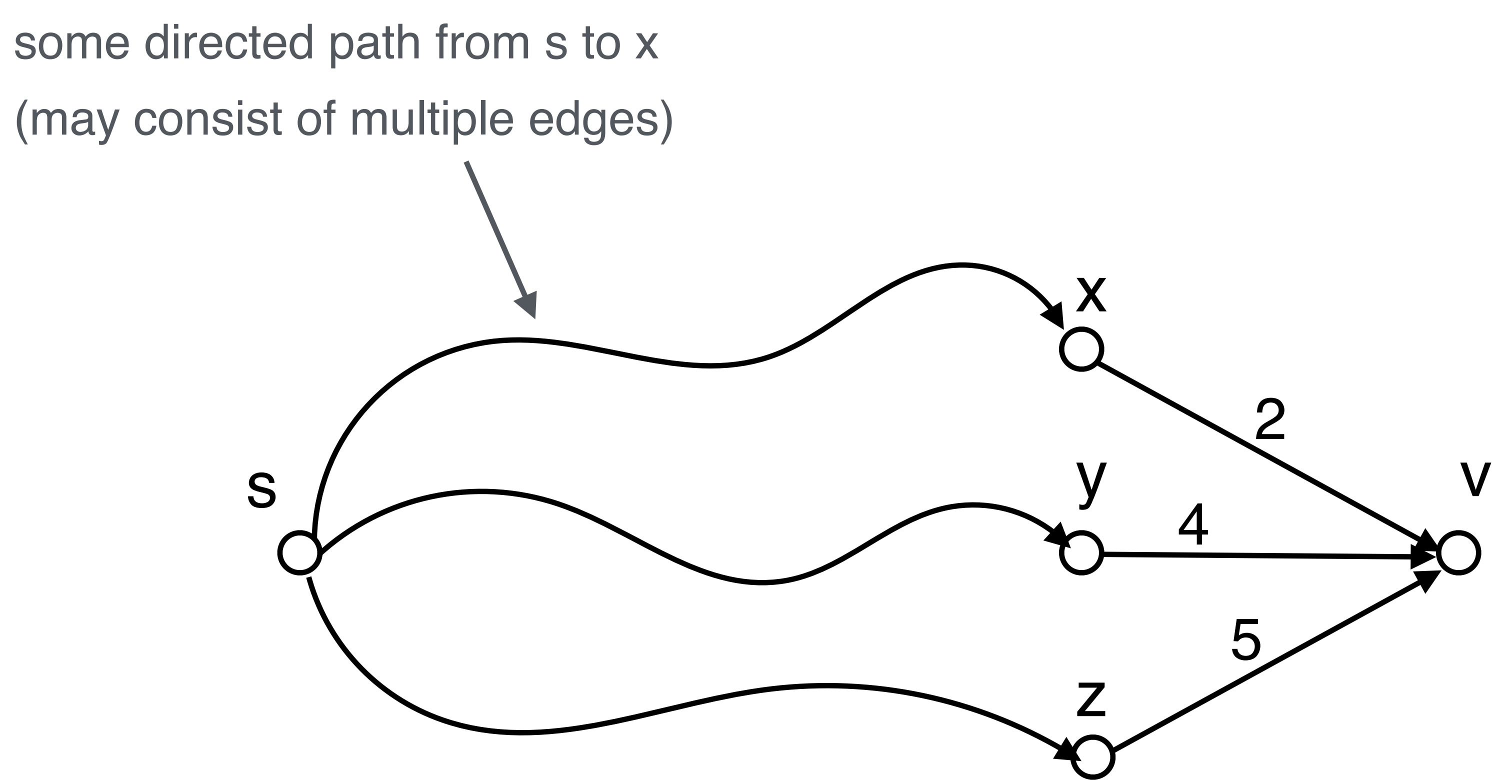
Idea: parts of a shortest paths are also shortest paths

observation. The shortest path from s to v will contain one of the edges (x,v), (y,v) or (z,v)



Top Hat Question

Question. Suppose that node *v* has 3 incoming edges *(x,v)*, *(y,v)* and *(z,v)*. Given the distance from *s* to *x, y, z* and the weights on each edge, what is *dist(s,v)*?



dist(s,x) = 8

dist(s,y) = 4

dist(s,z) = 4

A. dist(s,v) = 7
B. dist(s,v) = 8
C. dist(s,v) = 2
D. dist(s,v) = 4

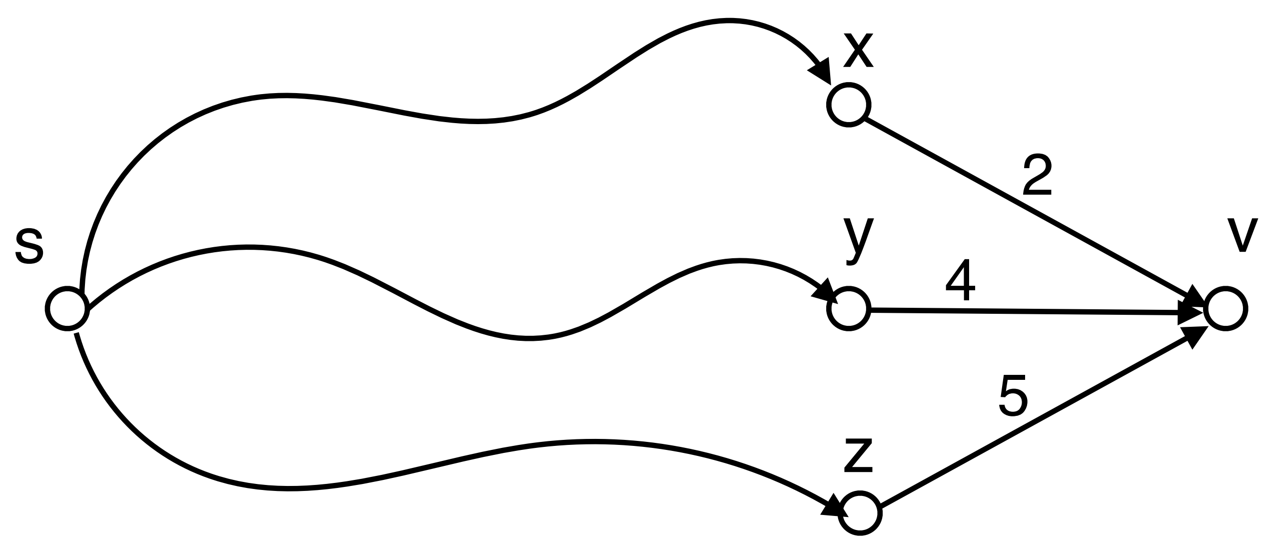
Dijkstra's algorithm — insight

Let *dist(s,u)* denote the shortest path length from s to u.

Claim. Suppose we know *dist(s,u)*. Further, there is an edge *(u,v)* with length *l(u,v)*.
Then we know that $\mathrm{dist}(s,v) \leq \mathrm{dist}(s,u) + l(u,v)$

Top Hat Question

Question. Suppose that node *v* has 3 incoming edges *(x,v)*, *(y,v)* and *(z,v)*. Given the distance from *s* to *x, y, z* and the weights on each edge, which one is the correct formula to compute *dist(s,v)*?



dist(s,x) = 8

dist(s,y) = 4

dist(s,z) = 4

A. $dist(s,v) = \min_{u:\ edge(u,v)} dist(s,u) + \ell(u,v)$

B. $dist(s,v) = \min_{u:\ edge\ (u,v)} \ell(u,v)$

C. $dist(s,v) = \sum_{u:\ edge\ (u,v)} \ell(u,v)$

D. $dist(s,v) = \min_{u:\ edg\ (u,v)} dist(s,u)$

Dijkstra's algorithm overview

For each node *v* we maintain the min length of path we know so far from *s* to *v*.

- this is the *best known* upper bound on dist(s,v) so far
- denoted by $\pi(v)$

Initialize: for each v $\pi(v) = \infty$

In each iteration:

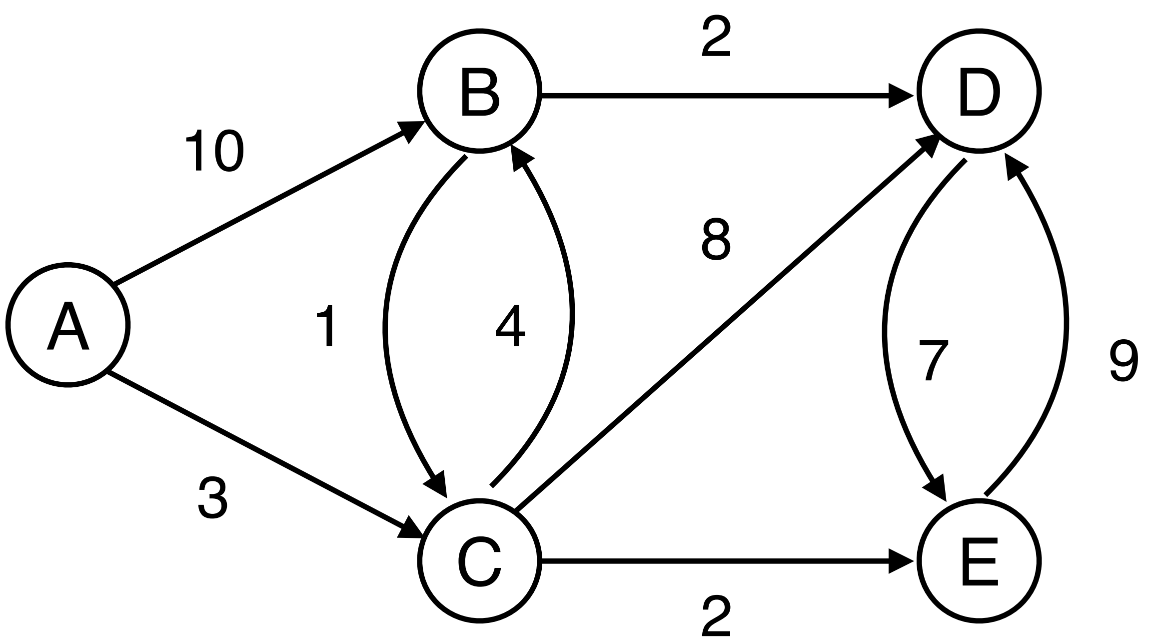
- find u with the lowest $\pi(u)$
- fix the distance *dist(s,u)* to be $dist(s,u) = \pi(u)$
- for each neighbor v of u, update their best known path

$$\pi(v) = \min\{\pi(v), \mathrm{dist}(s,u) + l(u,v)\}$$

Dijkstra's algorithm example

d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v



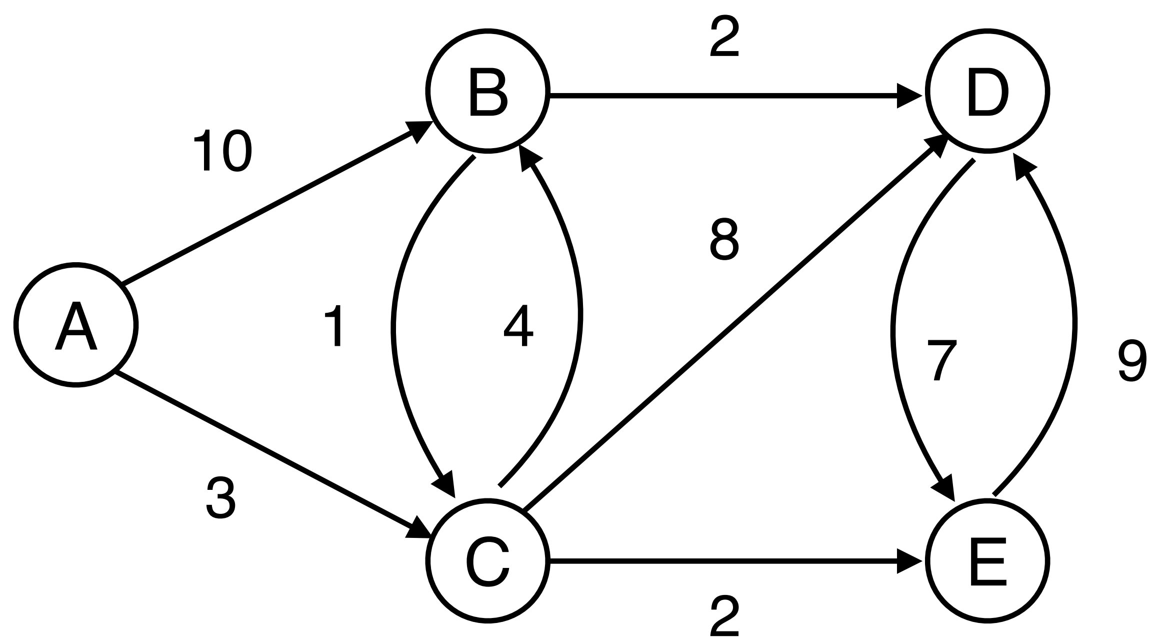
Dijkstra's algorithm example

d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

Initialize:

