CS 170
Efficient Algorithms and Intractable Problems

Lecture 8:
Paths in Graphs

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

HW4 and Disc 4 coming out today!

Changes to Office Hours:
→ Removed some Tuesday office hours (low attendance)
→ Instead increasing more office hours and TA presence on other days.

Where are annotated version of Lectures 6-7?
→ My iPad didn't save them, it seems. Sorry ...
→ Refer to the video and the blank slides to fill them in.
→ It's a good exercise!

Annotation Test!

Last two lectures

Exploring graphs via Depth First Search (DFS)

Use cases of DFS:

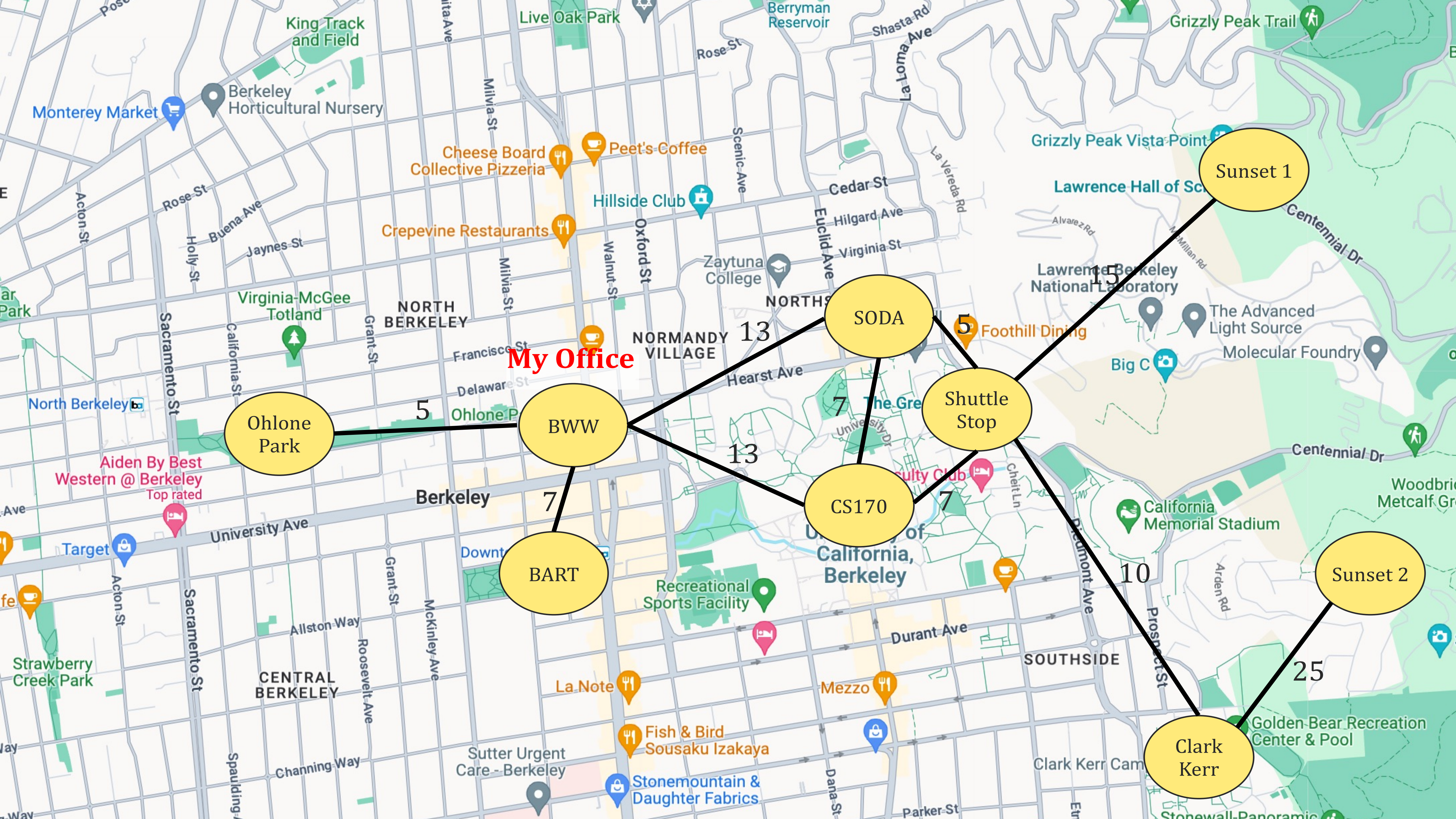
- Topological Sort
- Strongly connected component

Today

Another approach to exploring graphs!

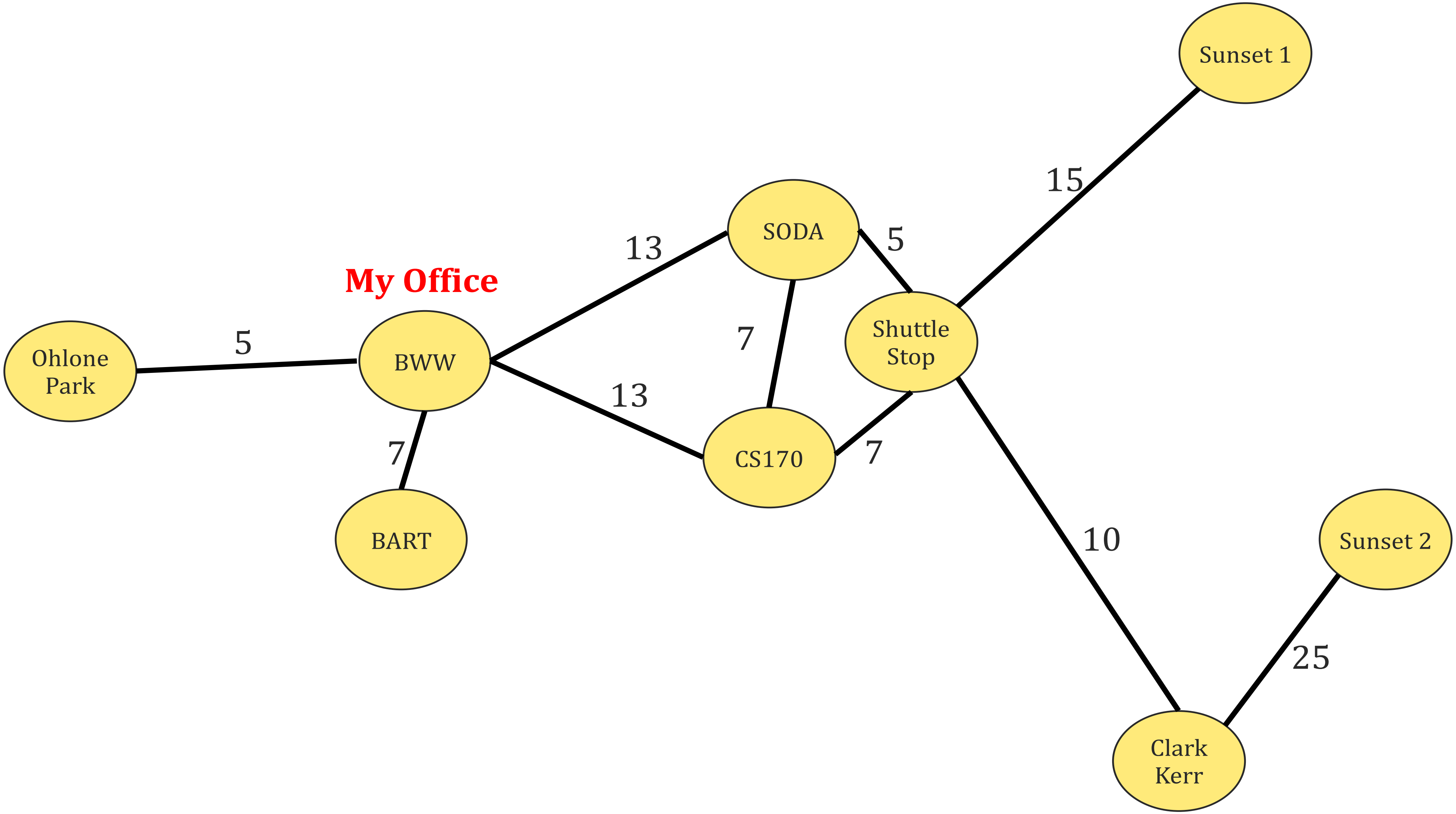
Breadth-First Search and related algorithms.

Finding single-source shortest path.



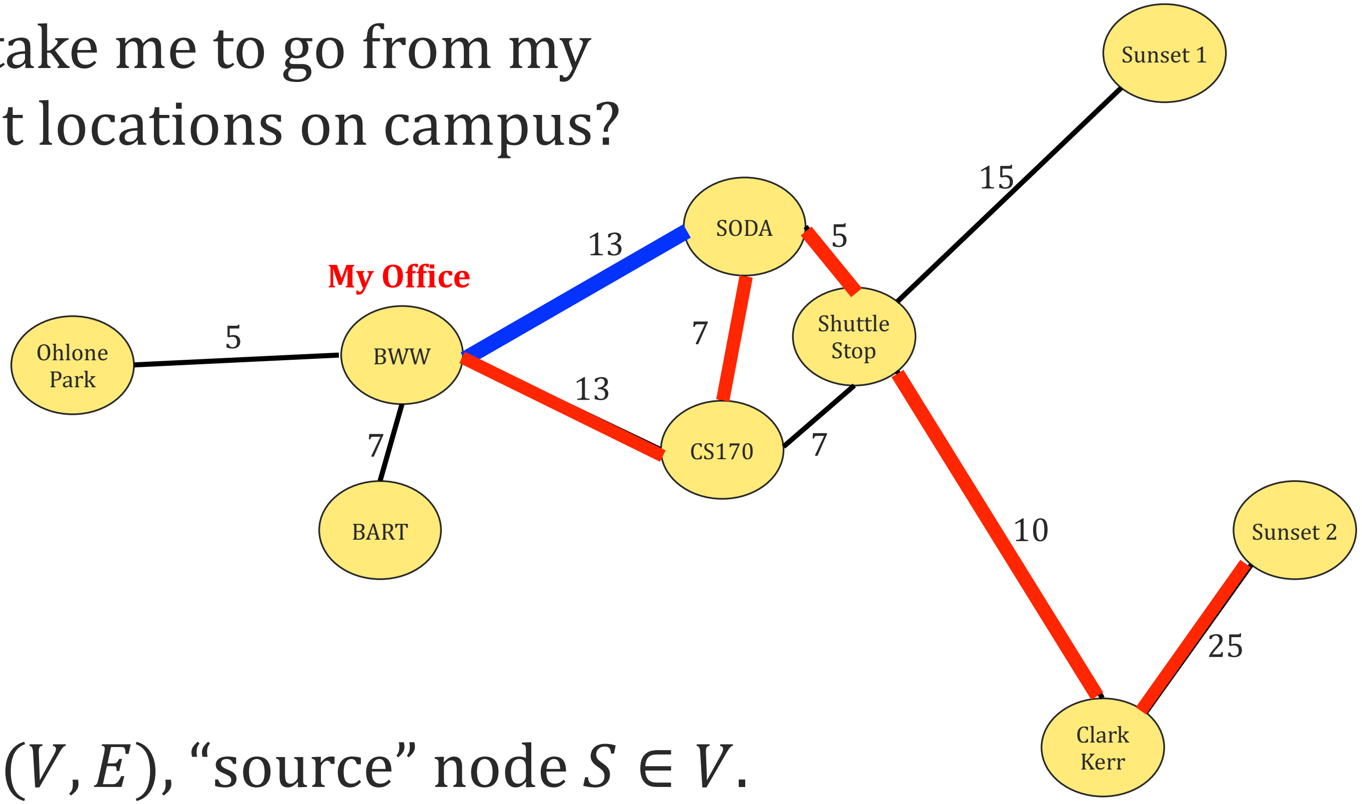
My Office
Ohlone Park
BWW
BART
SODA
CS170
Shuttle Stop
Sunset 1
Sunset 2
Clark Kerr
5
7
13
13
7
5
7
15
10
25

Just the graph



Single-Source Shortest Paths (SSSP)

How long does it take me to go from my office to important locations on campus?



<u>Input:</u> Graph $G = (V, E)$, "source" node $S \in V$.
<u>Output:</u> For all $u \in V$, $dist(s, u) =$ length of shortest path from s to u.

Why not DFS?

- DFS goes depth first. Might explore much farther nodes first.

Exploring for Shortest Path

Depth-First Search:

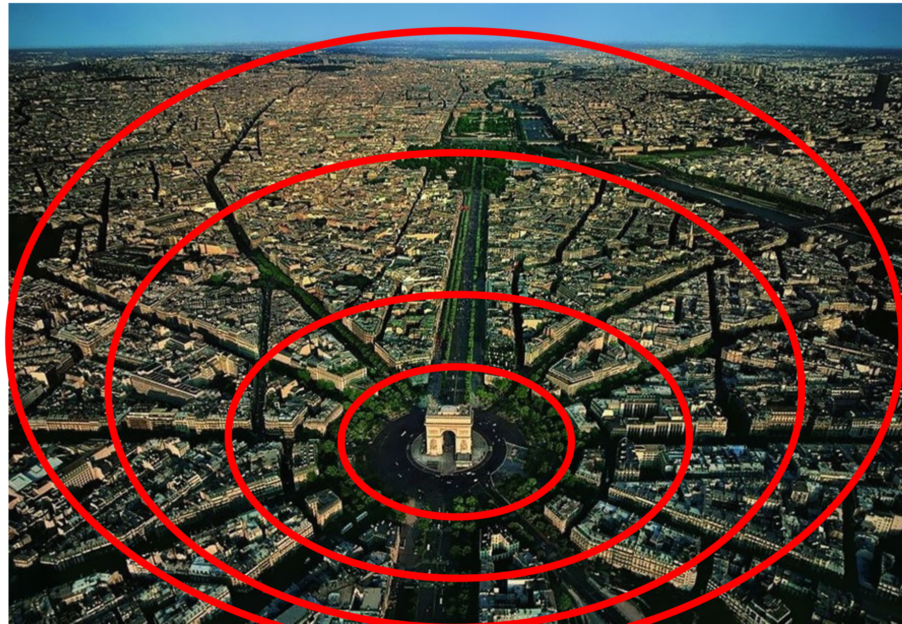
→explore a maze with a chalk and a string.

Breadth-First Search:

→explore a neighborhood from bird's eye perspective.

1. Explore direct neighbors
- everything at distance 1
2. Explore (unseen) neighbors of neighbors.
- everything at distance 2
3. Explore (unseen) neighbors of neighbors of neighbors
- Everything at distance 3

…



Single-Source Shorted Paths Algorithms

<u>Input:</u> Graph $G = (V, E)$, "source" node $S \in V$.
<u>Output:</u> For all $u \in V$, $dist(s, u) =$ length of shortest path from s to u.