- s = A
- Maintain data structure *Q*
- Initially for every *v* set $\pi(v) = \infty$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞

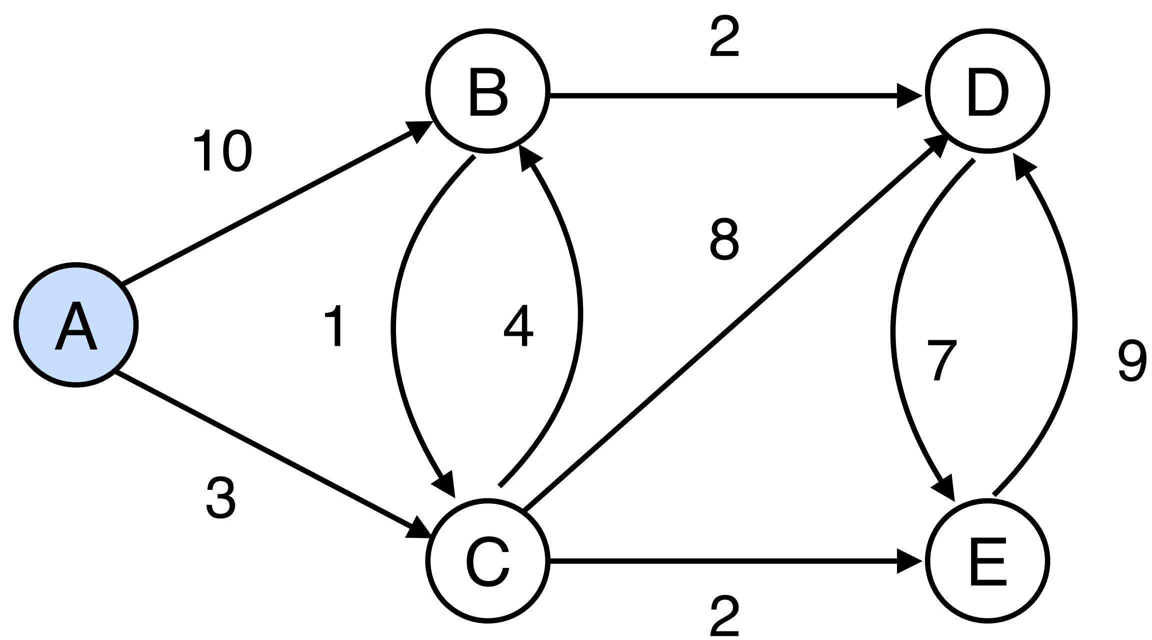
D = {d(A) = 0}

Dijkstra's algorithm example

$d(u)$ = distance from s to u

$\pi(v) =$ currently known shortest distance to v

Initialize:



- s = A
- Maintain data structure *Q*
- Initially for every *v* set $\pi(v) = \infty$

Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞

D = {d(A) = 0}

Dijkstra's algorithm example

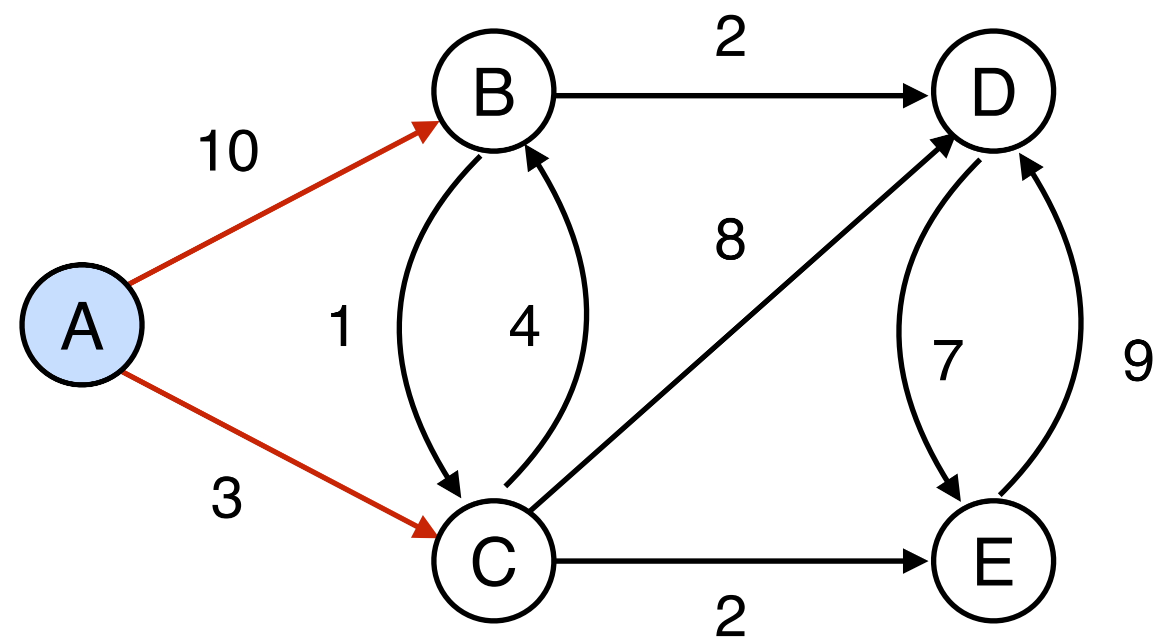
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v,w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞

D = {d(A) = 0}

Dijkstra's algorithm example

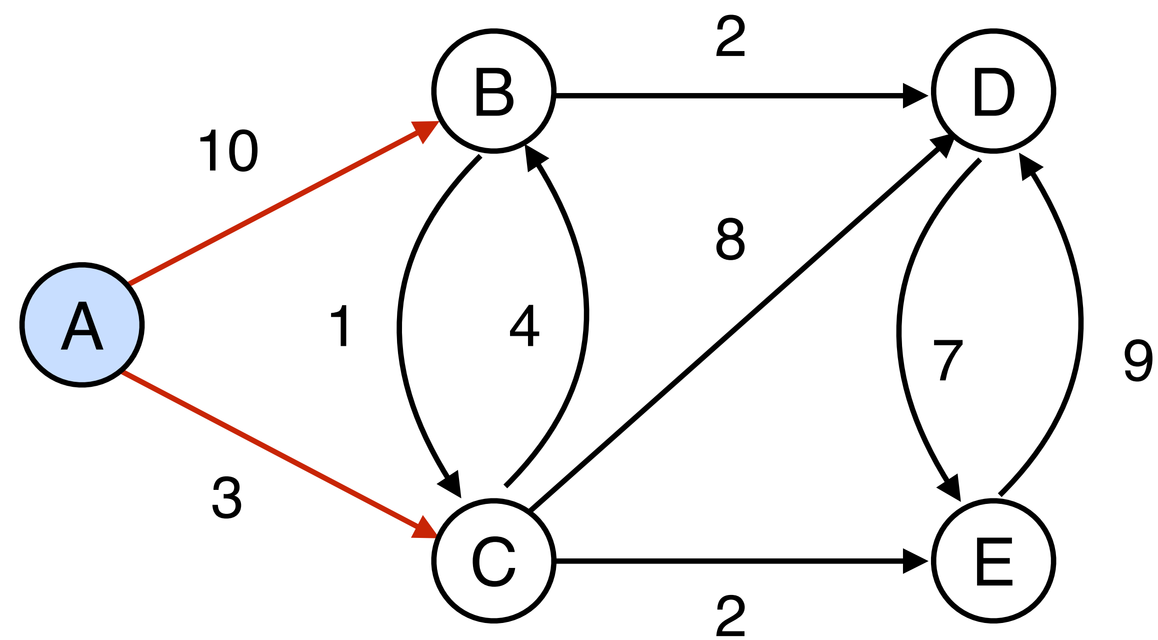
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v, w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞

D = {d(A) = 0}

Dijkstra's algorithm example

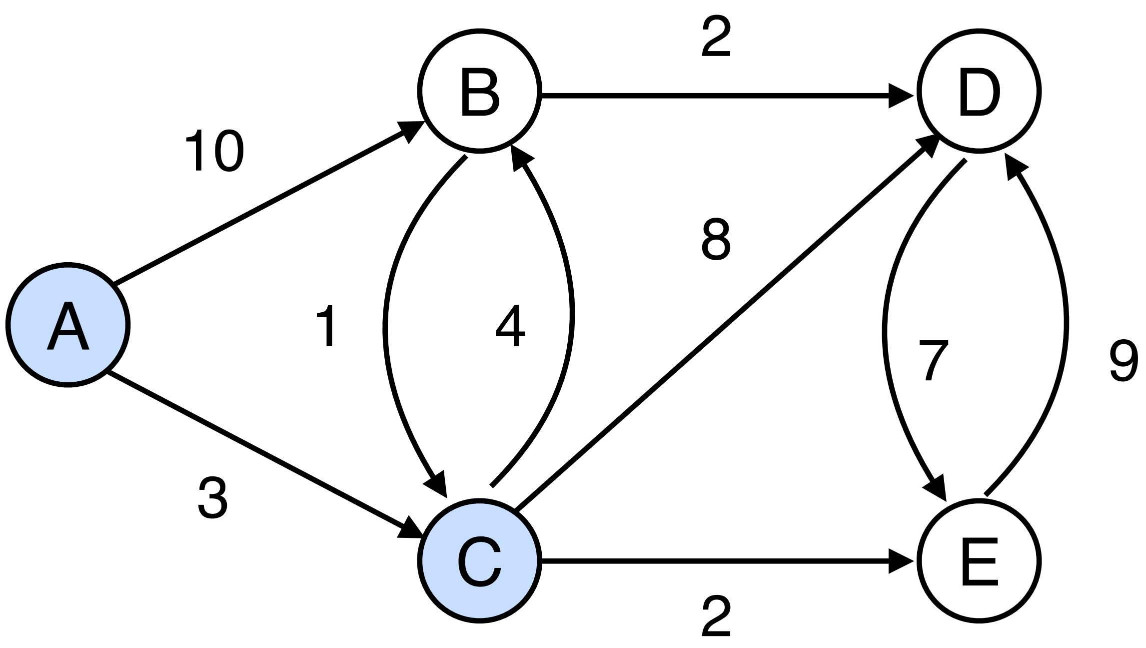
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v, w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞

D = {d(A) = 0, d(C) = 3}

Dijkstra's algorithm example

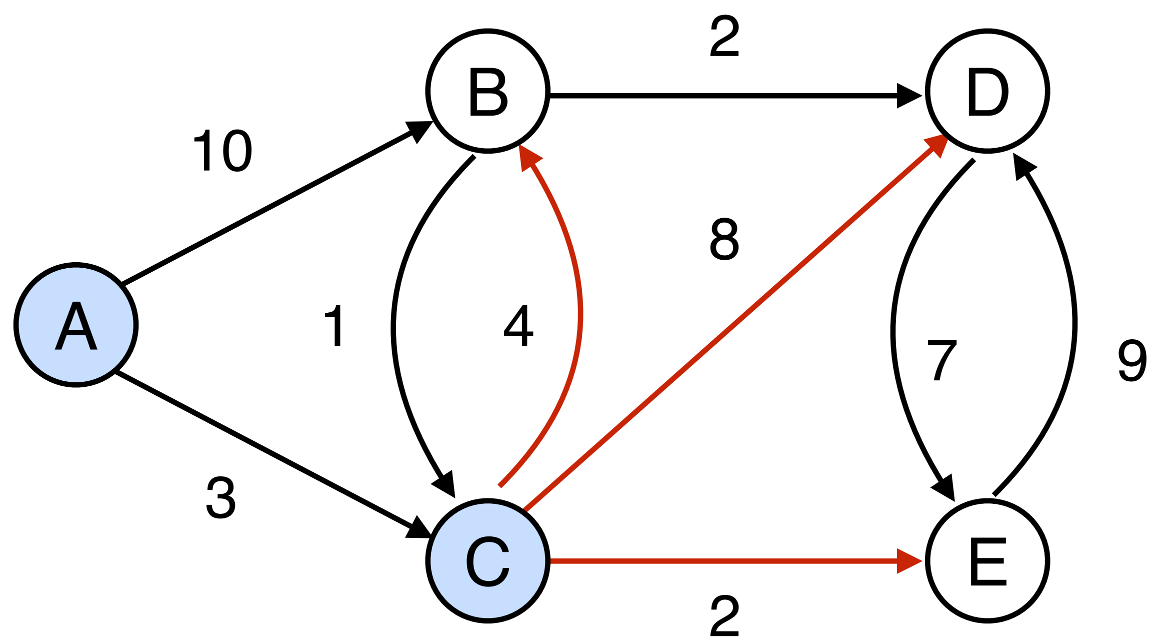
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v,w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	5

D = {d(A) = 0, d(C) = 3}

Dijkstra's algorithm example

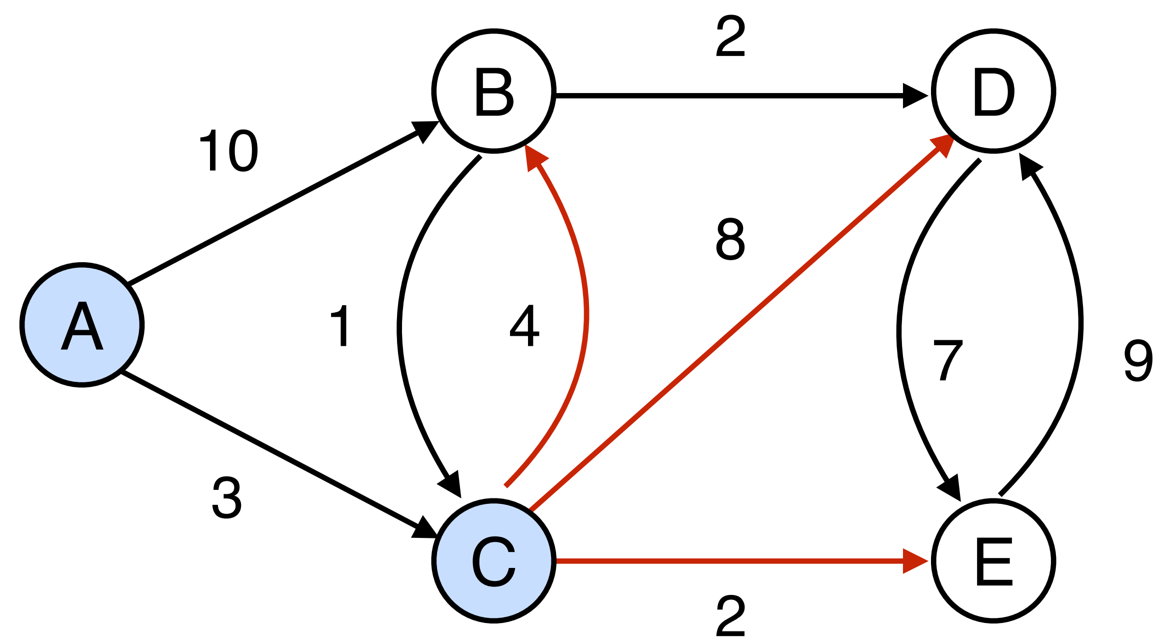
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v,w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

Notice, that the only values that (possible) change are for nodes adjacent to C

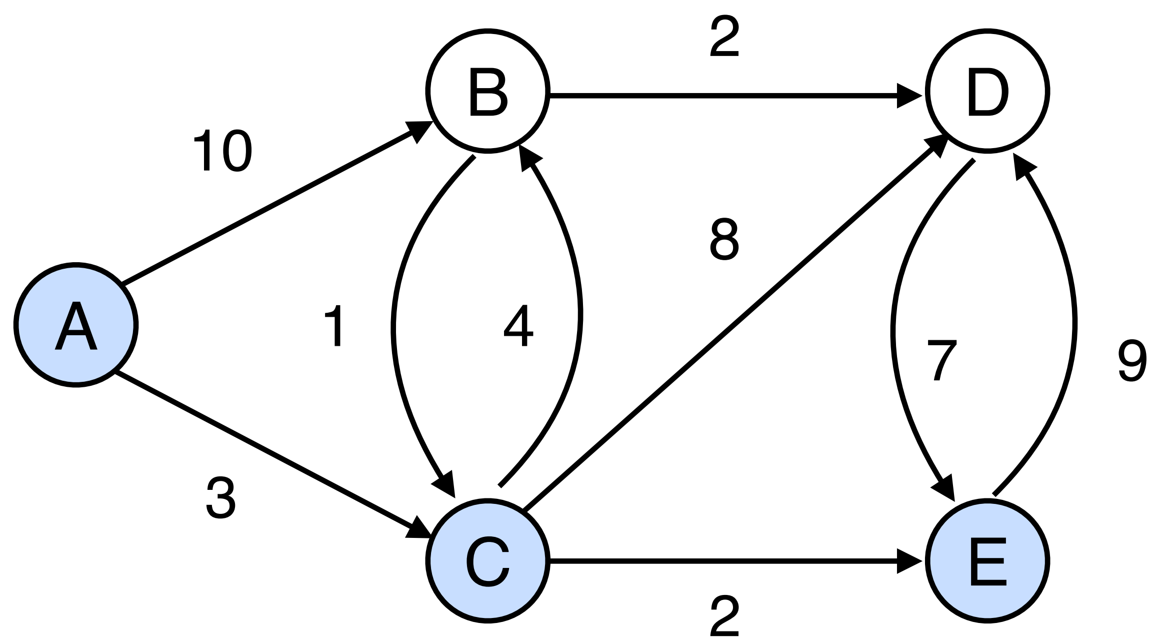
D = {d(A) = 0, d(C) = 3}

Dijkstra's algorithm example

d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?



To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v, w)\right)$$

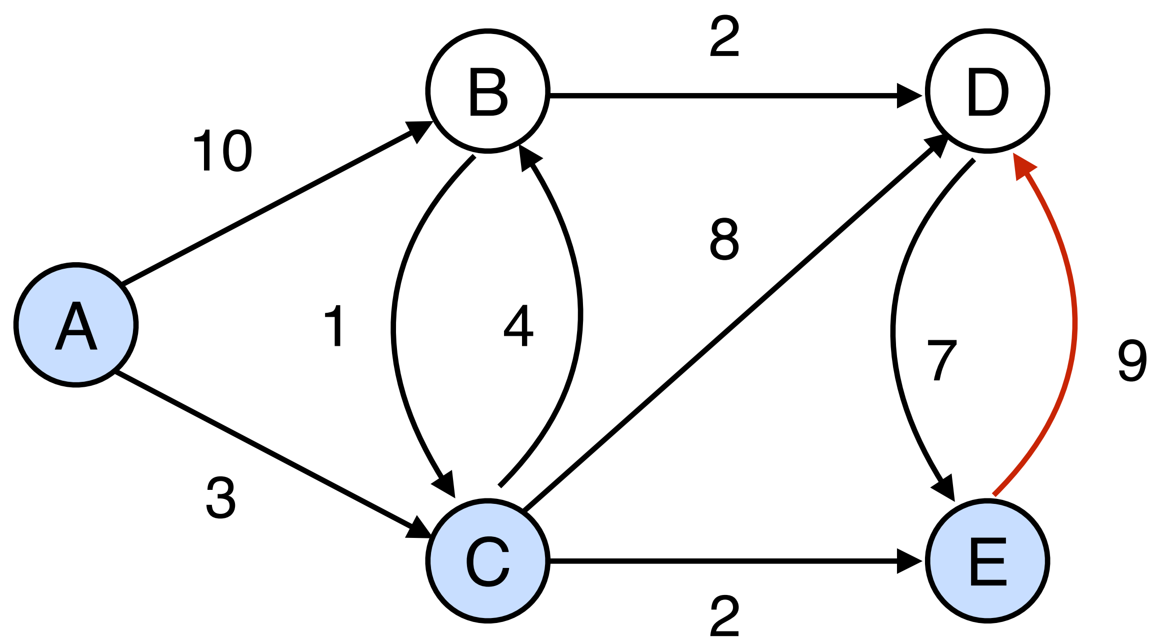
Q	A	B	C	D	E
$\pi(v)$	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$

D = {d(A) = 0, d(C) = 3, d(E) = 5}

Dijkstra's algorithm example

d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v



For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v,w)\right)$$

Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

D = {d(A) = 0, d(C) = 3, d(E) = 5}

Dijkstra's algorithm example

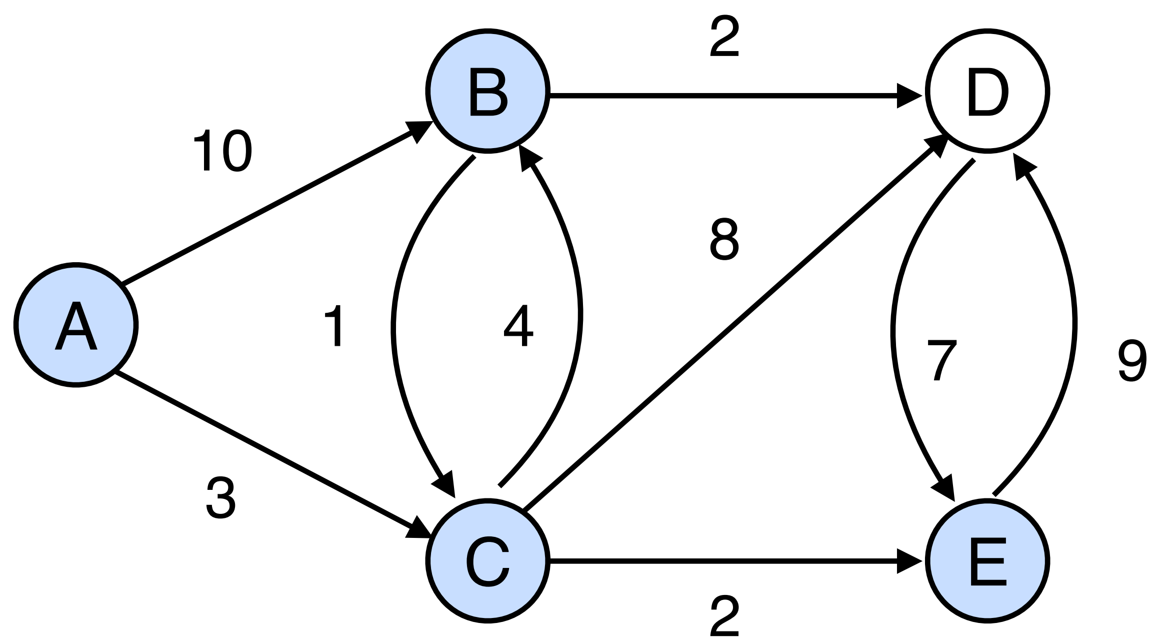
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v,w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

No change for this value!