Unweighted: All edges length 1
→ Breadth-First Search

Positive Weight: length function $\ell: E \rightarrow \{1, 2, \ldots\}$
→ Dijkstra's Algorithm

Arbitrary lengths: ℓ could be negative too
→ Bellman-Ford algorithm

More general

Less efficient

Breadth-First Search

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue(Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

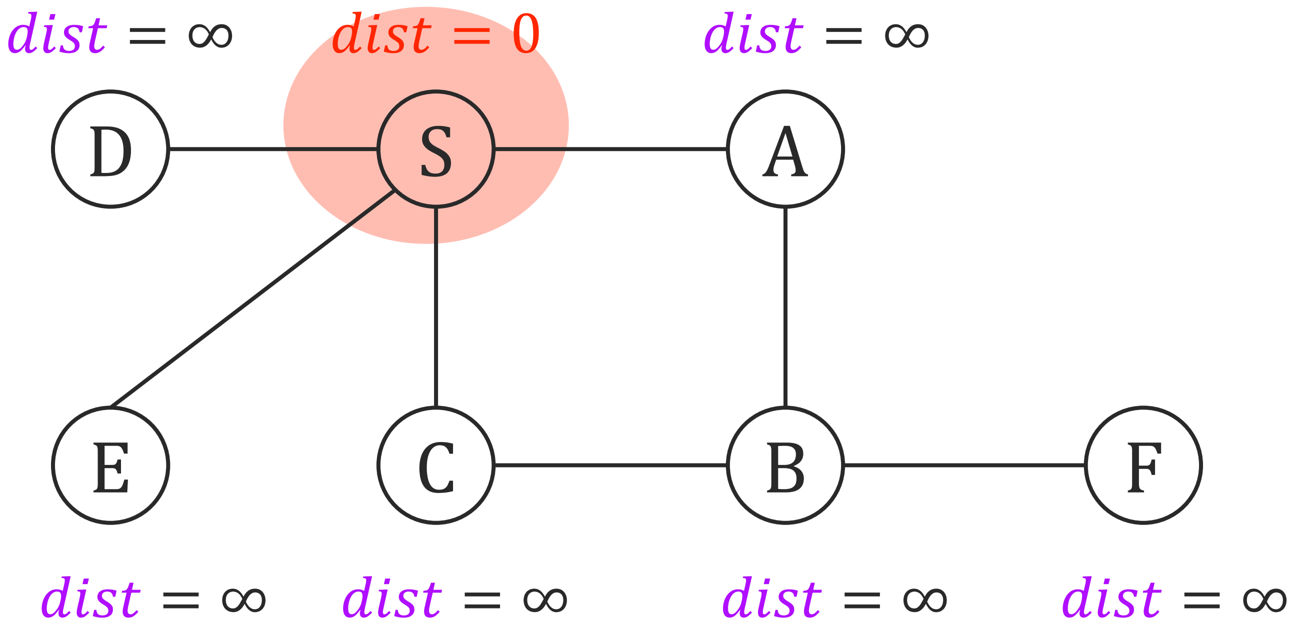
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{S\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

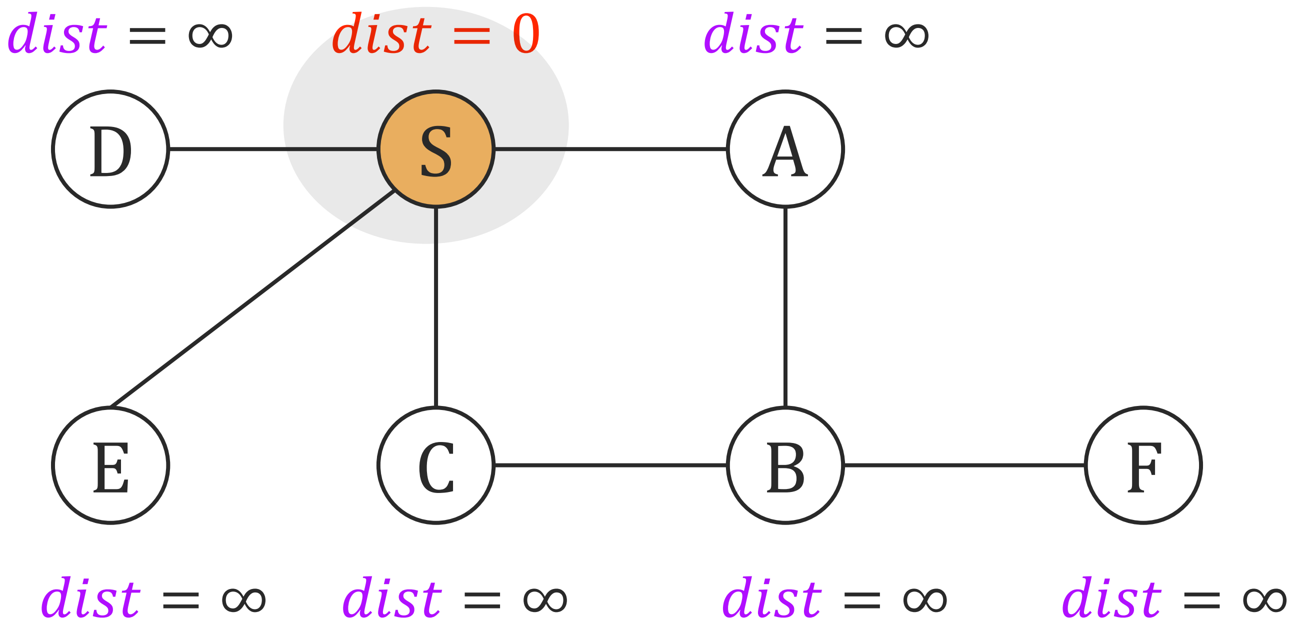
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

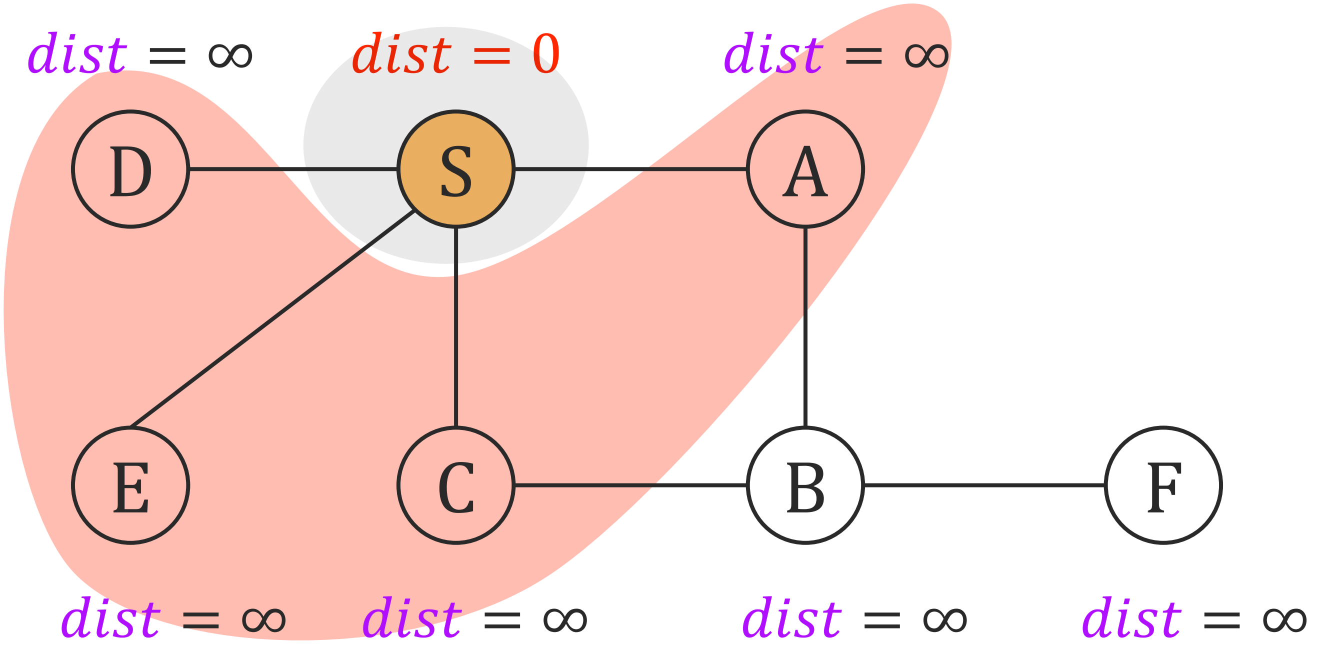
if $dist[v] = \infty$

$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node
Done, with that iteration of "While"



$Q = \{\cancel{S}$, A, C, D, E$\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue(Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

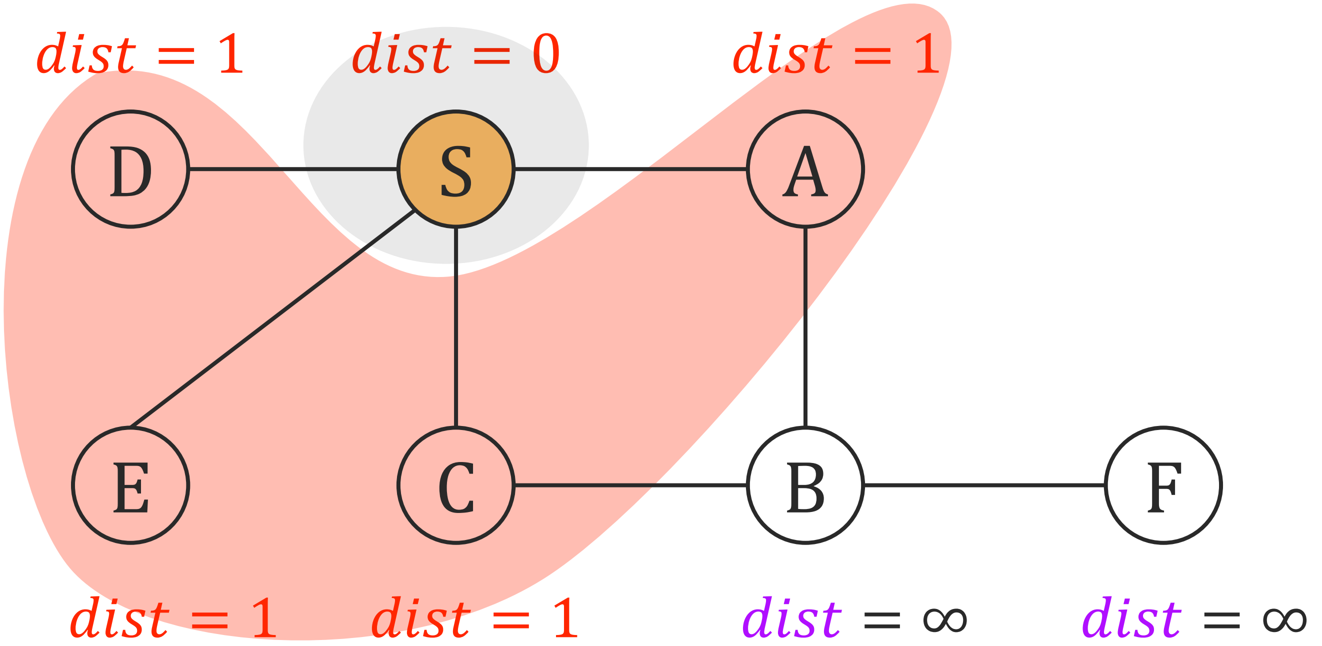
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}$, A, C, D, E$\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

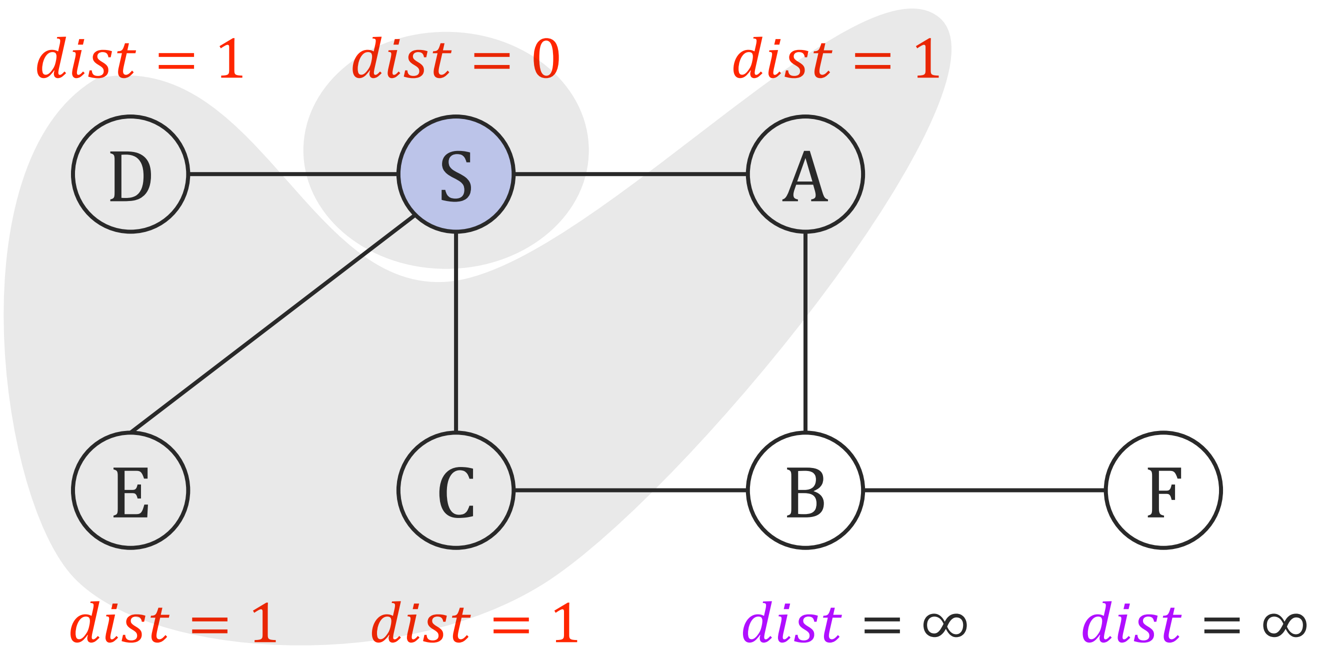
if $dist[v] = \infty$

$$\boxed{\begin{array}{l} enqueue(Q, v) \\ dist[v] = dist[u] + 1 \end{array}}$$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \text{A, C, D, E}\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

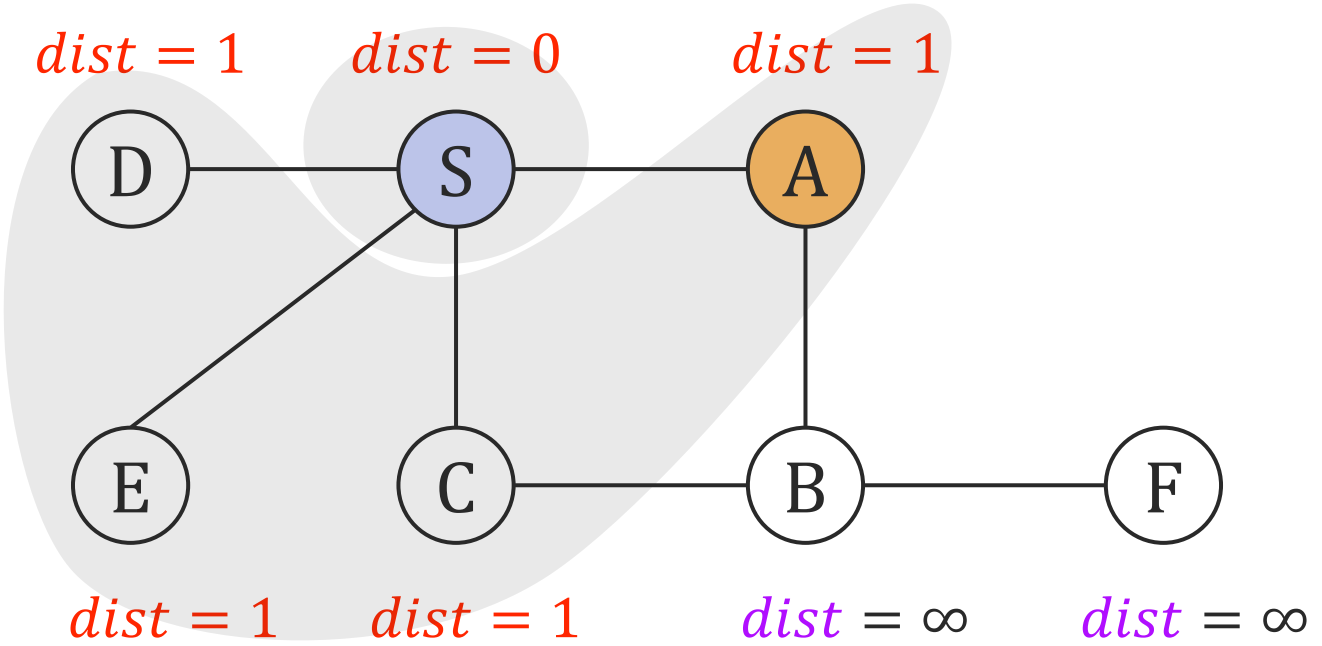
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, C, D, E\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue(Q)$