D = {d(A) = 0, d(C) = 3, d(E) = 5, d(B) = 7}

Dijkstra's algorithm example

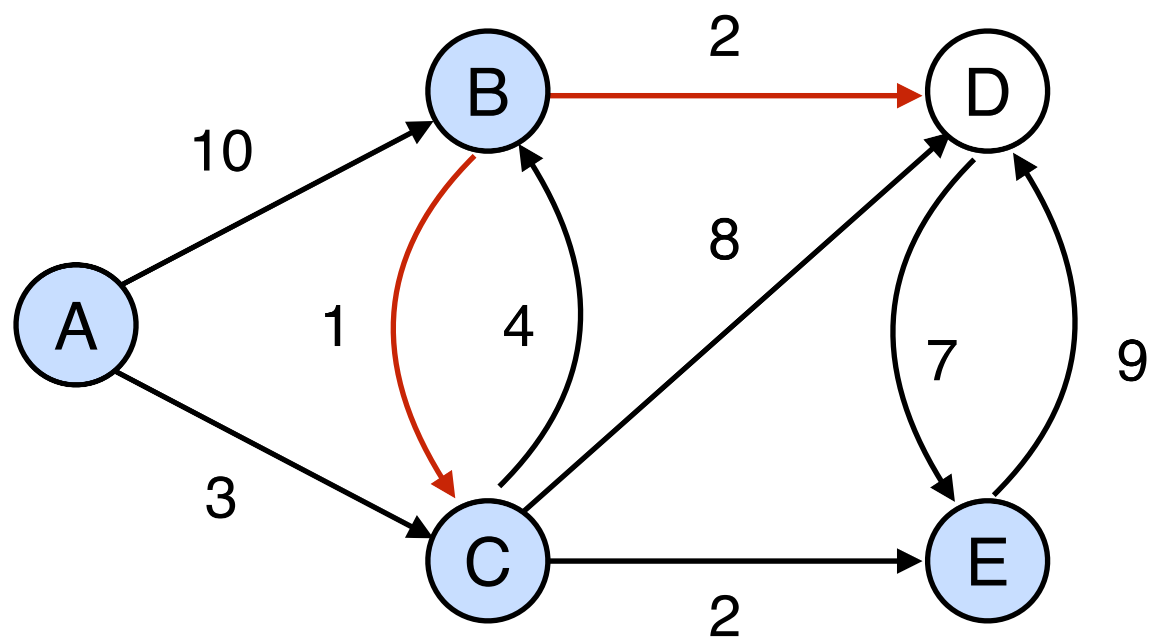
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v,w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

D = {d(A) = 0, d(C) = 3, d(E) = 5, d(B) = 7}

Dijkstra's algorithm example

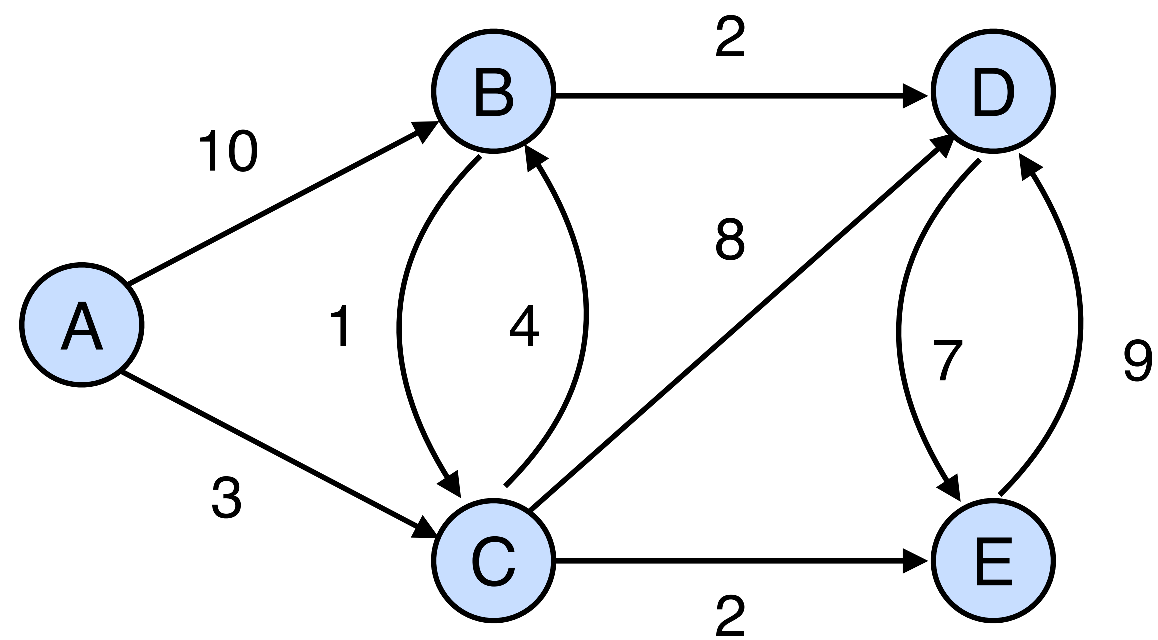
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v,w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

D = {d(A) = 0, d(C) = 3, d(E) = 5, d(B) = 7, D(d) = 9}

Dijkstra's algorithm example

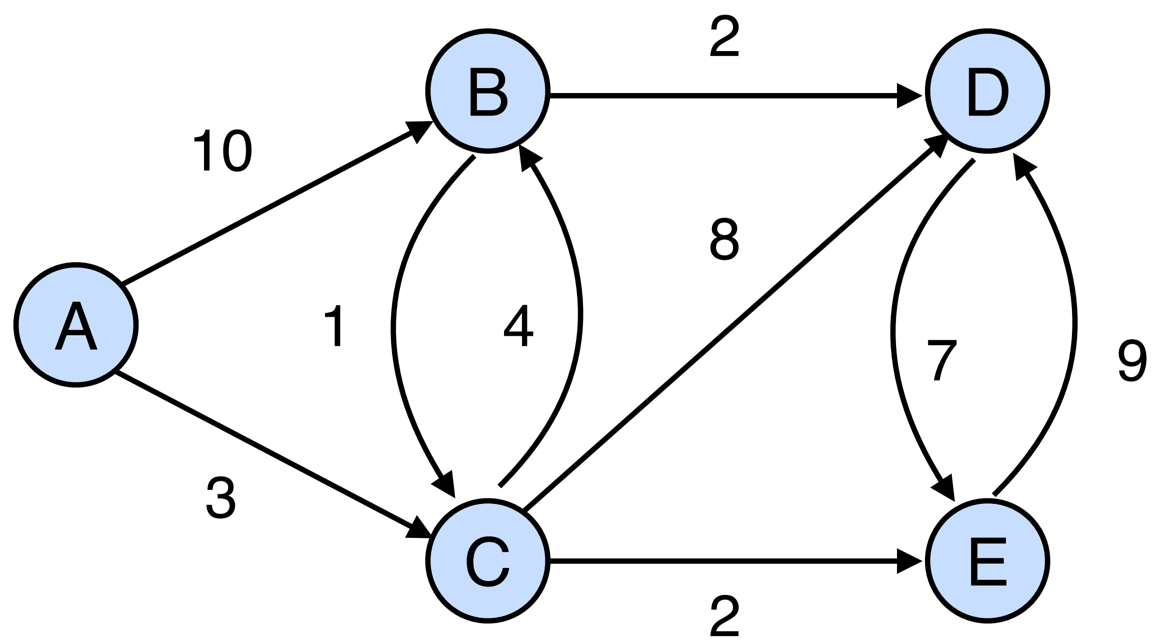
d(u) = distance from s to u

$\pi(v) =$ currently known shortest distance to v

For which nodes can we update their tentative distance $\pi(v)$?

To update compute

$$\pi(w) = \min\left(\pi(w), \pi(v) + l(v,w)\right)$$



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

Values highlighted in blue are the true d(v) distances.

D = {d(A) = 0, d(C) = 3, d(E) = 5, d(B) = 7, D(d) = 9}

Dijkstra's algorithm

Algorithm 1: Dijkstra(G, s)

/* G directed, weighted adjacency list, source s */

1 $\pi \leftarrow \{\ \}$ /* hash table, current best dist for v */

2 $d \leftarrow \{\ \}$ /* hash table, distance of v */

3 $parents \leftarrow \{\ \}$ /* hash table, parents in shortest paths tree */

4 **for** v *in* G **do**

5     $\pi[v] \leftarrow \infty$;

6 $\pi[s] \leftarrow 0$, $parents[s] \leftarrow$ None;

7 **for** $i = 1$ *to* n **do**

8     $u \leftarrow$ unfinished node with min $\pi[u]$;

9     $d[u] \leftarrow \pi[u]$ /* fix distance of u */

10   **for** v *in* $G[u]$ **do**

        /* update the distance of neighbors of u */

11   **if** $\pi[v] > d[u] + G[u][v]$ **then**

12             $\pi[v] \leftarrow d[u] + G[u][v]$;

13             $parents[v] = u$;

14 **return** d, parents

Questions about the implementation:

- Can we be more efficient about updating the distances? (lines 7-13)
- How do we find the minimum u? (line 8)

Priority queue – asymptotic running time of operations

In this course, when we refer to a Priority Queue, we always refer to a *binary heap implementation*. Given that the PQ holds n items the operations take:

O(1) : peaking at the root.
O(log n): INSERT, DECREASE-KEY, EXTRACT-MIN (a.k.a. DELETE-MIN)

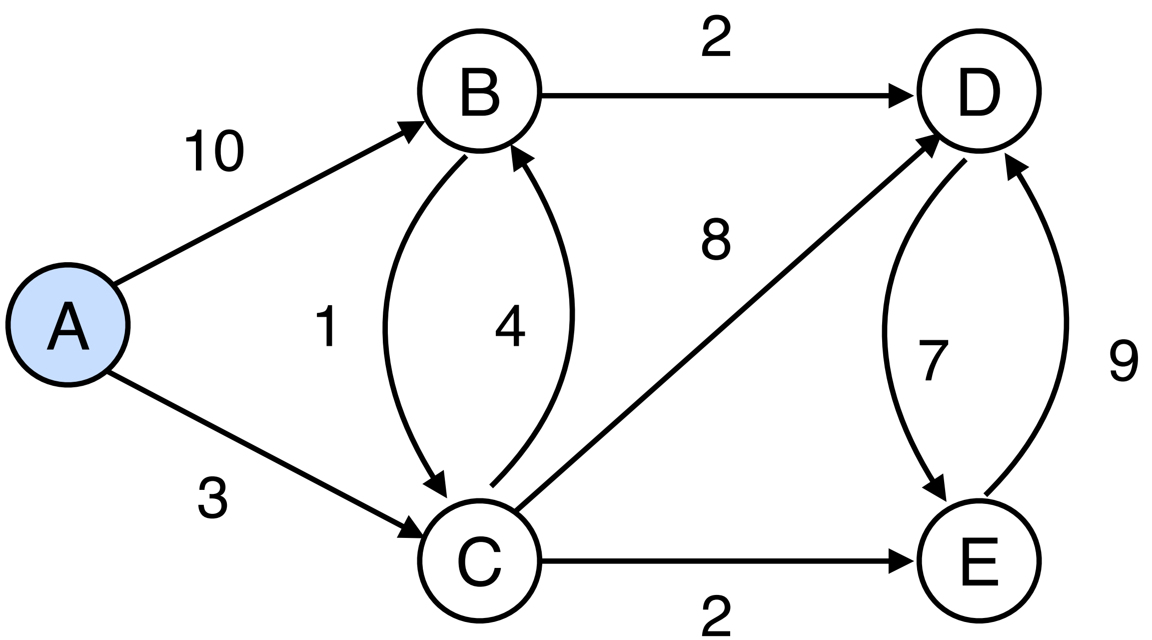
What this means for the running time of Dijkstra's:

- the PQ holds entries corresponding to each node of a graph
- during the algorithm we check and update once for every edge
- total time complexity of these operations is O(m log n).