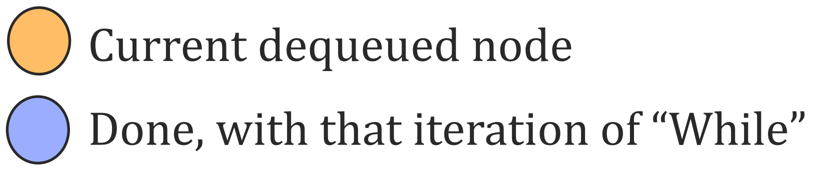
for all v, s.t. $(u, v) \in E$

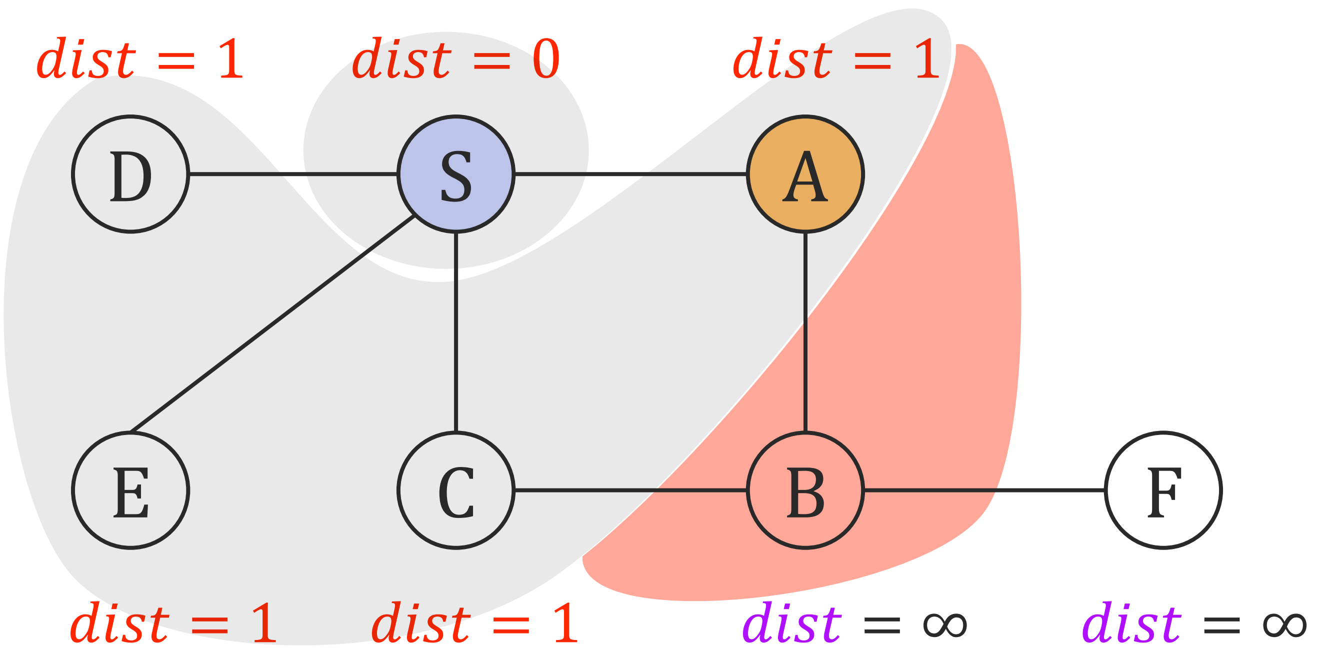
if $dist[v] = \infty$

$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search





$Q = \{\cancel{S}, \cancel{A}, \text{C, D, E, B}\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

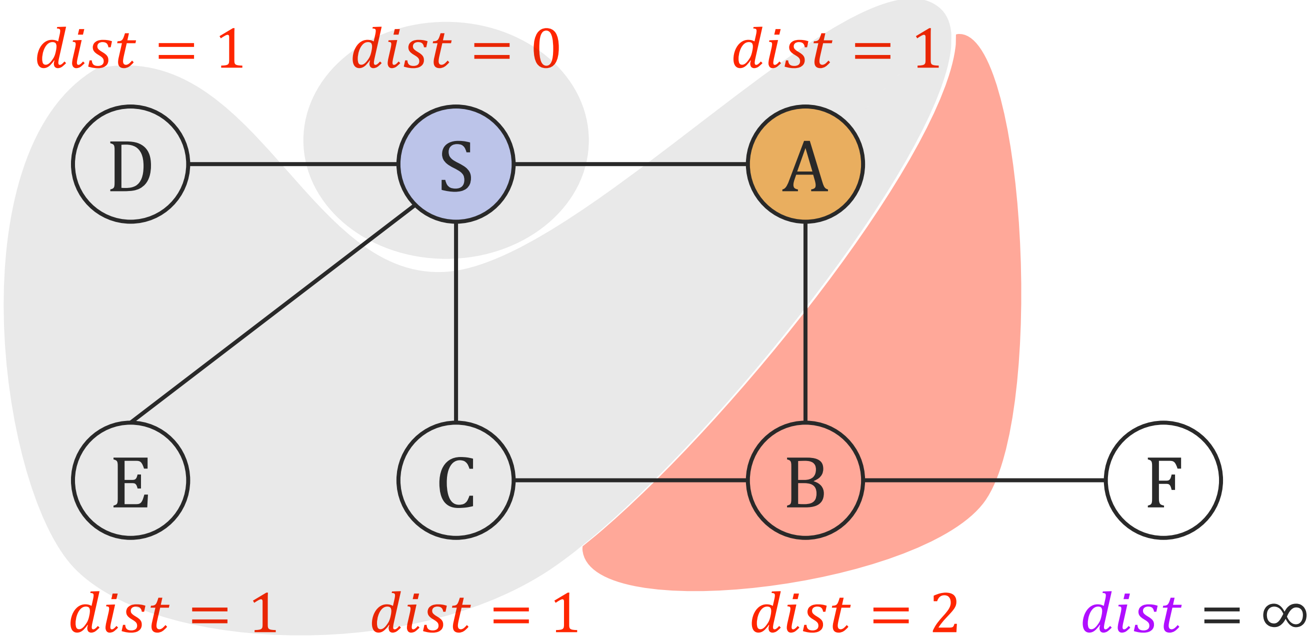
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$bfs(G, s)$

    int array $dist(n)$ // initialize to all ∞

    $dist[s] = 0$

    $Q = \{s\}$ //A queue containing s

    While Q is not empty

        $u = dequeue(Q)$

        for all v, s.t. $(u, v) \in E$

            if $dist[v] = \infty$

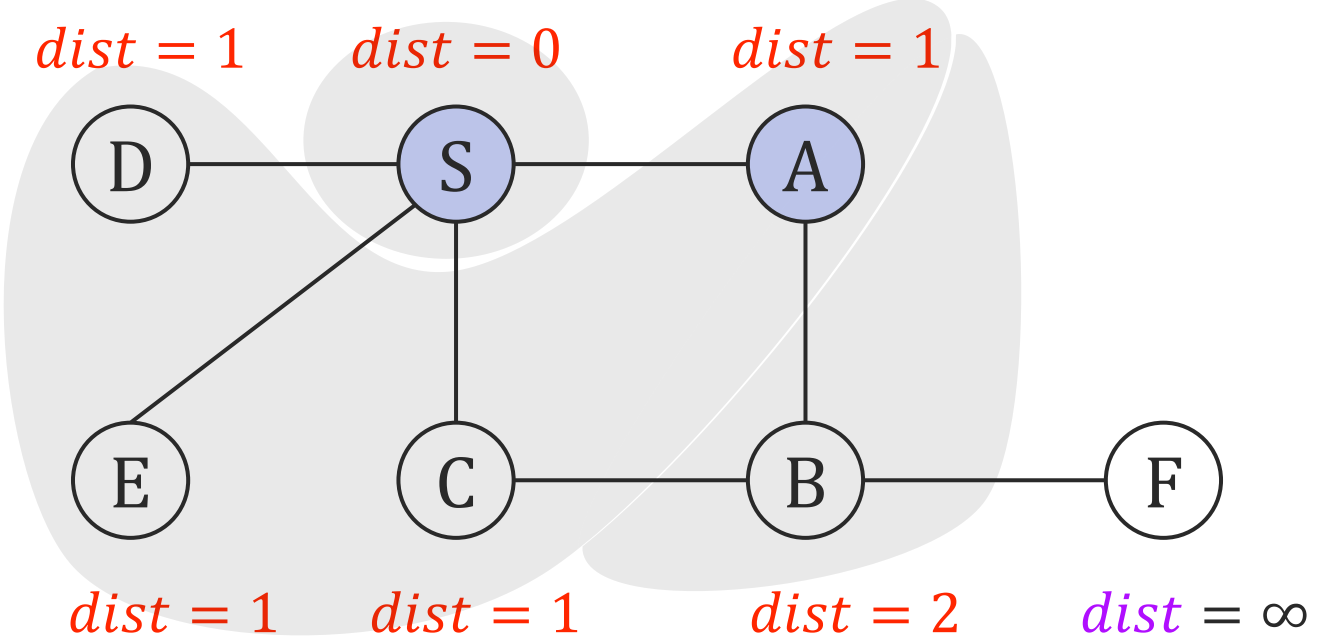
                $\boxed{\begin{array}{l} enqueue(Q, v) \\ dist[v] = dist[u] + 1 \end{array}}$

$Q = \{\cancel{S}, \cancel{A}, C, D, E, B\}$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, C, D, E, B\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

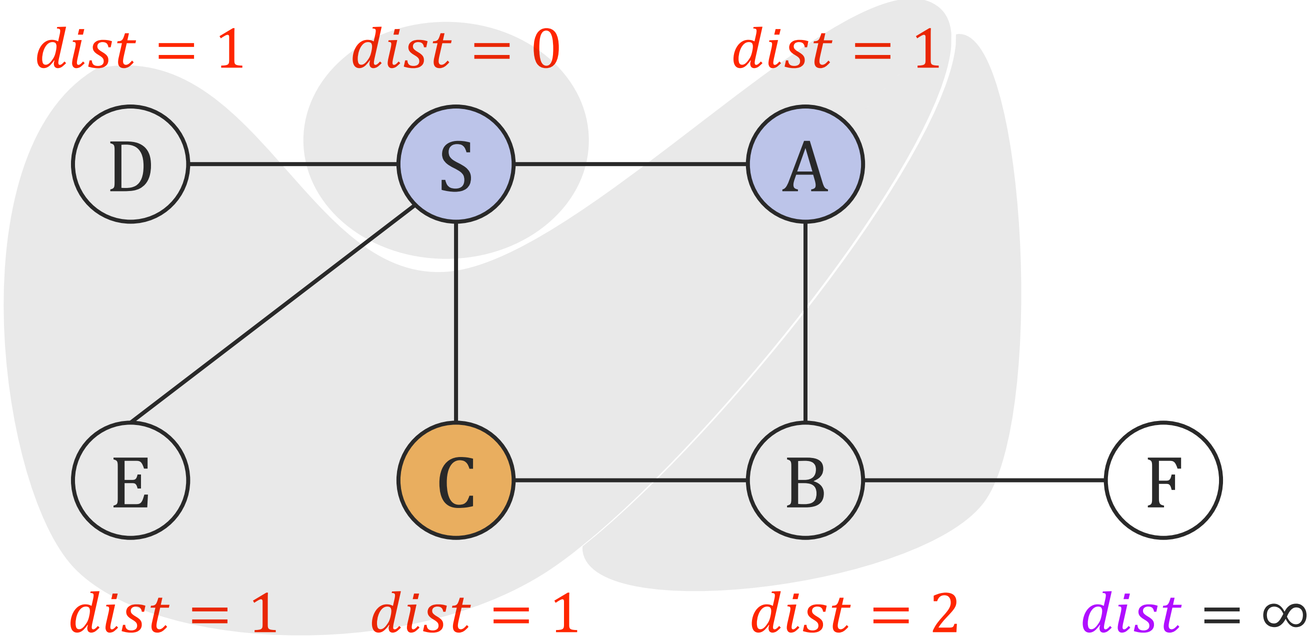
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, D, E, B\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

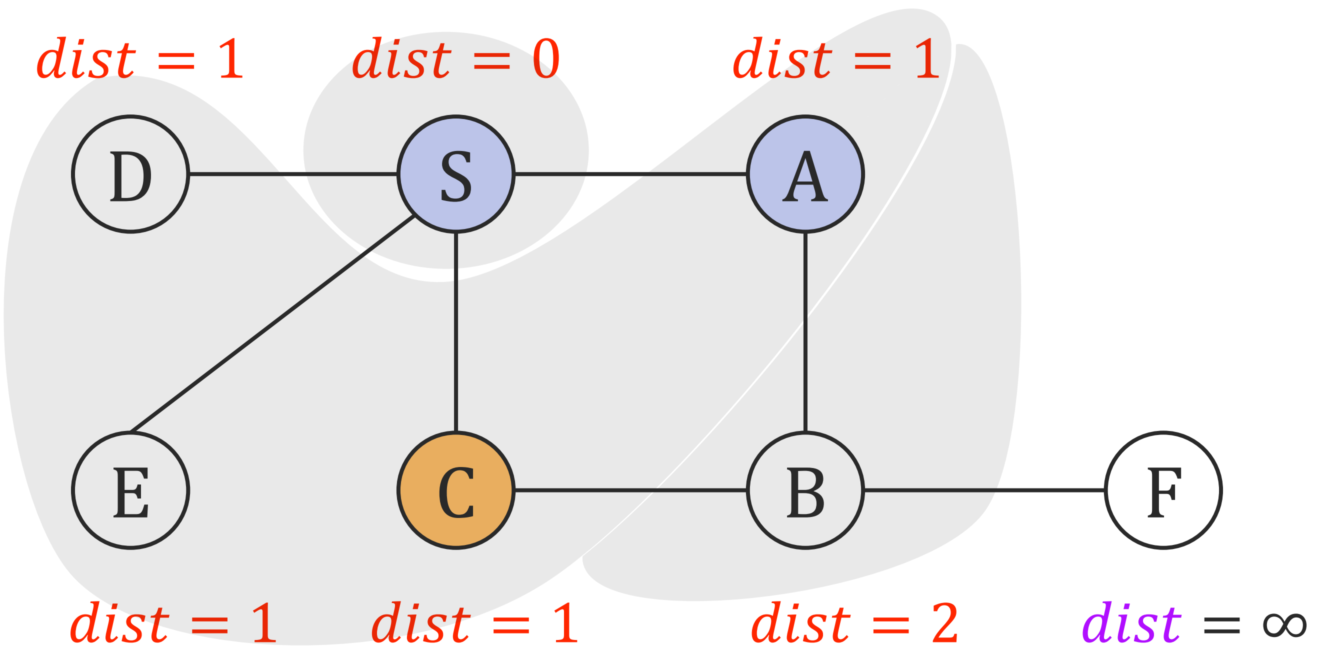
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \text{D}, \text{E}, \text{B}\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

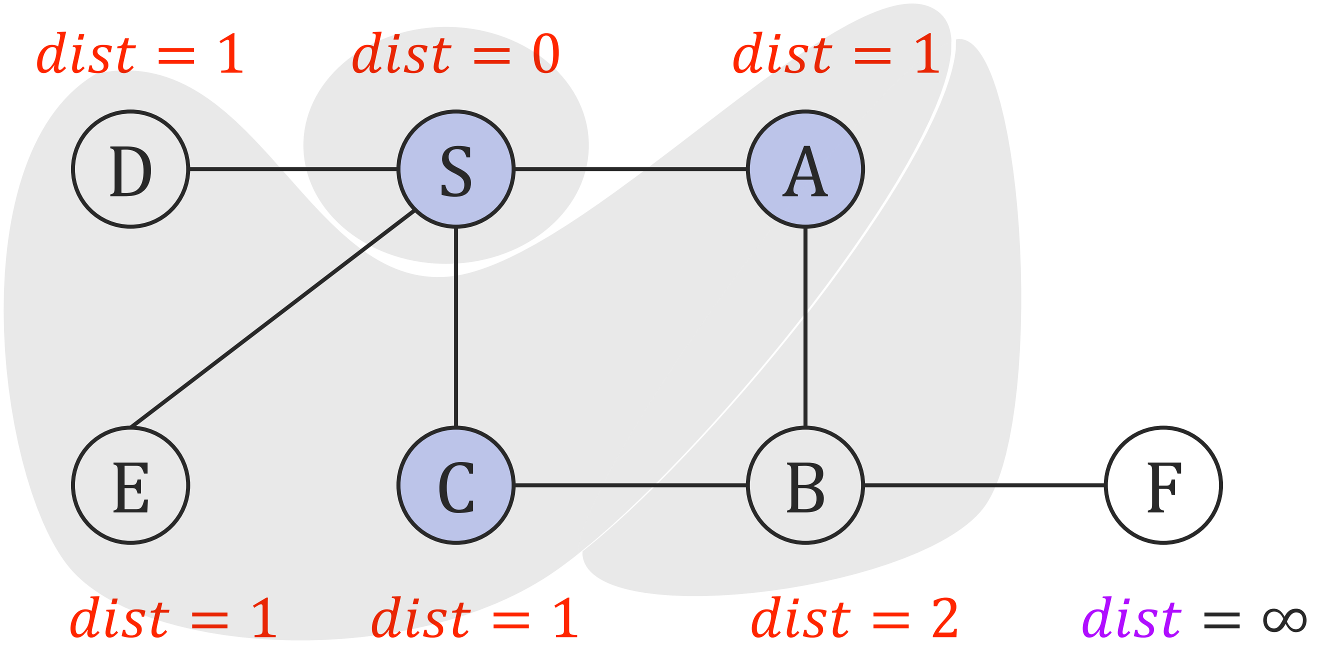
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, D, E, B\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

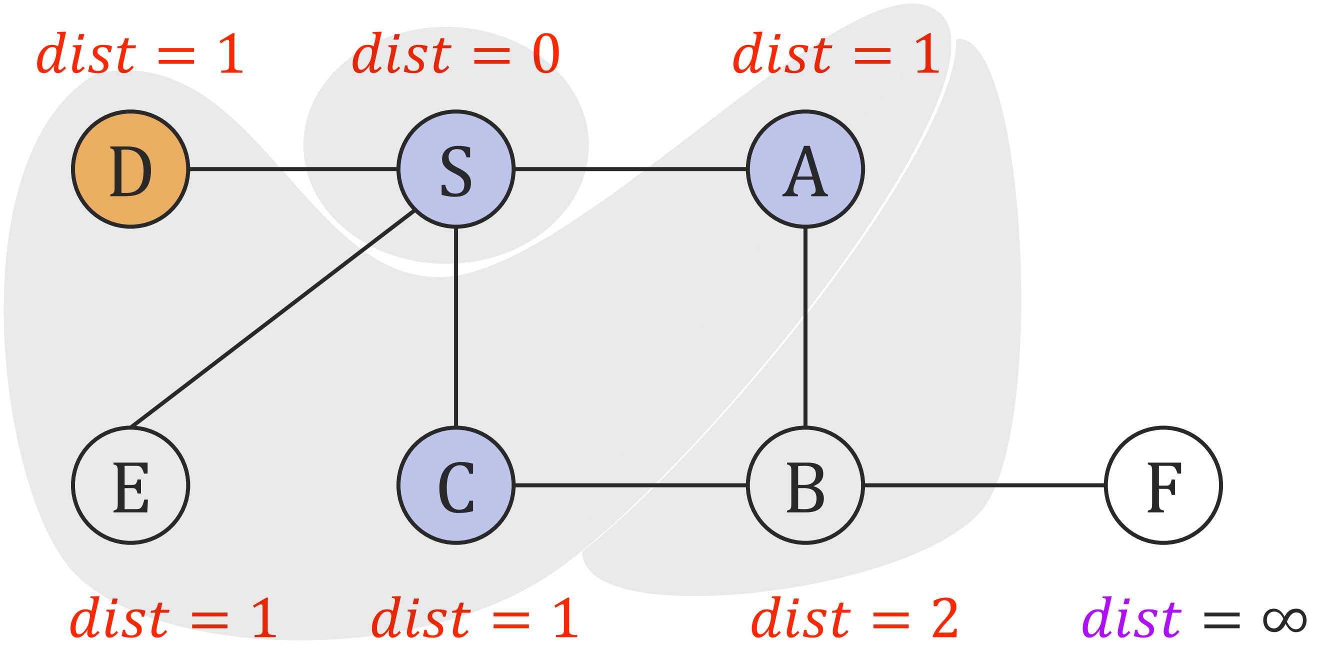
if $dist[v] = \infty$

$$\boxed{\begin{array}{l} enqueue(Q, v) \\ dist[v] = dist[u] + 1 \end{array}}$$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, E, B\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

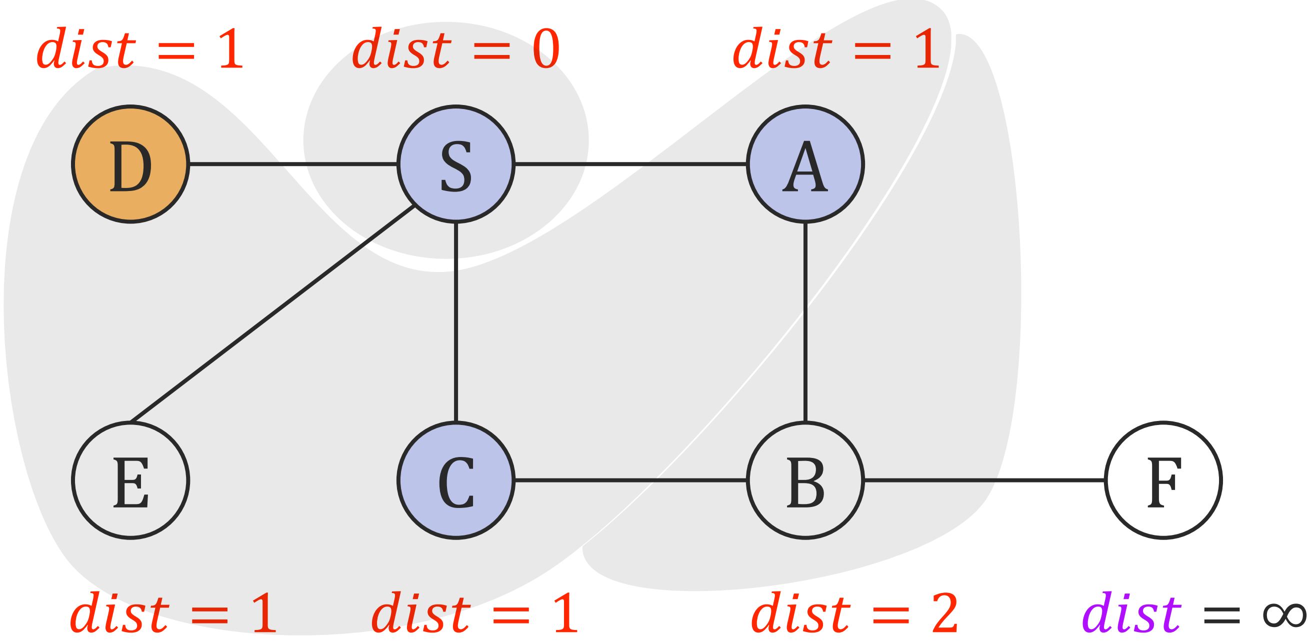
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, E, B\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

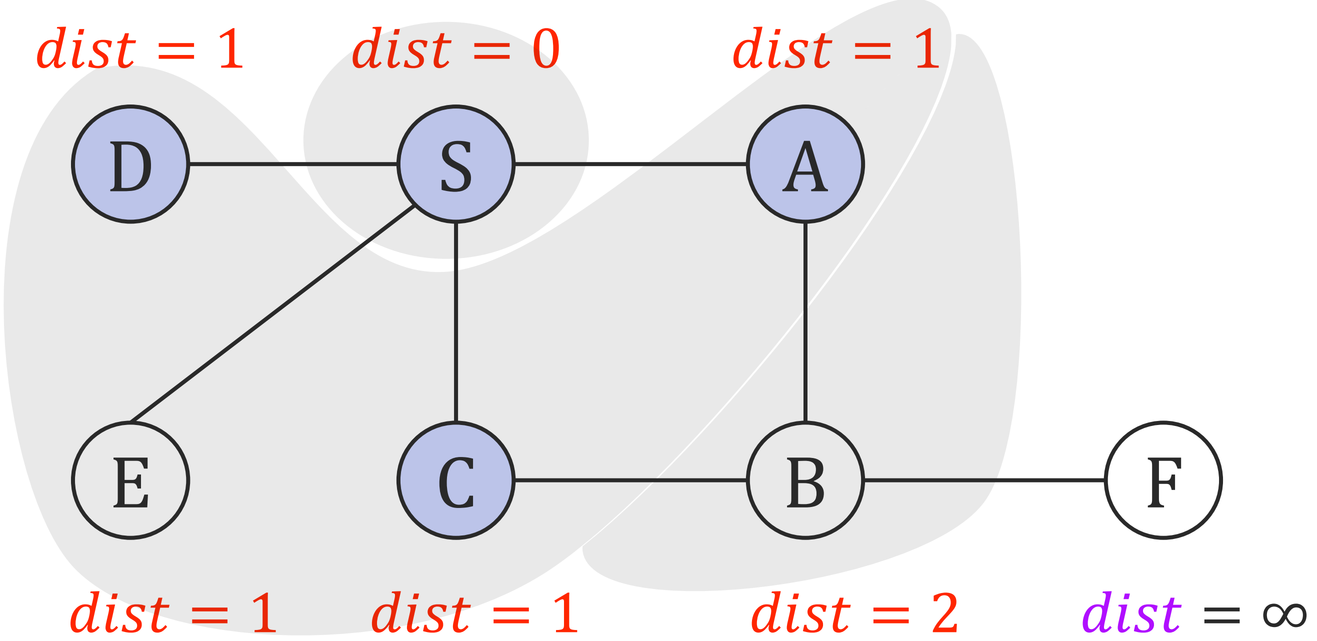
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, E, B\}$

$bfs(G, s)$

  int array $dist(n)$ // initialize to all ∞

  $dist[s] = 0$

  $Q = \{s\}$ //A queue containing s

  While Q is not empty

    $u = dequeue(Q)$

    for all v, s.t. $(u, v) \in E$

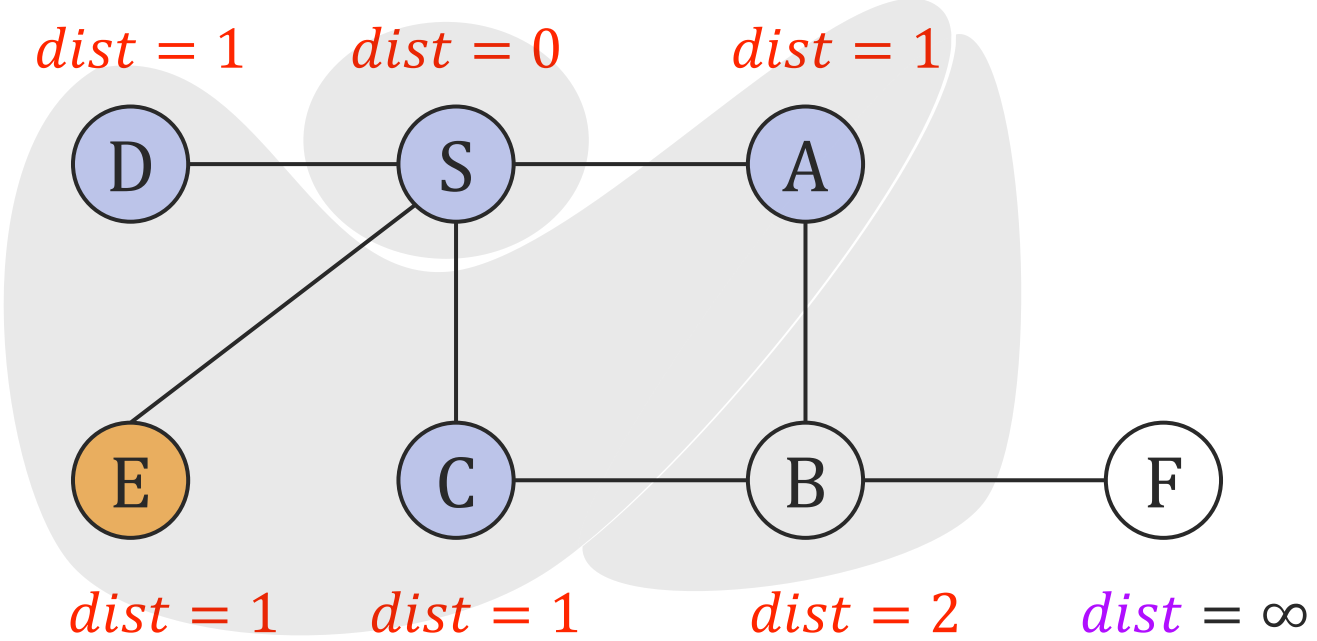
      if $dist[v] = \infty$

        $\boxed{enqueue(Q, v) \\ dist[v] = dist[u] + 1}$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, B\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

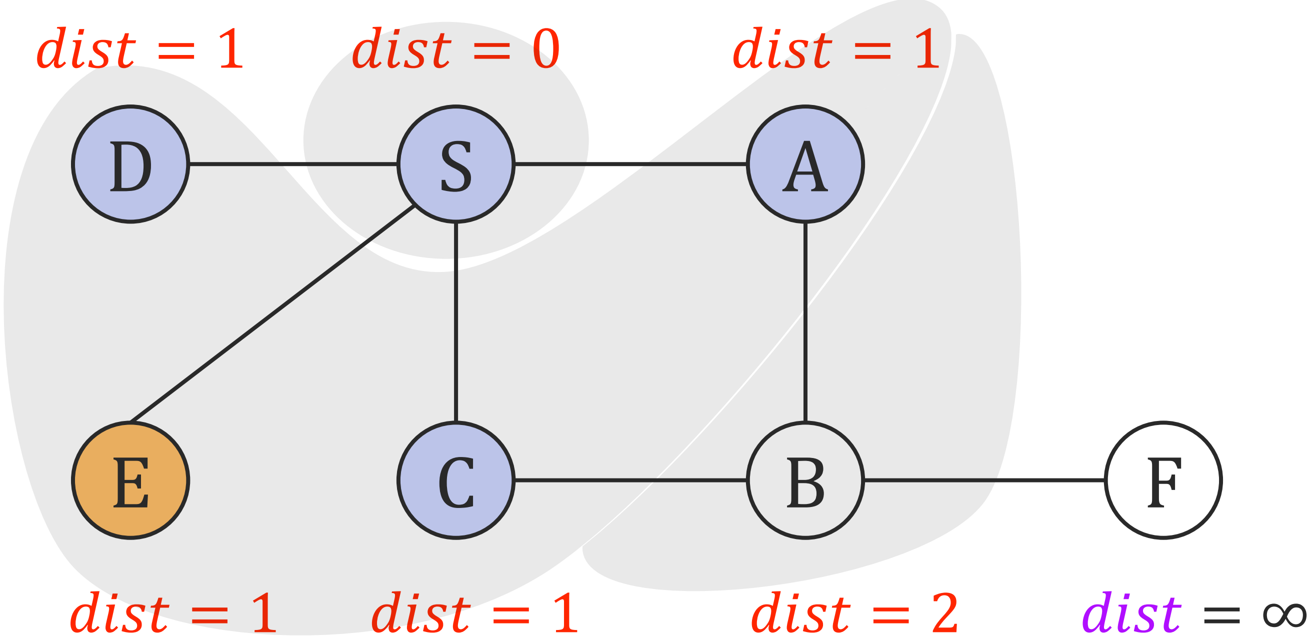
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, B\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ $\quad$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

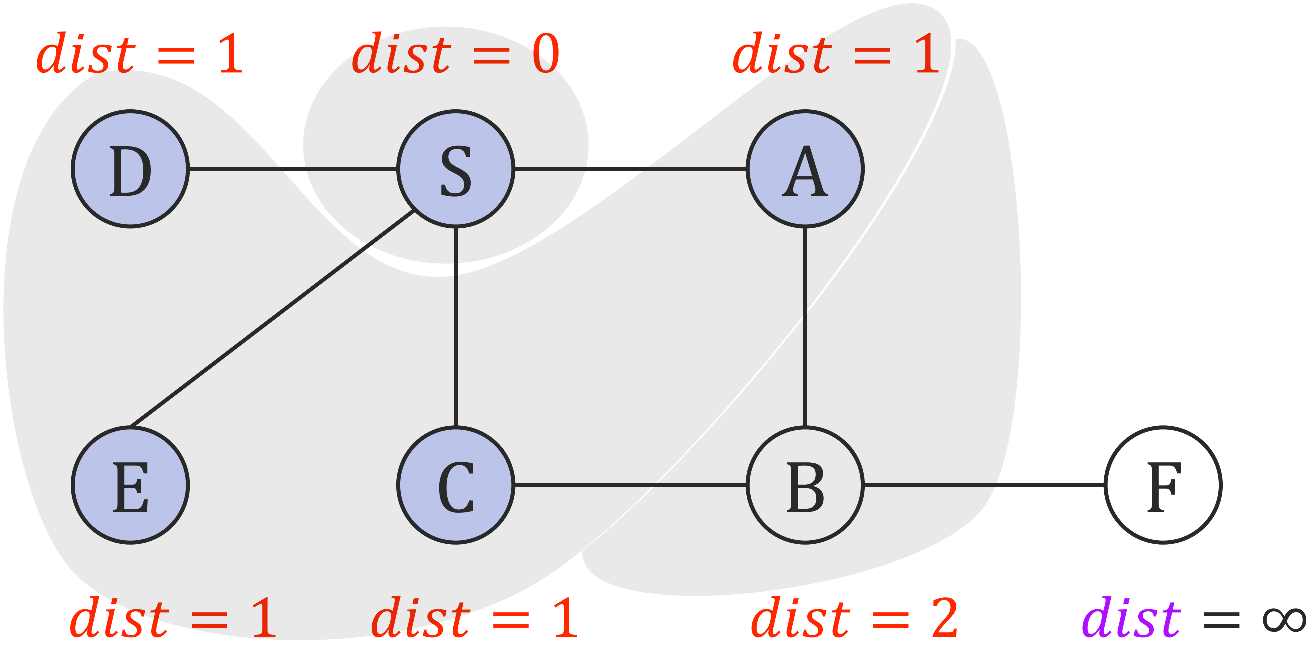
if $dist[v] = \infty$

$\boxed{\begin{array}{l} enqueue(Q, v) \\ dist[v] = dist[u] + 1 \end{array}}$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, B\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