Dijkstra's algorithm example – priority queue operations

Initialize Q:

$Q.\text{INSERT}\,(<0, A>)$
$Q.\text{INSERT}\,(<\infty, B>)$
$Q.\text{INSERT}\,(<\infty, C>)$
$Q.\text{INSERT}\,(<\infty, D>)$
$Q.\text{INSERT}\,(<\infty, E>)$



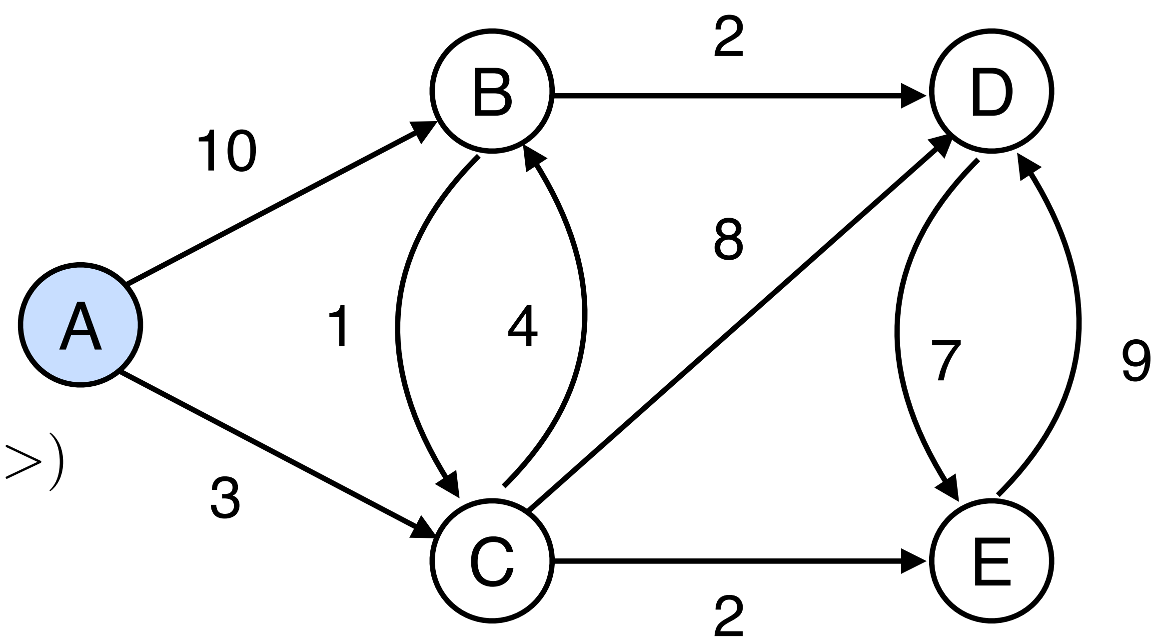
$$<0, A>= Q.\text{EXTRACT\_MIN}()$$

Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞

Dijkstra's algorithm example – priority queue operations

Update distances in Q:

$Q.\text{DECREASE\_KEY}\,(<\infty, B>, <10, B>)$
$Q.\text{DECREASE\_KEY}\,(<\infty, C>, <3, C>)$



Current content of Q:

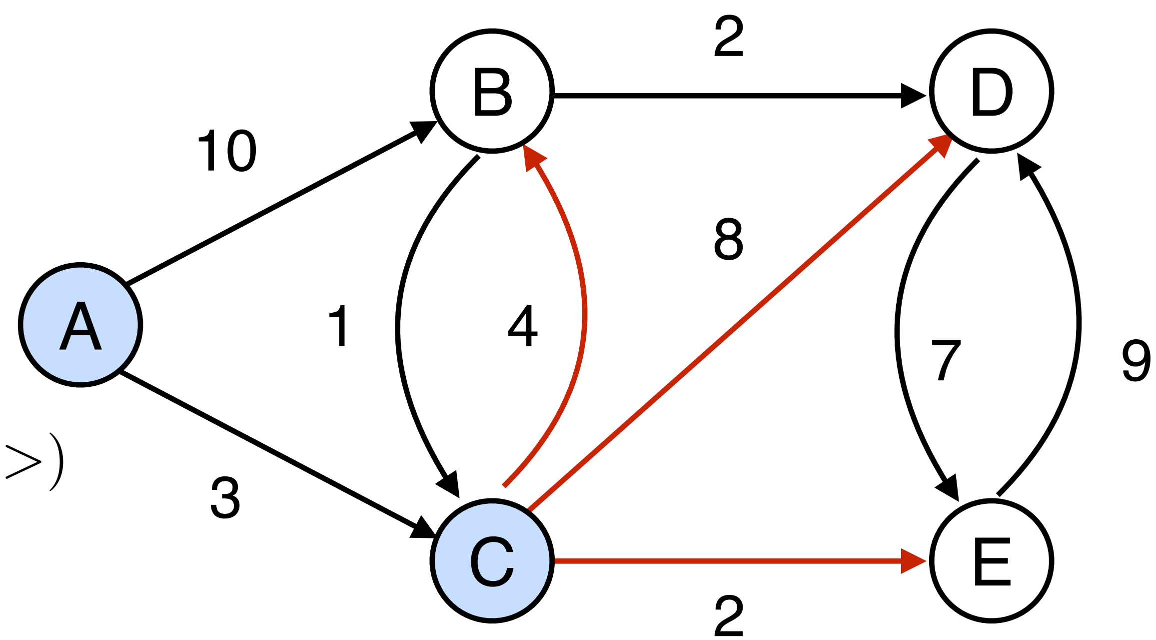
$$Q = (<10, B>, <3, C>, <\infty, D>, <\infty, E>)$$

Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞

Dijkstra's algorithm example – priority queue operations

Current content of Q:

$$Q = (< 10, B >, < 3, C >, < \infty, D >, < \infty, E >)$$



$$< 3, C > = Q.\text{EXTRACT\_MIN}()$$

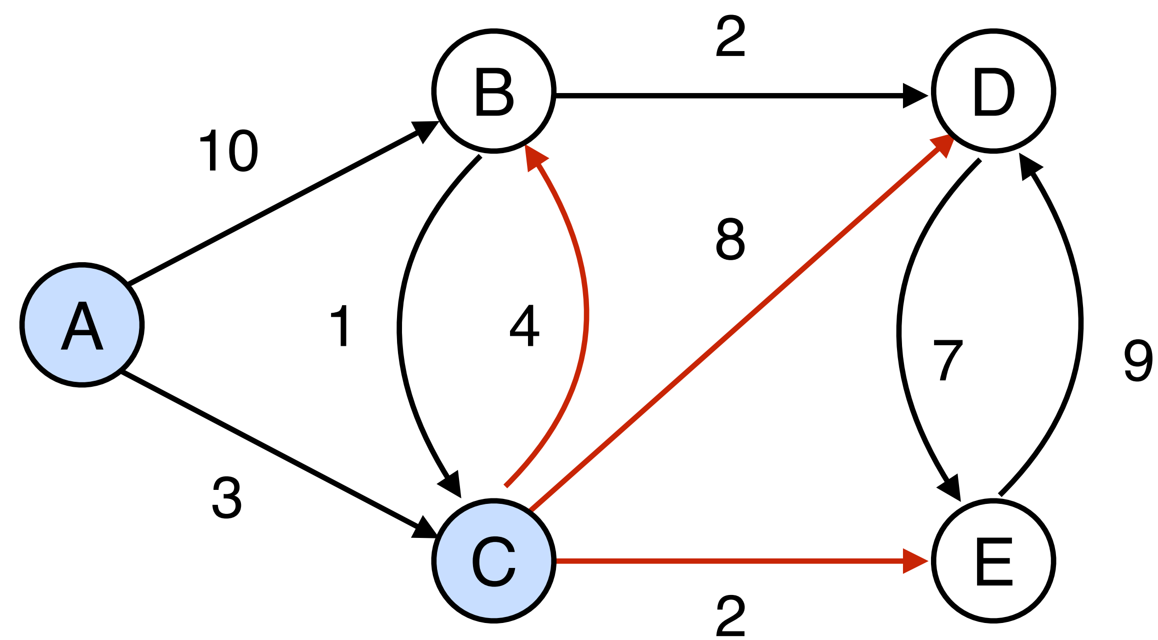
Q	A	B	C	D	E
$\pi(v)$	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞

Dijkstra's algorithm example – priority queue operations

Update distances in Q:

$Q.\mathrm{DECREASE\_KEY}\,(<10, B>, <7, B>)$

same for D and E



Q	A	B	C	D	E
$\pi(v)$	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

Dijkstra's algorithm

Algorithm 1: Dijkstra(G, s)

/* G directed, weighted adjacency list, source s */
$\pi \leftarrow \{\ \}$/* hash table, current best dist for v */
$d \leftarrow \{\ \}$/* hash table, distance of v */
$parents \leftarrow \{\ \}$/* hash table, parents in shortest paths tree */
$Q \leftarrow$ priority queue/* keys are current best distances $\pi[v]$ */
for $v \neq s$ *in* G **do**
    $\pi[v] \leftarrow \infty$;
    Q.INSERT($< \pi[v], v >$);
$\pi[s] \leftarrow 0$, $parents[s] \leftarrow$ None, Q.INSERT($< \pi[s], s >$);
while Q *is not empty* **do**
    $< \pi[u], u > \leftarrow$EXTRACT-MIN(Q);
    $d[u] \leftarrow \pi[u]$/* fix distance of u */
    for v *in* $G[u]$ **do**
        /* update the distance of neighbors of u */
        if $\pi[v] > d[u] + G[u][v]$ **then**
            $\pi[v] \leftarrow d[u] + G[u][v]$;
            $parents[v] = u$;
            DECREASE-KEY($< \pi[v], v >, < d[u] + G[u][v], v >$);
return d, parents

$O(m \log(n))$

at most one call to DECREASE-KEY for each incoming neighbor across iterations.