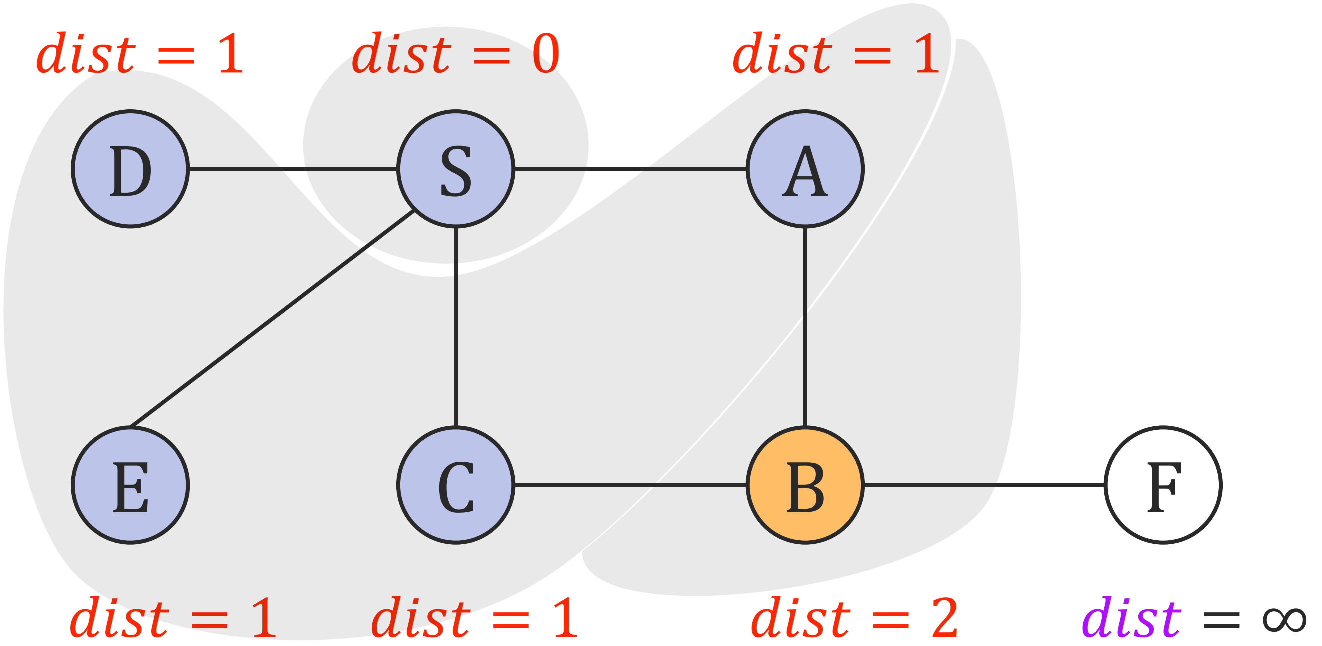
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, \cancel{B}\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

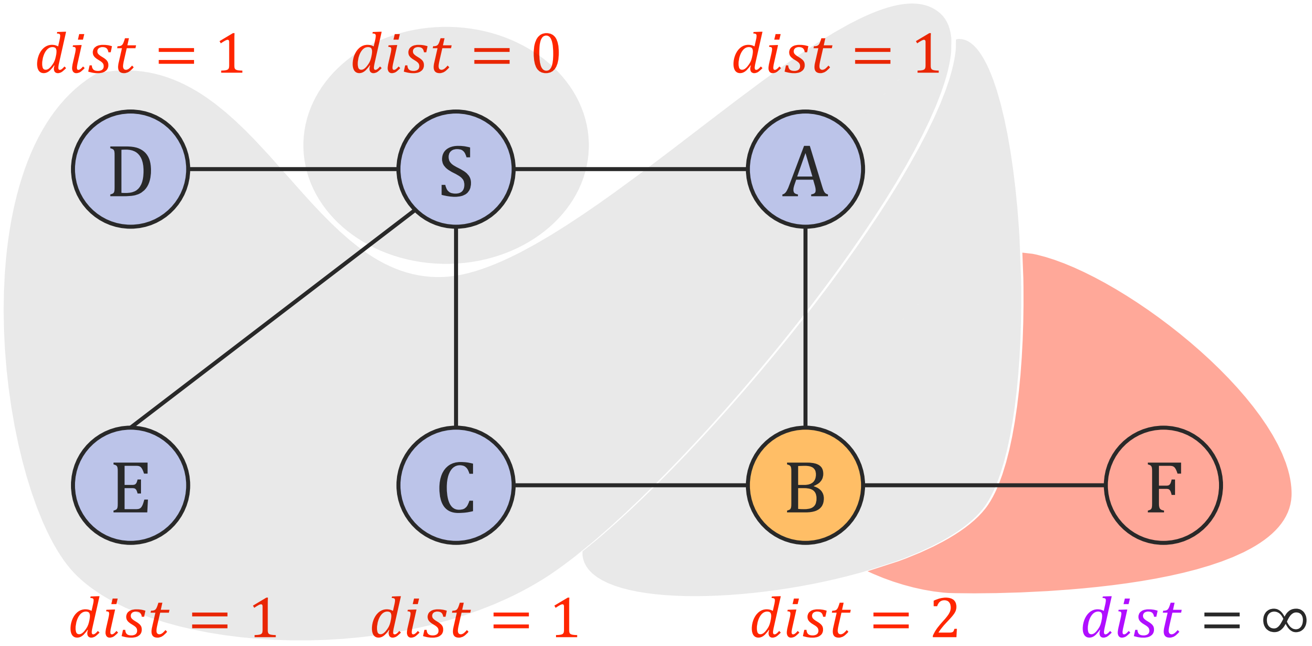
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, \cancel{B}, F\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

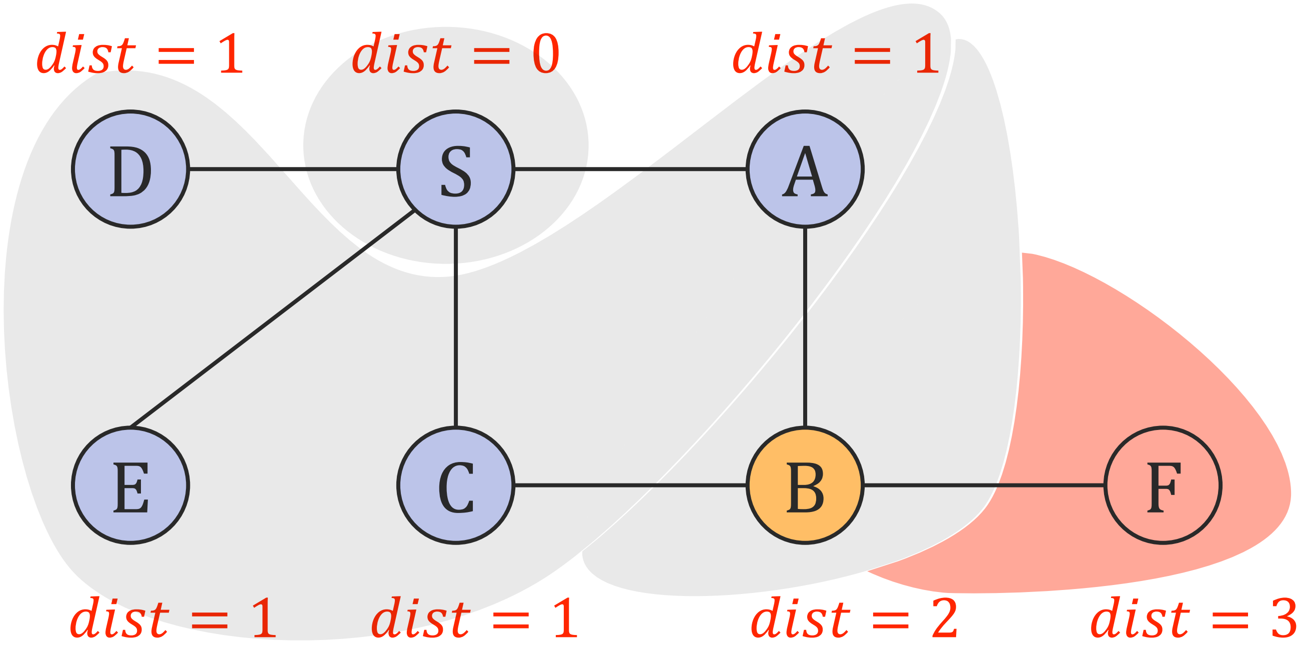
if $dist[v] = \infty$

$\boxed{\begin{array}{l} enqueue(Q, v) \\ dist[v] = dist[u] + 1 \end{array}}$

Breadth-First Search

Current dequeued node

Done, with that iteration of "While"



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, \cancel{B}, F\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

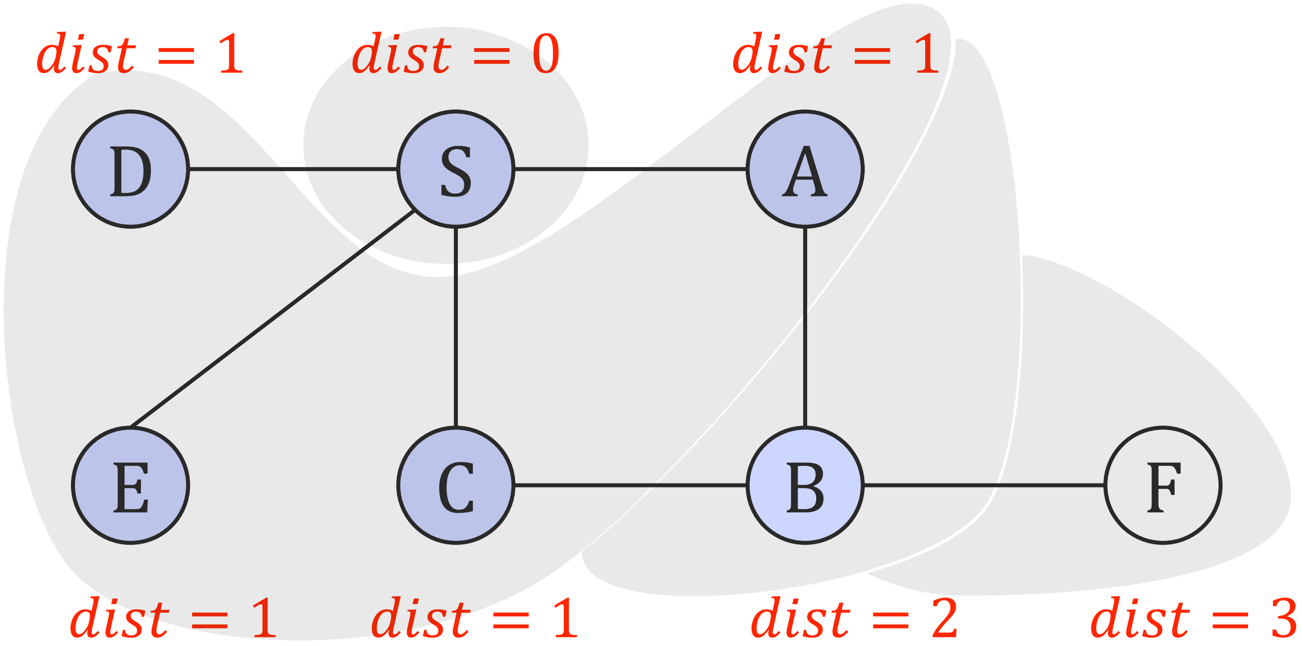
if $dist[v] = \infty$

$$\boxed{\begin{array}{l} enqueue(Q, v) \\ dist[v] = dist[u] + 1 \end{array}}$$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, \cancel{B}, F\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

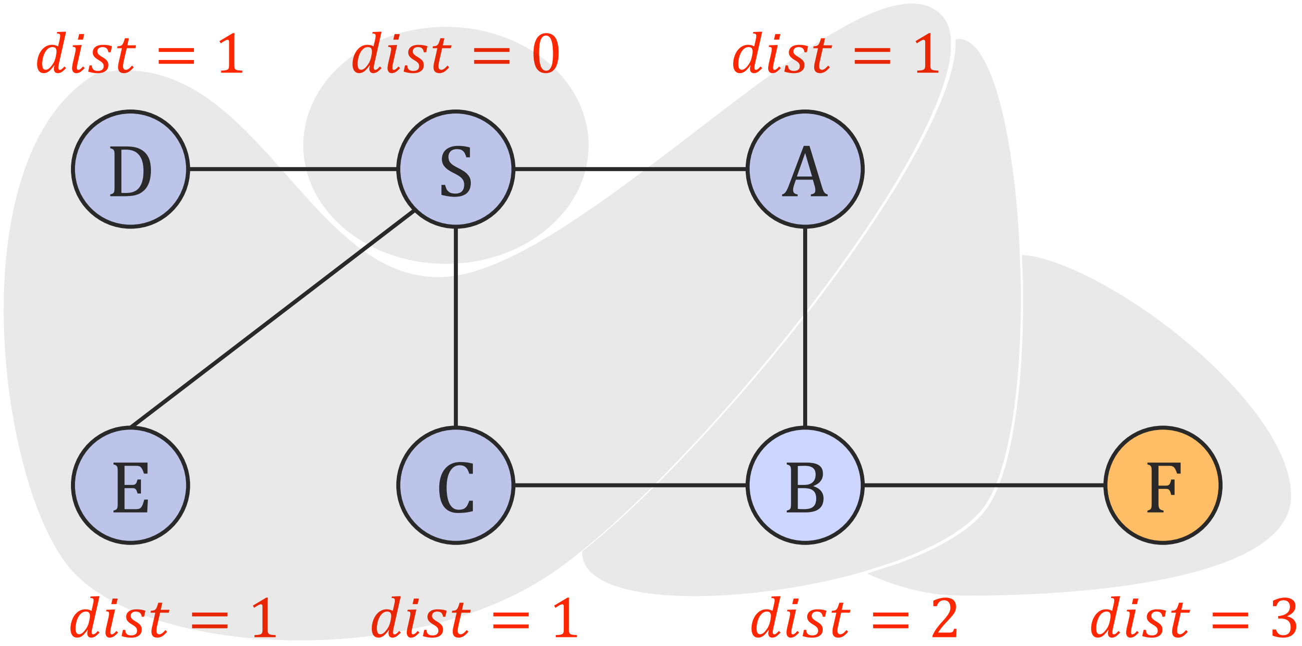
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, \cancel{B}, \cancel{F}\}$

$bfs(G, s)$

  int array $dist(n)$ // initialize to all ∞

  $dist[s] = 0$

  $Q = \{s\}$ //A queue containing s

  While Q is not empty

    $u = dequeue\ (Q)$

    for all v, s.t. $(u, v) \in E$

      if $dist[v] = \infty$

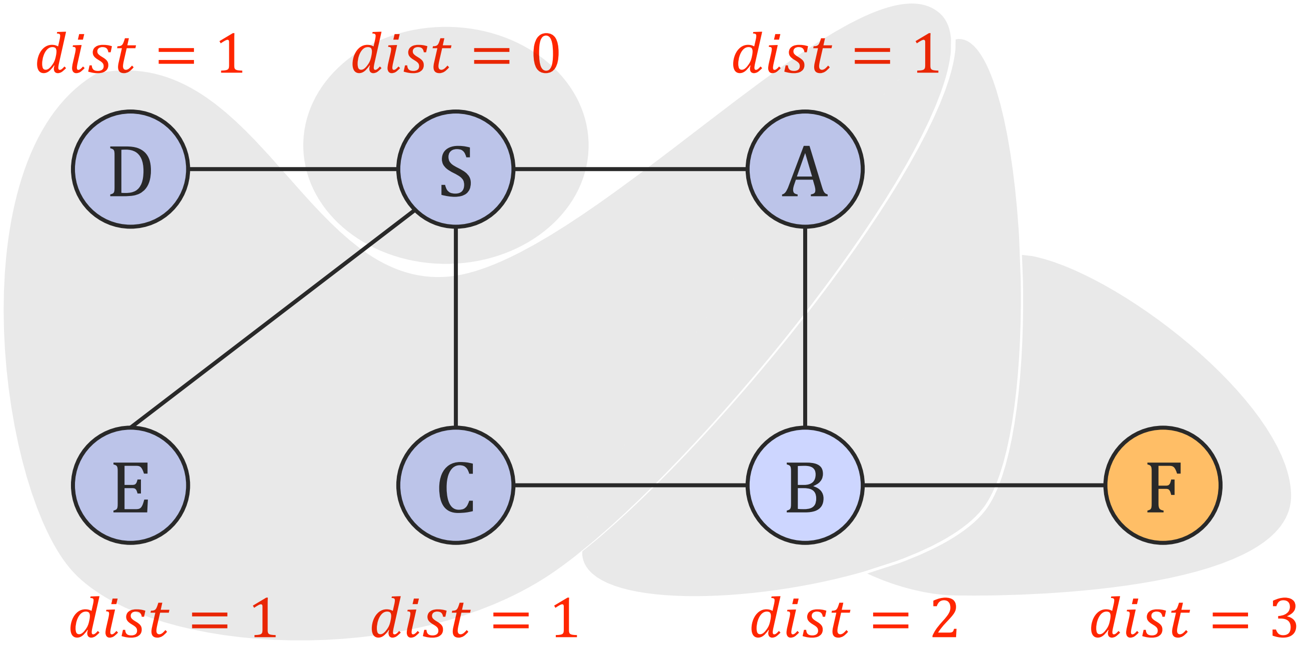
        $enqueue(Q, v)$

        $dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$bfs(G, s)$

  int array $dist(n)$ // initialize to all ∞

  $dist[s] = 0$

  $Q = \{s\}$ //A queue containing s

  While Q is not empty

    $u = dequeue\ (Q)$

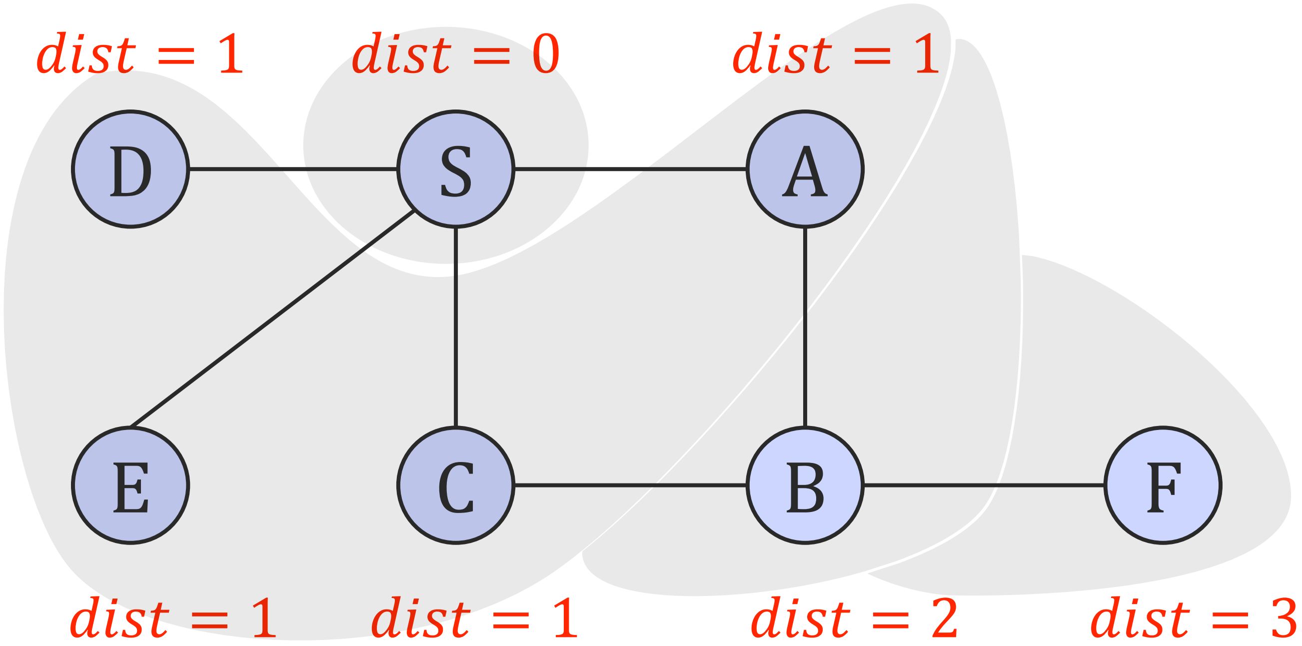
    for all v, s.t. $(u, v) \in E$

      if $dist[v] = \infty$

        $\boxed{\begin{array}{l} enqueue(Q, v) \\ dist[v] = dist[u] + 1 \end{array}}$

$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, \cancel{B}, \cancel{F}\}$

Breadth-First Search



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, \cancel{B}, \cancel{F}\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ $\quad$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

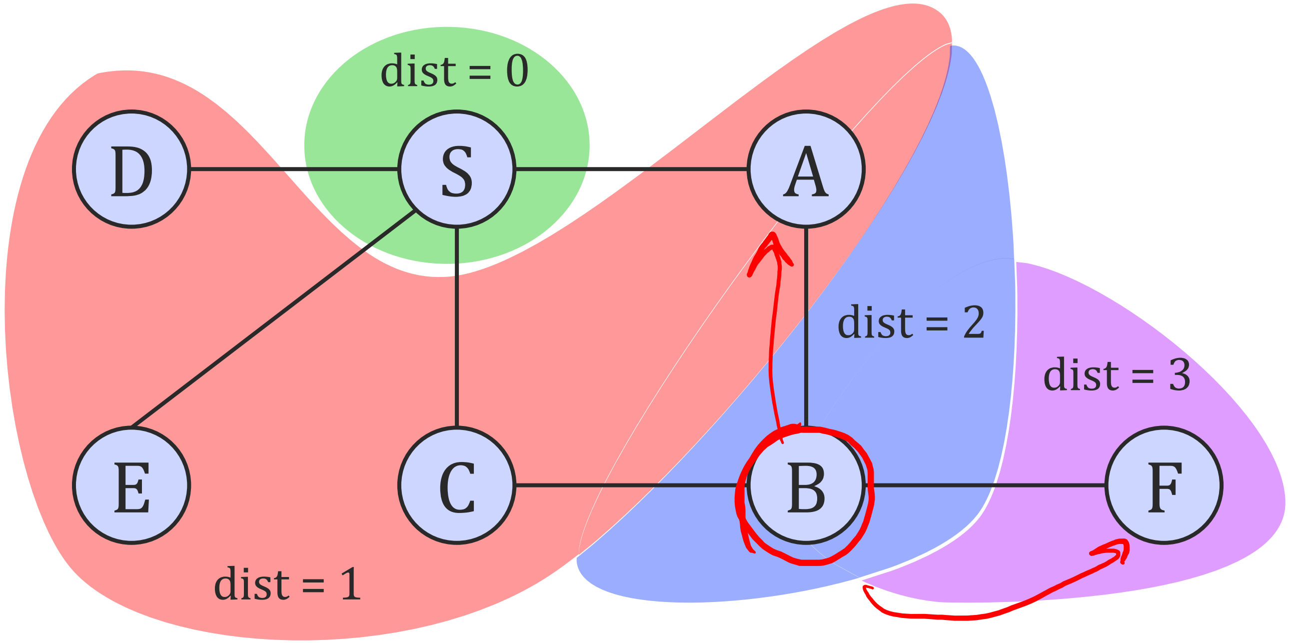
$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Breadth-First Search

Current dequeued node

Done, with that iteration of “While”



$Q = \{\cancel{S}, \cancel{A}, \cancel{C}, \cancel{D}, \cancel{E}, \cancel{B}, \cancel{F}\}$

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\ (Q)$

for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

$\boxed{\begin{array}{l} enqueue(Q, v) \\ dist[v] = dist[u] + 1 \end{array}}$

Runtime of BFS

enqueue and *dequeue* called only once per node
→ $O(1)$ per node.

For every node u, check its neighbors once
→ $O(\deg(u))$ per node.

$$\sum_{u \in V} O(1 + \deg(u)) = O(n + m)$$

Just like DFS. Is this a coincidence?

- Nope!
- DFS is exactly BFS, if queue were to be replaced with a stack

$bfs(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$Q = \{s\}$ //A queue containing s

While Q is not empty

$u = dequeue\,(Q)$

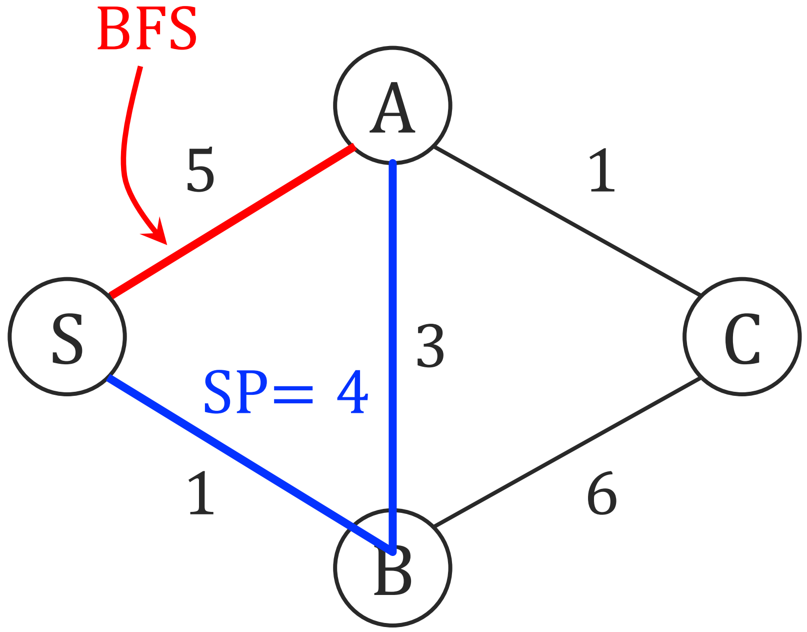
for all v, s.t. $(u, v) \in E$

if $dist[v] = \infty$

$enqueue(Q, v)$

$dist[v] = dist[u] + 1$

Weighted Graphs



Ignoring the weights and playing BFS is an issue

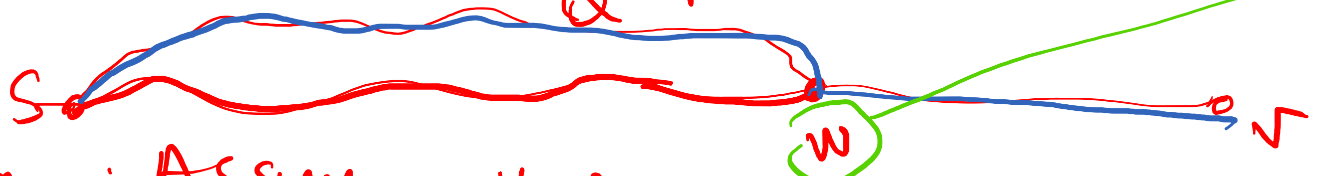
If P is the shortest s-v path, for any $w \in V$ on path P, the shortest s–w path is also on P.

Useful fact

Any sub-path of a shortest path is also a shortest path.

w is any node on path P

Assume P is the shortest path from s to v



Proof by Contradiction: Assume that shortest s–w path isn't P, it's Q: $d_Q(s,w) < d_P(s,w)$

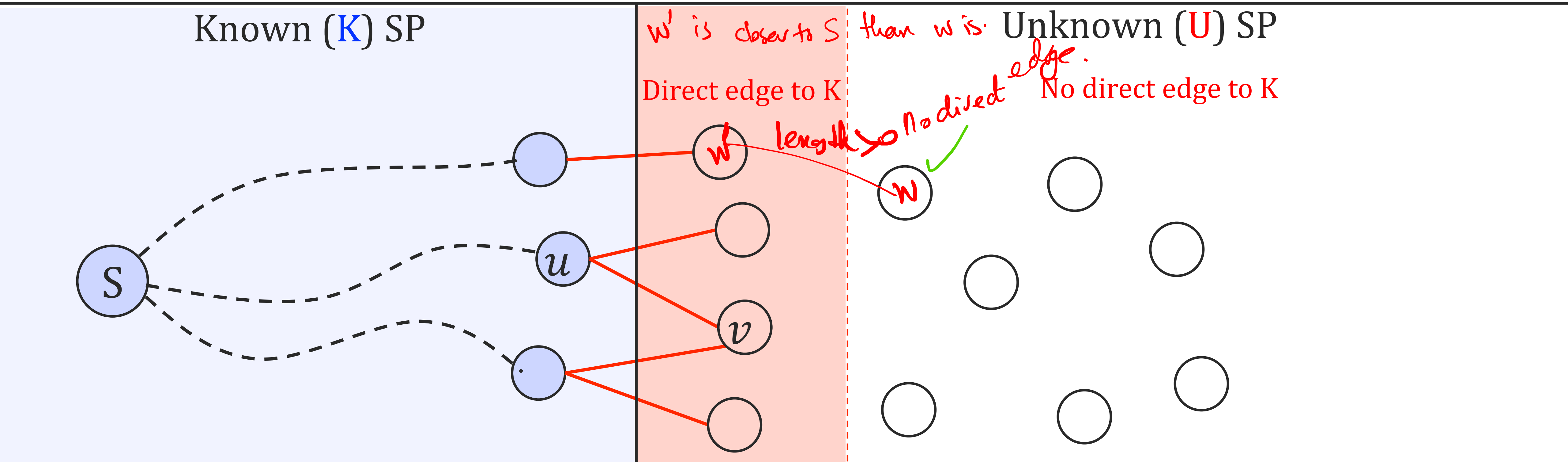
then blue path even shorter s,v path:

$$d_Q(s,w) + d_P(w,v) < d_P(s,w) + d_P(w,v) \stackrel{\leq}{=} d_P(s,v)$$

Dijkstra's Algorithm Intuition

K : Set of "known" nodes where length of SP is computed (and less than "unknown" nodes)

The next node to add to K: v must have a direct edge to K. Why?

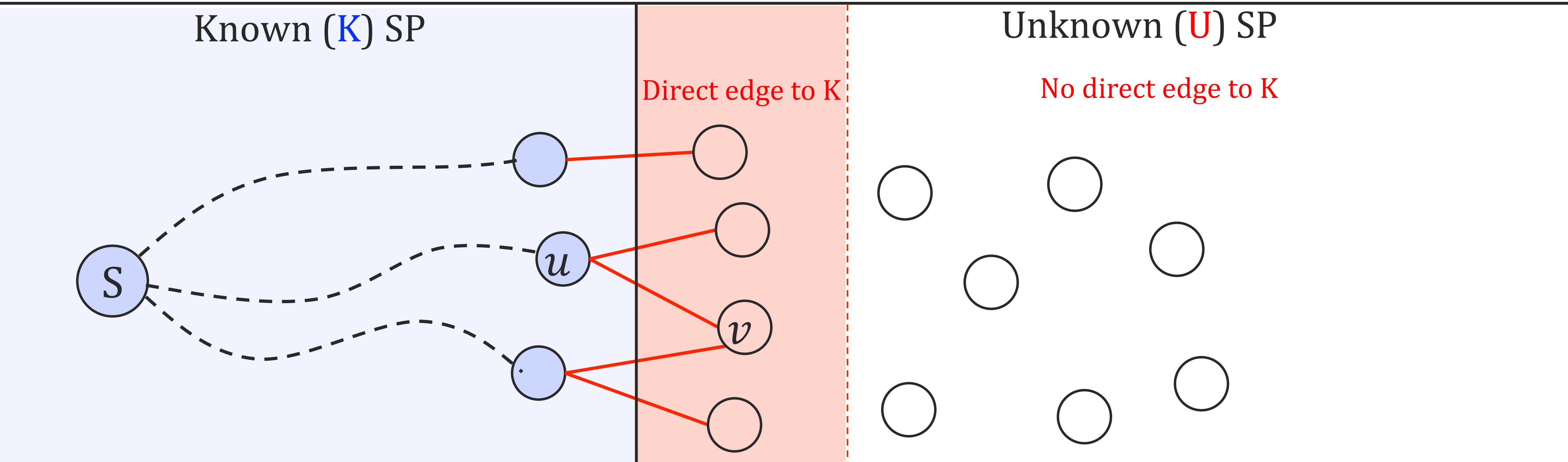


Dijkstra's Algorithm Intuition

K : Set of "known" nodes where length of SP is computed (and less than "unknown" nodes)

The next node to add to K: v must have a direct edge to K. Why?

→ Which one? The one with smallest $dist(s, u) + \ell(u, v)$.



Dijkstra's Algorithm Intuition

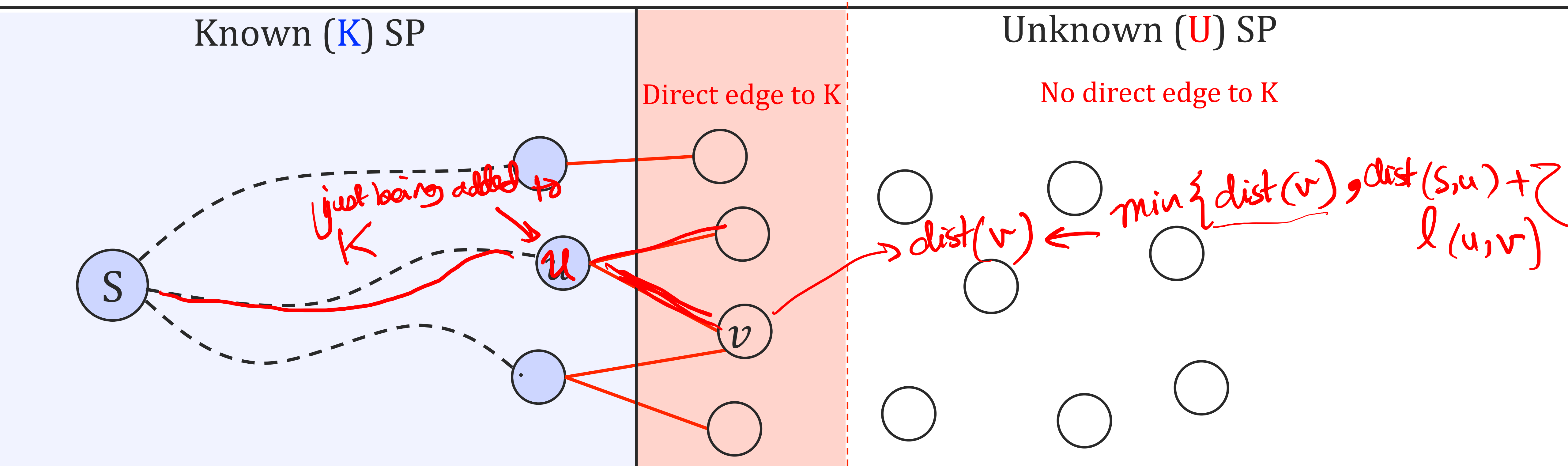
K : Set of "known" nodes where length of SP is computed (and less than "unknown" nodes)

The next node to add to K: v must have a direct edge to K. Why?