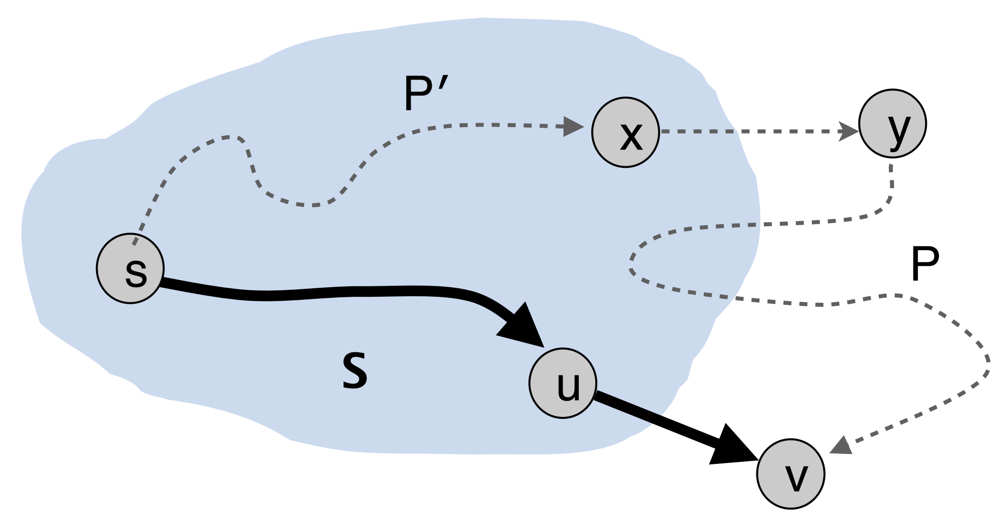
Overall running time O(m log(n)).

You may read online about a running time of O(m + n log(n)). This corresponds to an implementation using Fibonacci heaps (we will use O(m log(n))).

Dijkstra's algorithm: proof of correctness

Invariant. For each node $u \in S$, $d(u)$ is the length of a shortest $s \leadsto u$ path.

(Note: this implies that once u is added to S, d(u) is never changed)



Dijkstra's algorithm: proof of correctness

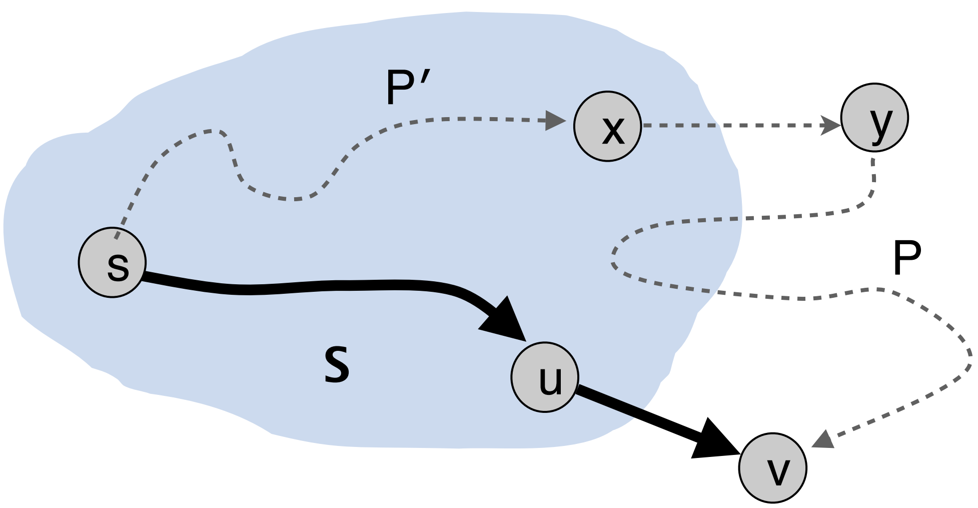
Invariant. For each node $u \in S$, $d(u)$ is the length of a shortest $s \rightsquigarrow u$ path.

Pf. [by induction on $|S|$]

Base case: $|S| = 1$ is easy since $S = \{ s \}$ and $d(s) = 0$.

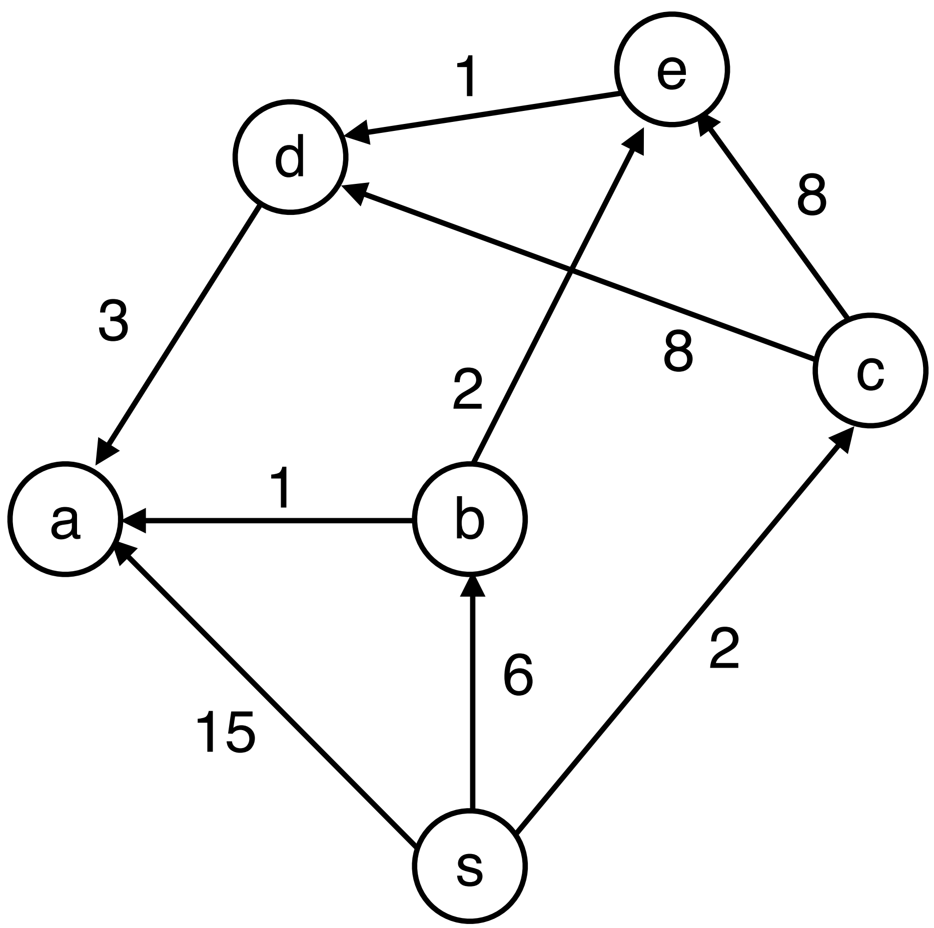
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let (u, v) be the final edge.
- A shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
- Consider any $s \rightsquigarrow v$ path P. We show that it is no shorter than $\pi(v)$.
- Let (x, y) be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it reaches y.

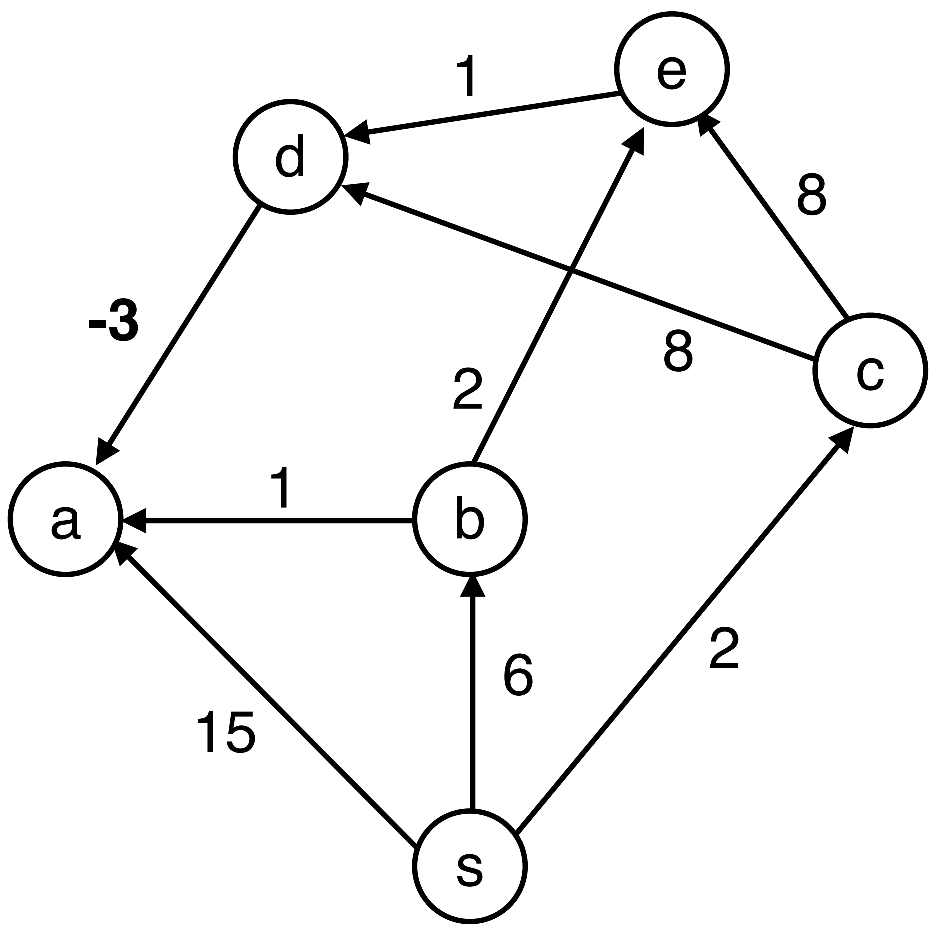


$$\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v) \quad \blacksquare$$

Dijkstra example



Dijkstra example - negative edge



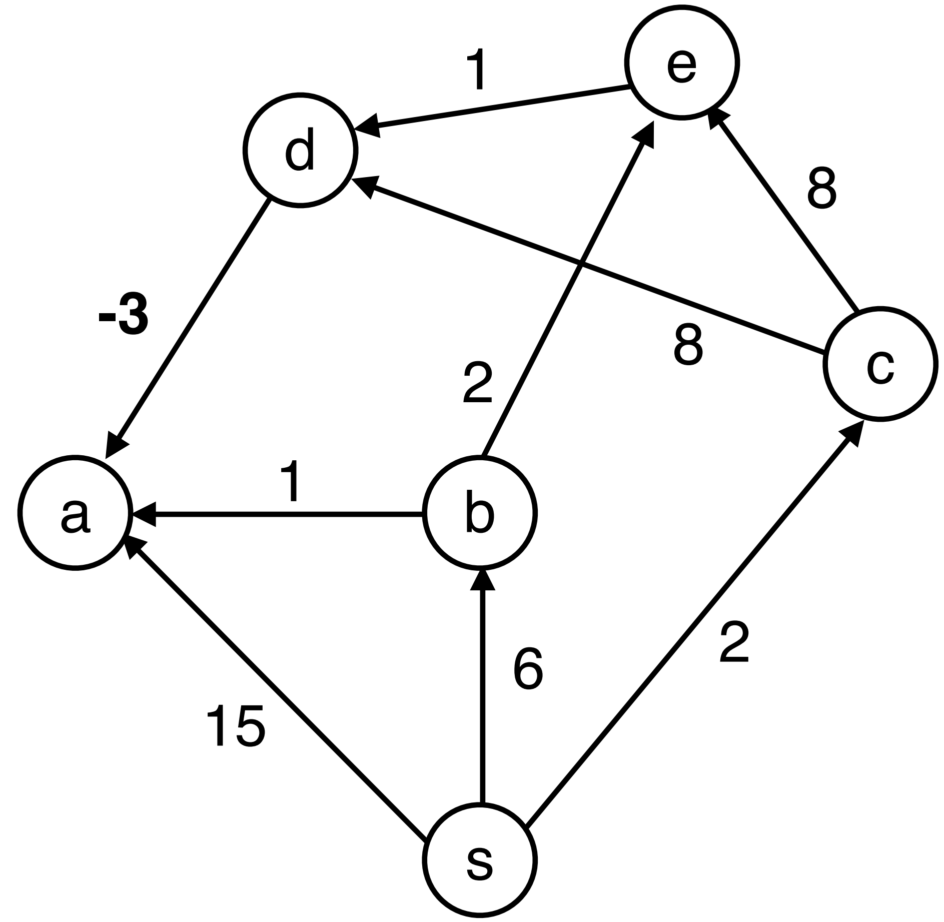
Dijkstra's fails with negative edge weights. What does that mean?