→ Which one? The one with smallest $dist(s, u) + \ell(u, v)$.

Don't recomputing all of these distances at every round.

→ Keep overestimates of distances for U and update estimates when a neighbor enters K.



Dijkstra's Algorithm

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = \mathrm{V}$ $\qquad$ $// \; K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

$U \leftarrow U \setminus v$

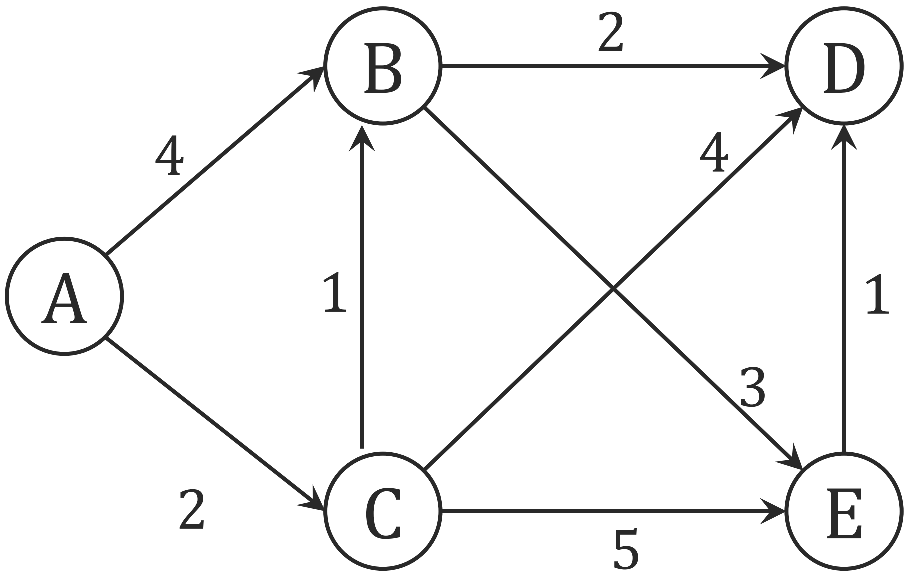
for all $(v, w) \in E$

If $dist[w] \cancel{<} > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm

on the next n slides.



U	A	B	C	D	E
$dist$	0	∞	∞	∞	∞

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $\quad // K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

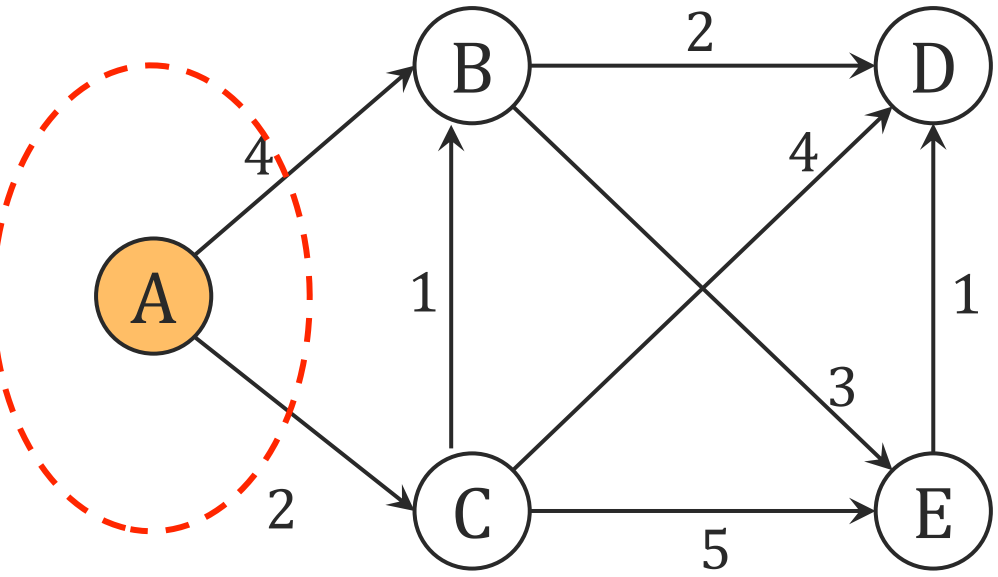
$U \leftarrow U \setminus v$ $\quad dist(w) = \min\{dist(w), dist(v) + \ell(v,w)\}$

for all $(v, w) \in E$

If $dist[w] \cancel{>} dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	A	B	C	D	E
$dist$	0	∞	∞	∞	∞

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $\quad // \; K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

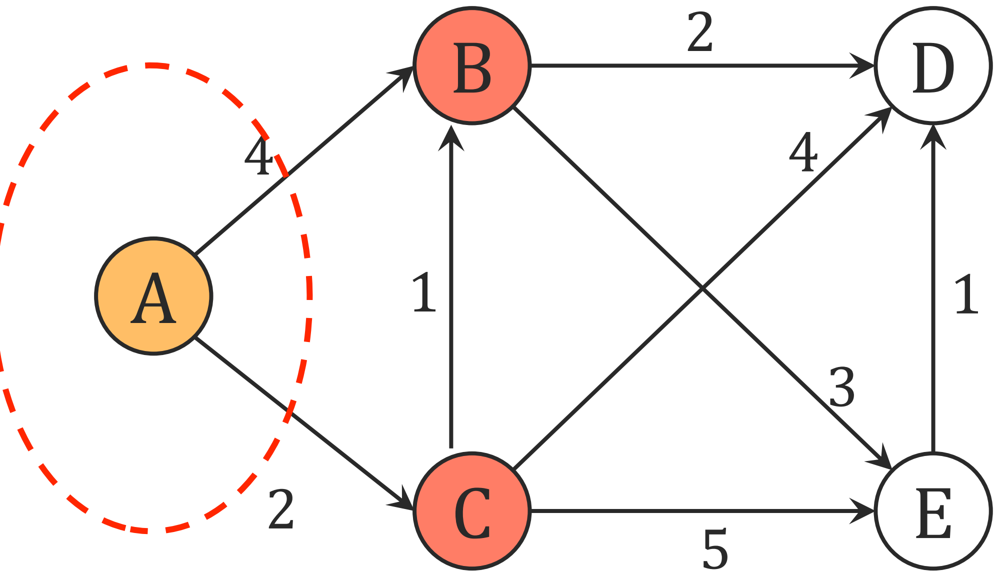
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	A	B	C	D	E
$dist$	0	4	2	∞	∞

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = \mathrm{V}$ $\quad$ $// \; K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

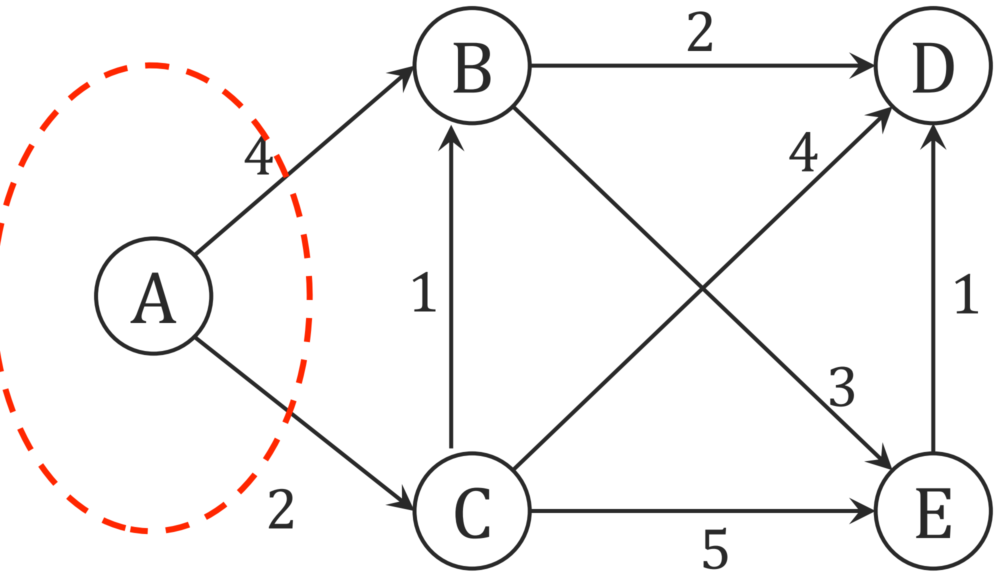
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	B	C	D	E
$dist$	0	4	2	∞	∞

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = \mathrm{V}$ $\quad$ $// \; K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

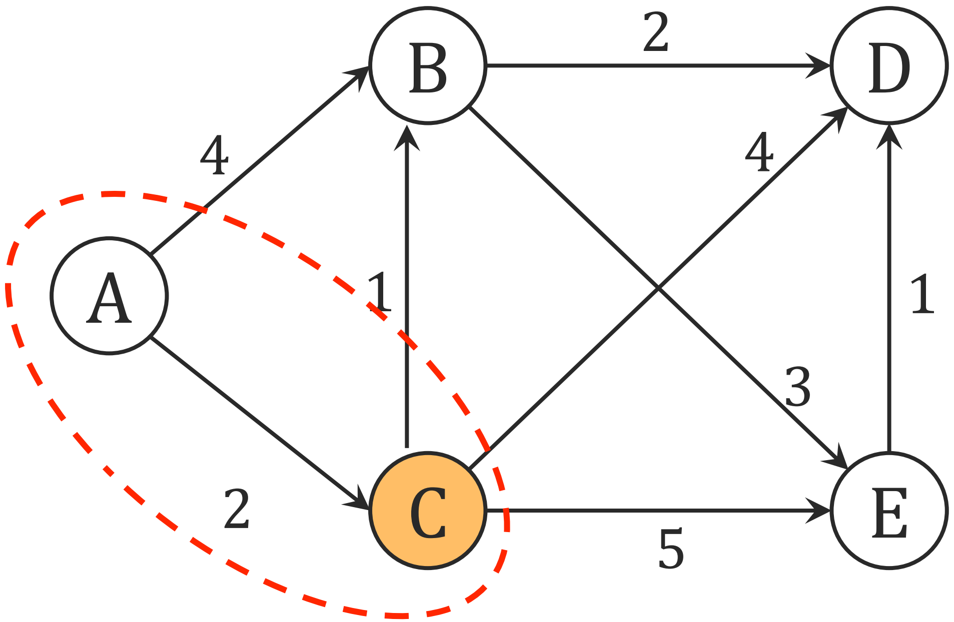
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] \cancel{<} > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	B	C	D	E
$dist$	0	4	2	∞	∞

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $\quad$ $// K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

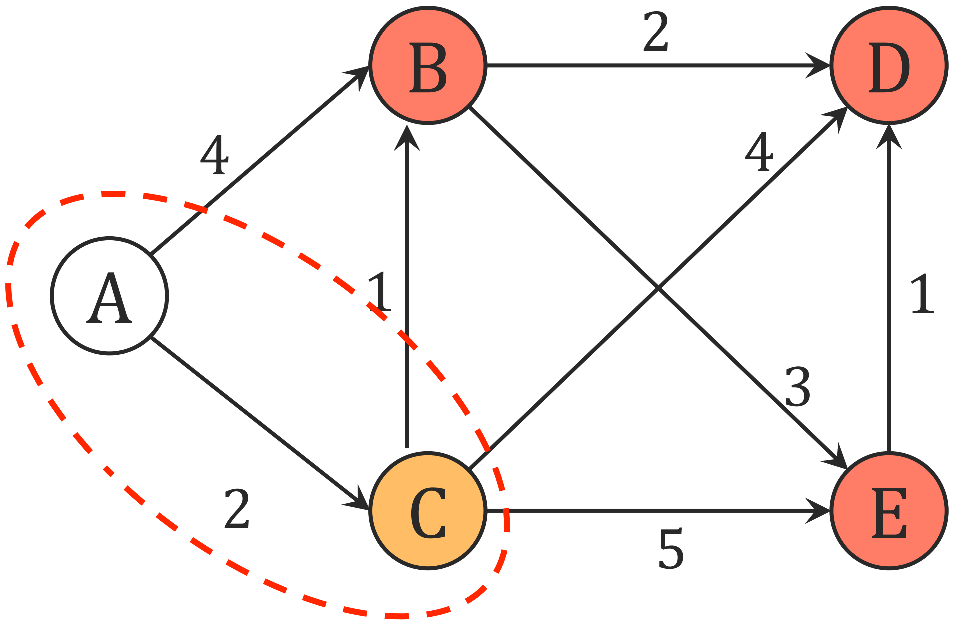
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	B	C	D	E
$dist$	0	3	2	6	7

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $\qquad$ $// \ K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

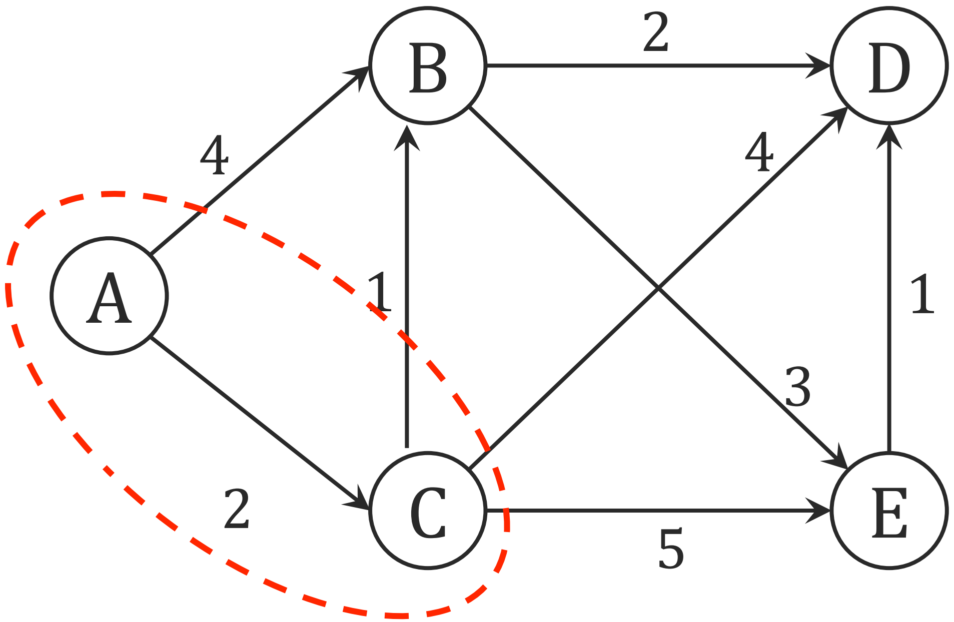
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	B	~~C~~	D	E
$dist$	0	3	2	6	7

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $\qquad$ $// \ K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

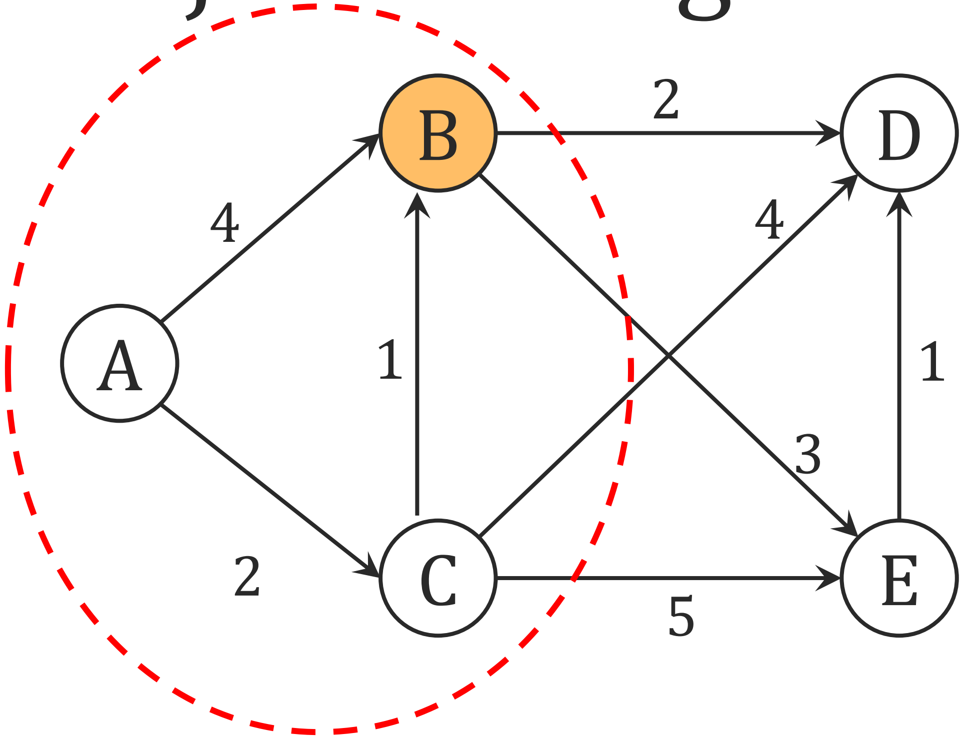
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	B	~~C~~	D	E
$dist$	0	3	2	6	7