TopHat

Notation:

d[a] = the distance value returned by Dijkstra's

$\ell(a)$ = true shortest path length from s to a



Run Dijkstra's algorithm from node s. What distance value will the algorithm return for node a and what is the correct length of the shortest path from s to a?

A. $d[a] = 15,\ \ell(a) = 7$

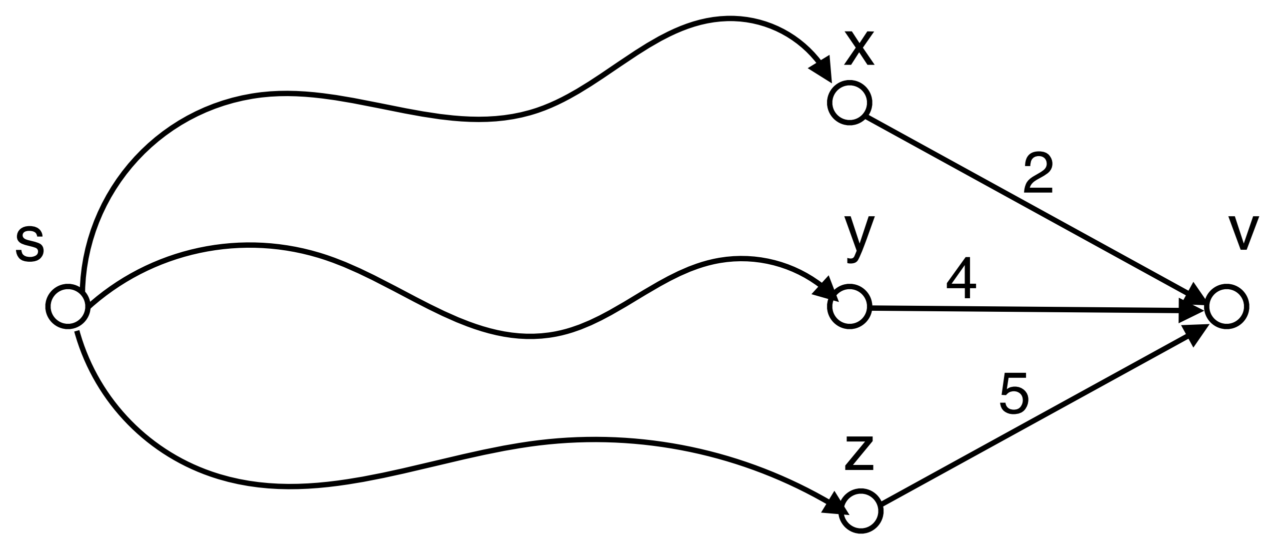
B. $d[a] = 7,\ \ell(a) = 7$

C. $d[a] = 2,\ \ell(a) = 1$

D. $d[a] = 7,\ \ell(a) = 6$

Review

Question. Suppose that node *v* has 3 incoming edges *(x,v)*, *(y,v)* and *(z,v)*. Given the distance from *s* to *x, y, z* and the weights on each edge, what is *dist(s,v)*?



dist(s,x) = 8

dist(s,y) = 4

dist(s,z) = 4

Conclusion: the shortest path length can be computed as the minimum over the in-neighbors of v:

$$dis(s,v) = \min_{u:\ edge\ (u,v)} \{dist(s,u) + \ell(u,v)\}$$

"Greedy" approach: in order for the above to work we need to know for sure that dist(s,u) is correct at the time that we use it for dist of node v.

Dijkstra's algorithm overview

For each node *v* we maintain the min length of path we know so far from *s* to *v*.

- this is the *best known* upper bound on dist(s,v) so far
- denoted by $\pi(v)$

Initialize: for each v $\pi(v) = \infty$

In each iteration:

- find u with the lowest $\pi(u)$
- fix the distance *dist(s,u)* to be $dist(s,u) = \pi(u)$
- for each neighbor v of u, update their best known path

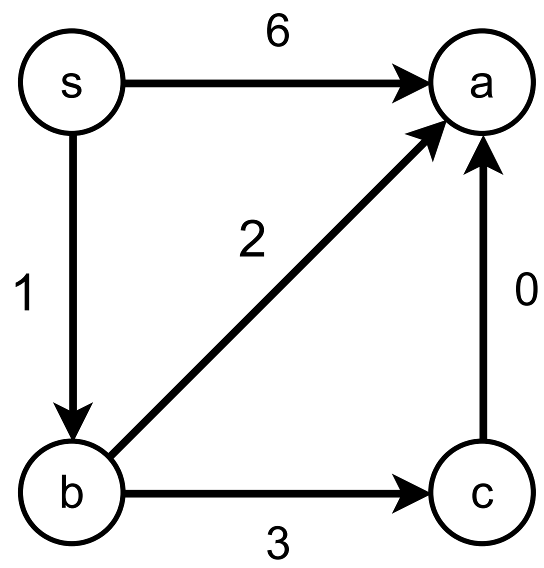
$$\pi(v) = \min\{\pi(v), \mathrm{dist}(s,u) + l(u,v)\}$$

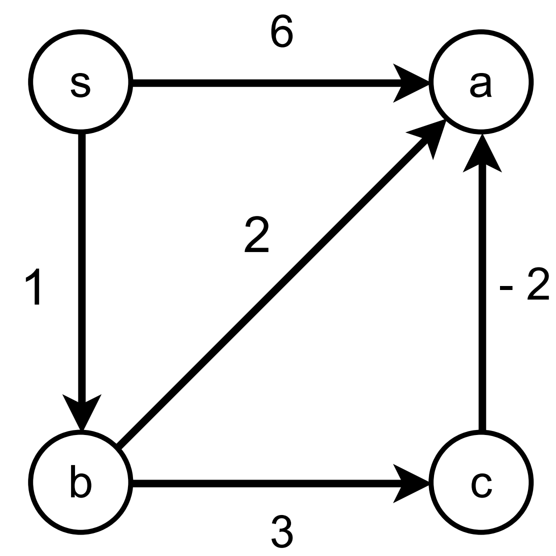
Implementation:

- to compute $\pi(v)$ instead of taking the minimum over *in*-neighbors of v
- we check whether we can update $\pi(v)$ of the *out*-neighbors of u when the distance of a node u gets fixed.

Dijkstra example - negative edge

Dijkstra's fails with negative edge weights. What does that mean?





TopHat

Select the true statements for a directed weighted graph G, with no negative edge weight and source s.

A. It’s possible for a node v to have two shortest paths between s to v.

B. If all edge weights are *unique* then the shortest path to each node is *unique*.

C. If there are two different shortest paths from s to v, then Dijkstra’s always finds the one with fewer edges.

History

Philip II of Macedon (BC359)

- divide et impera = divide and rule
- creating or encouraging divisions among the subjects to prevent alliances that could challenge the sovereign

Macchiavelli - The Art of War (1521)

- divide the enemy army in to two and then conquer each half one at a time

Image source: Wikipedia

Divide-and-conquer paradigm

- Break up problem into several parts
- Solve each part recursively
- Combine solutions to subproblems into overall solution

Examples:

- mergesort, quicksort, binary search
- geometric problems: convex hull, nearest neighbors
- efficient computations: multiplication of numbers, matrix multiplication
- algorithms for processing on trees
- many data structures: binary search trees, heaps
- parallelizations

Sorting

Input: n numbers

output: n numbers in sorted order

brute-force: all possible orders of n numbers $O(n!)=O(n^n)$

bubble sort: if two neighboring numbers are in opposite order then swap. $O(n^2)$

- this is already polynomial

Divide-and-conquer paradigm - Mergesort

Input: n numbers

output: n numbers in sorted order

one of the earliest sorting algorithms (1945, John von Neumann)

7	10	18	14	3	2	16	23	20	11

7	10	18	14	3		2	16	23	20	11	divide O(1)

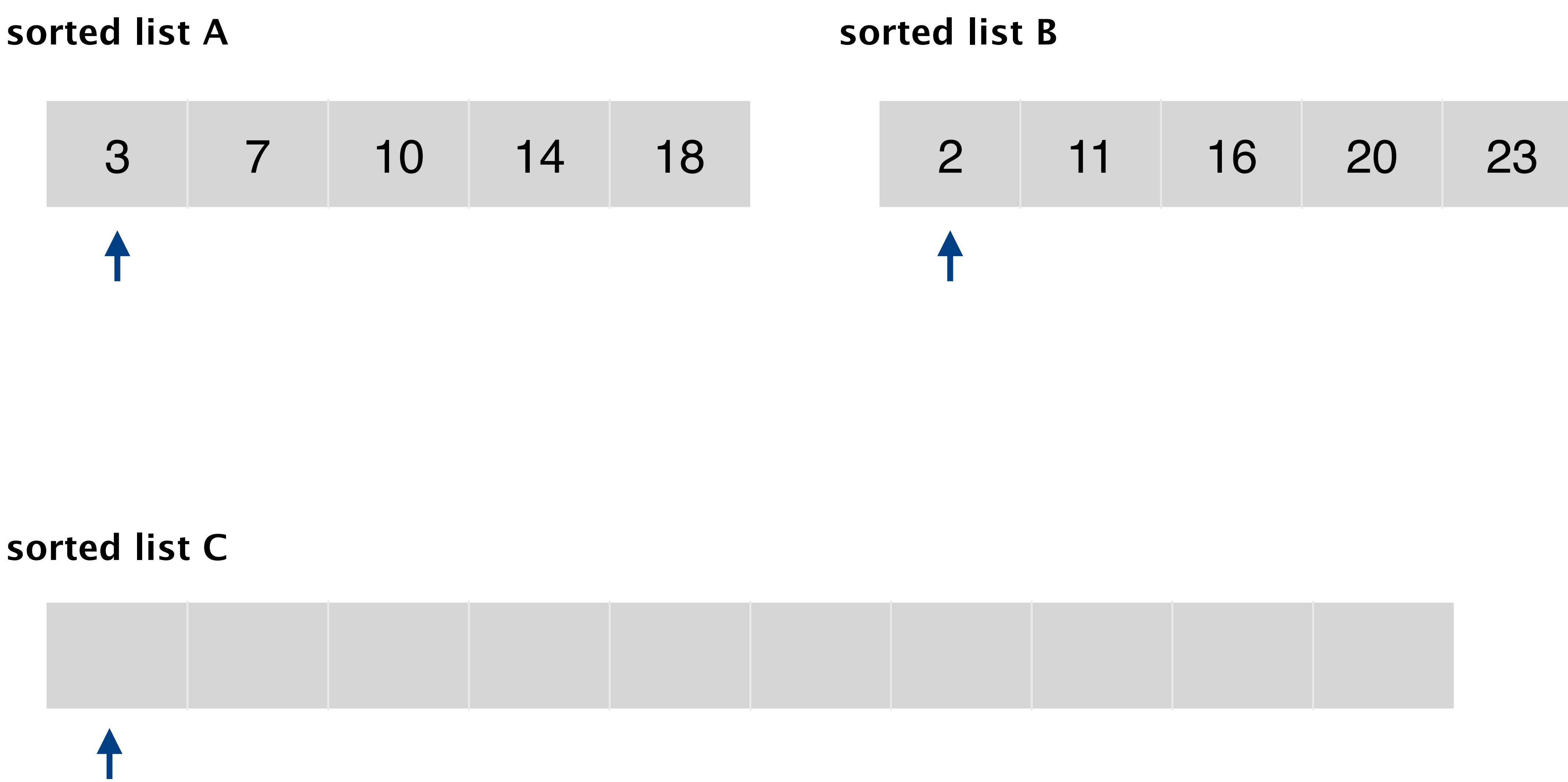
sort by recursive call to algorithm

3	7	10	14	18		2	11	16	20	23	sort 2T(n/2)

2	3	7	10	11	14	16	18	20	23	merge O(n)

Merge in MergeSort

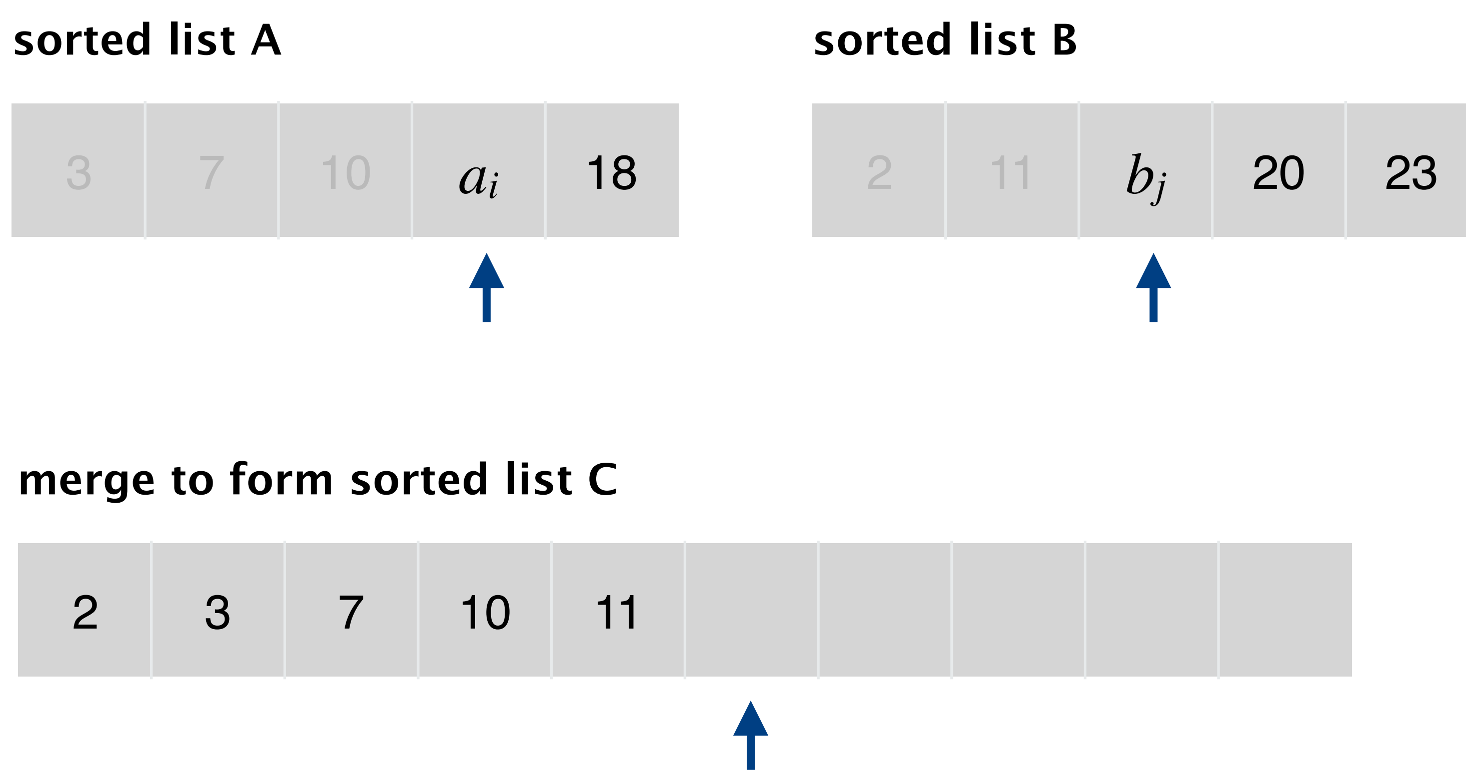
Given two *sorted* lists A and B, merge into sorted list C.



Merge in MergeSort

Goal. Combine two *sorted* lists A and B into a sorted whole C.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i \le b_j$, append a_i to C (no larger than any remaining element in B).
- If $a_i > b_j$, append b_j to C (smaller than every remaining element in A).



Exercise. Write the pseudocode for merge

Analyzing recursive algorithms

- Correctness almost always uses strong induction
  - prove correctness of base case (typically: $n <$ constant)
  - for arbitrary n:
    - assume algorithm performs correctly on all input sizes $k < n$
    - prove the algorithm is correct on input size n

- Time/space complexity often uses recurrence
  - structure of recurrence reflects algorithm

MergeSort correctness

MergeSort correctness

Claim. The array returned by MergeSort is sorted

proof. Induction on the length n of the input array

Base case:

- n=1 sorted. (or n=2, MergeSort correctly puts the smaller of two numbers first)

Inductive hypothesis:

- assume that MergeSorts correctly sorts any array of length k < n

Prove for n:

- MergeSort breaks the problem into two arrays A and B of length n/2 each
- By the inductive assumption MergeSort correctly sorts A and B in the recursive calls
- We need to show that the merge step maintains the sorted order
  - a in A and b in B are the current lowest values in their lists
  - Merge selects a if $a \leq b$.
  - a is less than all numbers in A, as A is sorted
  - a is less than all in B, as b is less than all other elements in B

MergeSort

Algorithm 1: MergeSort(A, p, r)

/* Sorts the subarray $A[p:r]$ in place */

if $p == r$ **then**

    return A

$q \leftarrow \lfloor \frac{p+r}{2} \rfloor$;

$A[p:q] \leftarrow$ MergeSort(A, p, q);

$A[q+1:r] \leftarrow$ MergeSort(A, $q+1$, r);

$A[p,r] \leftarrow$ Merge(A, p, q, r);

return A

Note that Merge is called on two *sorted* lists

- due to recursive call on MergeSort

Recursive, top-down approach

- initial call on array of length *n*
- recursive calls deal with the subarray
- each recursive call is on the left and right half of the input

MergeSort — running time

Algorithm 1: MergeSort(A, p, r)

/* Sorts the subarray $A[p:r]$ in place */

if $p == r$ **then**

  return A

$q \leftarrow \lfloor \frac{p+r}{2} \rfloor$;

$A[p:q] \leftarrow \text{MergeSort}(A,\ p,\ q)$;

$A[q+1:r] \leftarrow \text{MergeSort}(A,\ q+1,\ r)$;

Recursive calls — how many?

$A[p,r] \leftarrow \text{Merge}(A,\ p,\ q,\ r)$;

O(r-p)

return A

Recurrence

Def. $T(n)$ = worst case running time on an input of size n

Recurrence: T(n) expressed using a *recursive* function

Mergesort:

1. divide array into two halves
2. recursive calls to mergesort on both halves
3. merge

$$T(n) = \begin{cases} \Theta(1) \text{ if } n == 1 \\ 2T(n/2) + \Theta(n) \text{ if } n > 1 \end{cases}$$

(Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ but it doesn't matter asymptotically —> we often assume that n is a power of 2)

(often we omit the base case as for const n the running time is const)

MergeSort — recurrence

Initial call on *p = 0* and *r = n-1*

Algorithm 1: MergeSort(A, p, r)

```
/* Sorts the subarray A[p : r] in place                */
if p == r then
    return A
q ← ⌊(p+r)/2⌋;
A[p : q] ← MergeSort(A, p, q);
A[q + 1 : r] ← MergeSort(A, q + 1, r);
A[p, r] ← Merge(A, p, q, r);
return A
```

length of subarray is n/2

O(n) for p=0, r=n-1

$$T(n) = 2 \cdot T(n/2) + \Theta(n)$$

running time on input array of length n

running time on input array of length n/2, called on both halves

running time of non-recursive part

Write the recurrences

(You may assume that n is always a power of 2 or 3 or whatever is needed)

Some algorithm takes as input an array of n elements. It divides the array into 3 equal parts and calls itself recursively on all 3 parts. Then it performs O(n) additional computational steps.

T(n) =

Some other algorithm takes as input an array of n elements. It divides itself into 2 equal parts and calls itself recursively on one part. It then performs constant many additional operations.

T(n) =

TopHat – write recurrence

Question. Here is a hypothetical algorithm. What is the corresponding recurrence?

An algorithm takes as input *n* items. After performing *n/2* comparison, it divides the data in to *three equal parts*. Calls itself *recursively on two* of the parts.

A. $T(n) = 2 \cdot T(n/2) + n/2$

B. $T(n) = 3 \cdot T(n/2) + \Theta(1)$

C. $T(n) = 2 \cdot T(n/3) + \Theta(n)$

D. $T(n) = 3 \cdot T(n/3) + \Theta(1)$

TopHat – write recurrence 2

Question. Here is a hypothetical algorithm. What is the corresponding recurrence?

An algorithm takes as input *n* items. It divides the data into *four parts*, *two* of size n/3 the other *two* of size n/6. It makes recursive calls on all parts and finally performs one more operation.

A. $T(n) = 4T\left(\frac{n}{3}\right) + \Theta(1)$

B. $T(n) = 4T\left(\frac{n}{6}\right) + \Theta(1)$

C. $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{6}\right) + \Theta(1)$

D. $T(n) = 2T\left(\frac{n}{3}\right) + 2T\left(\frac{n}{6}\right) + \Theta(1)$

writing recurrences

Write the recurrences (no need to solve) for the following problems:

1. An algorithm takes as input n items. After performing 1 comparison, it divides the data in to two equal parts. Calls itself recursively on only one of the parts.

2. An algorithm takes as input n items. It divides the input in to 3 equal parts, makes a recursive call on each part. Then combines it in O(n) time.

3. An algorithm takes as input n items. It divides the data in to parts of size n/3, n/6, 3n/6 and makes recursive calls to it. It combines the results in O(log n) time.

writing recurrences

Write the recurrences (no need to solve) for the following problems:

1. An algorithm takes as input n items. After performing 1 comparison, it divides the data in to two equal parts. Calls itself recursively on only one of the parts.

$T(n) = T(n/2) + O(1)$

2. An algorithm takes as input n items. It divides the input in to 3 equal parts, makes a recursive call on each part. Then combines it in O(n) time.

$T(n) = 3T(n/3) + O(n)$

3. An algorithm takes as input n items. It divides the data in to parts of size n/3, n/6, 3n/6 and makes recursive calls to it. It combines the results in O(log n) time.

$T(n) = T(n/3)+T(n/6)+T(n/2)+O(\log n)$

MergeSort — recurrence

Def. $T(n)$ = worst case running time on an input of size n

Mergesort recurrence.

$$T(n) = \begin{cases} \Theta(1) \text{ if } n == 1 \\ 2T(n/2) + \Theta(n) \text{ if } n > 1 \end{cases}$$

Closed form formula.

- we don't know how to interpret the running time from the recursive formula
- we need to find a formula that is a mathematical function of n with no recursive function calls
- solution for mergesort: $T(n)$ is $\Theta(n \log n)$

Solving a recurrence.

- find the formula for T(n)
    - multiple methods
- prove by induction that it works

Recursion tree method

- write out tree of recursive calls
- each node gets assigned the work done during that call to the procedure (dividing and combining)
- total work is the sum of work done at all nodes