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $\quad // K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

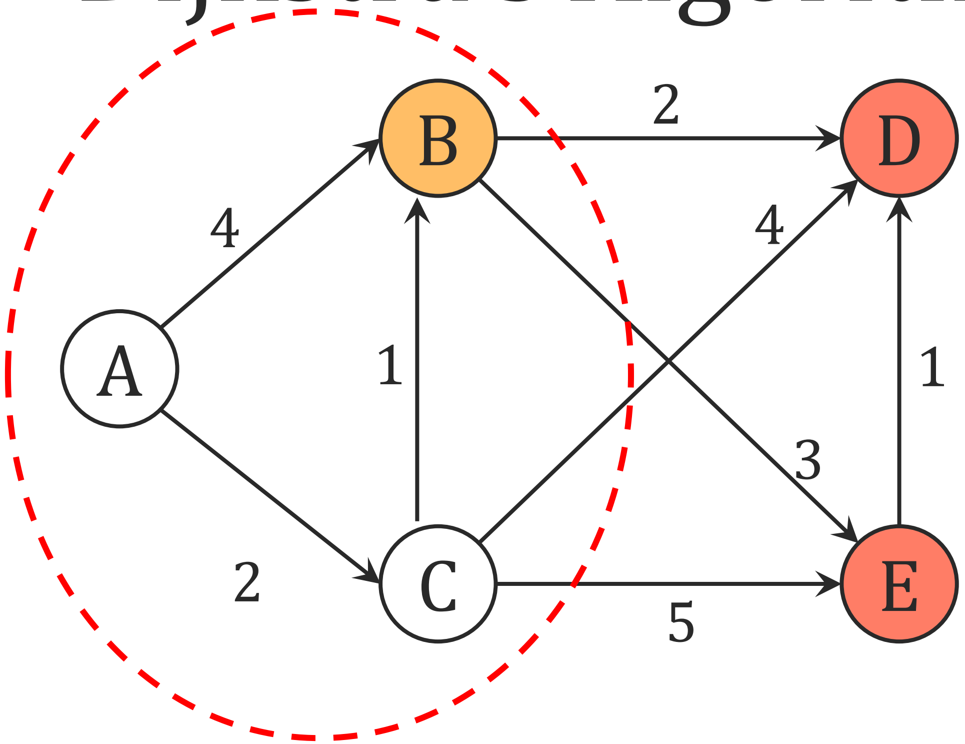
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] \cancel{<} > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	B	~~C~~	D	E
$dist$	0	3	2	5	6

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = \mathrm{V}$ $\quad // K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

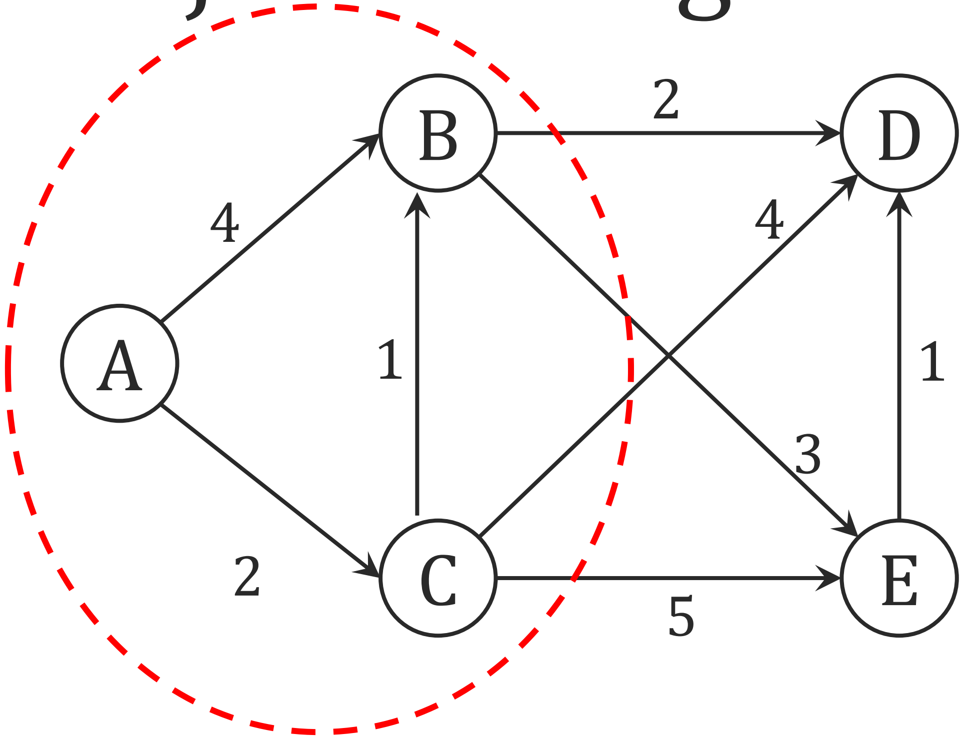
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	~~B~~	~~C~~	D	E
$dist$	0	3	2	5	6

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $// K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

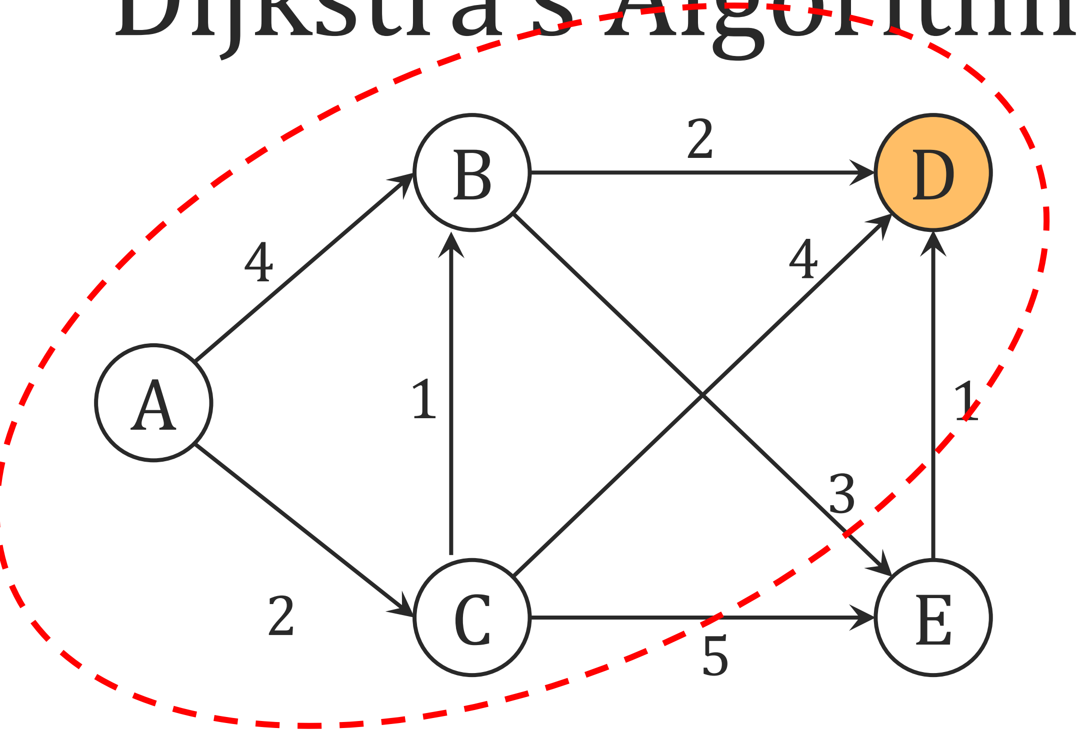
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	~~B~~	~~C~~	D	E
$dist$	0	3	2	5	6

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = \mathrm{V}$ $\quad$ // $K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

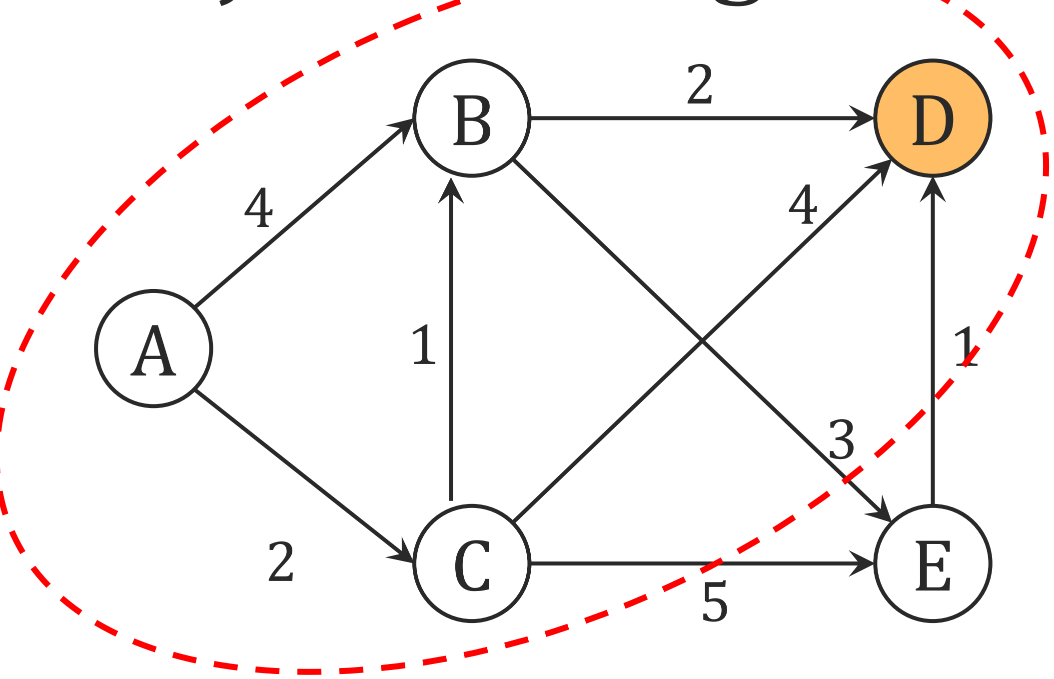
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	~~B~~	~~C~~	D	E
$dist$	0	3	2	5	6

$dijkstra(G, s)$

  int array $dist(n)$ // initialize to all ∞

  $dist[s] = 0$

  $U = \mathrm{V}$ $\qquad$ $// \; K = V \setminus U$

  While U is not empty

    $v \leftarrow$ node in U with smallest $dist[v]$

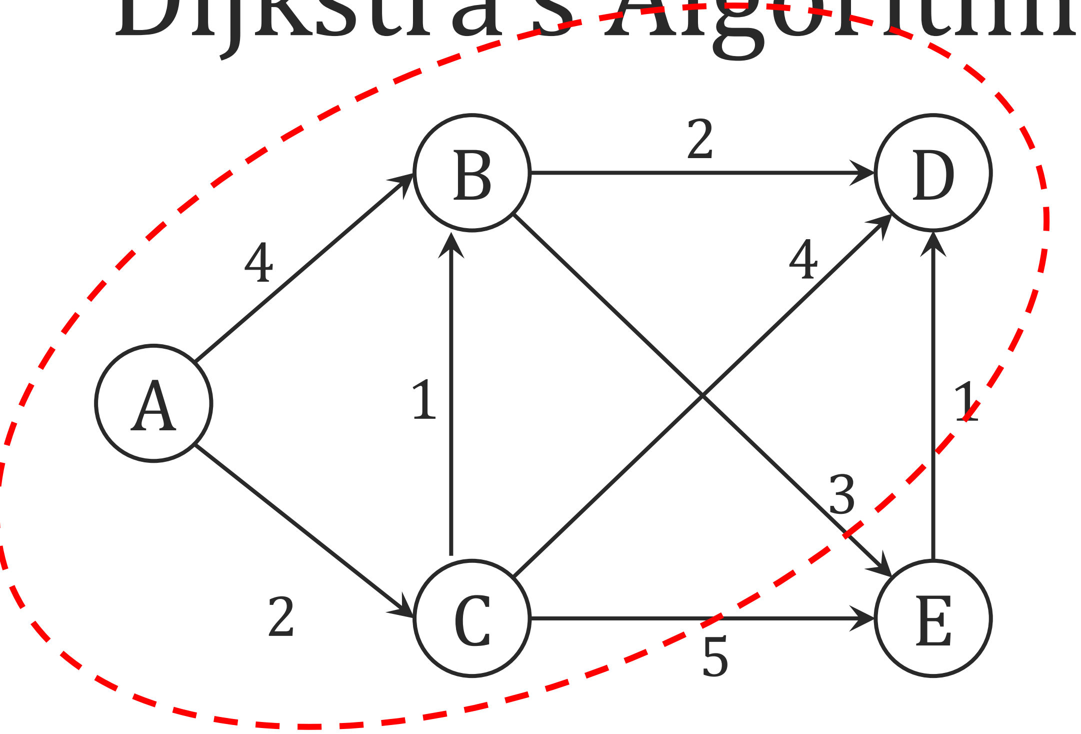
    $U \leftarrow U \setminus v$

    for all $(v, w) \in E$

      If $dist[w] > dist[v] + \ell(v, w)$

        $dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	~~B~~	~~C~~	~~D~~	E
$dist$	0	3	2	5	6

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = \mathrm{V}$ $\quad$ $// \ K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

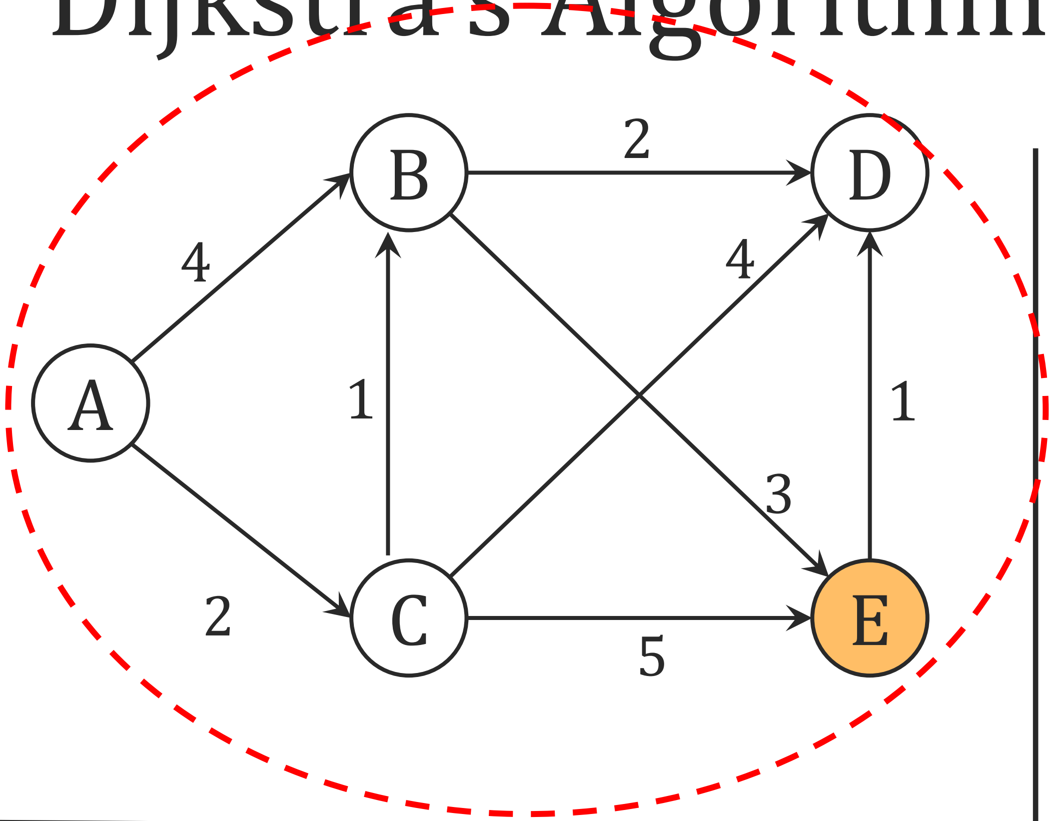
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] \cancel{<} > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	~~B~~	~~C~~	~~D~~	E
$dist$	0	3	2	5	6

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $\quad$ $// K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

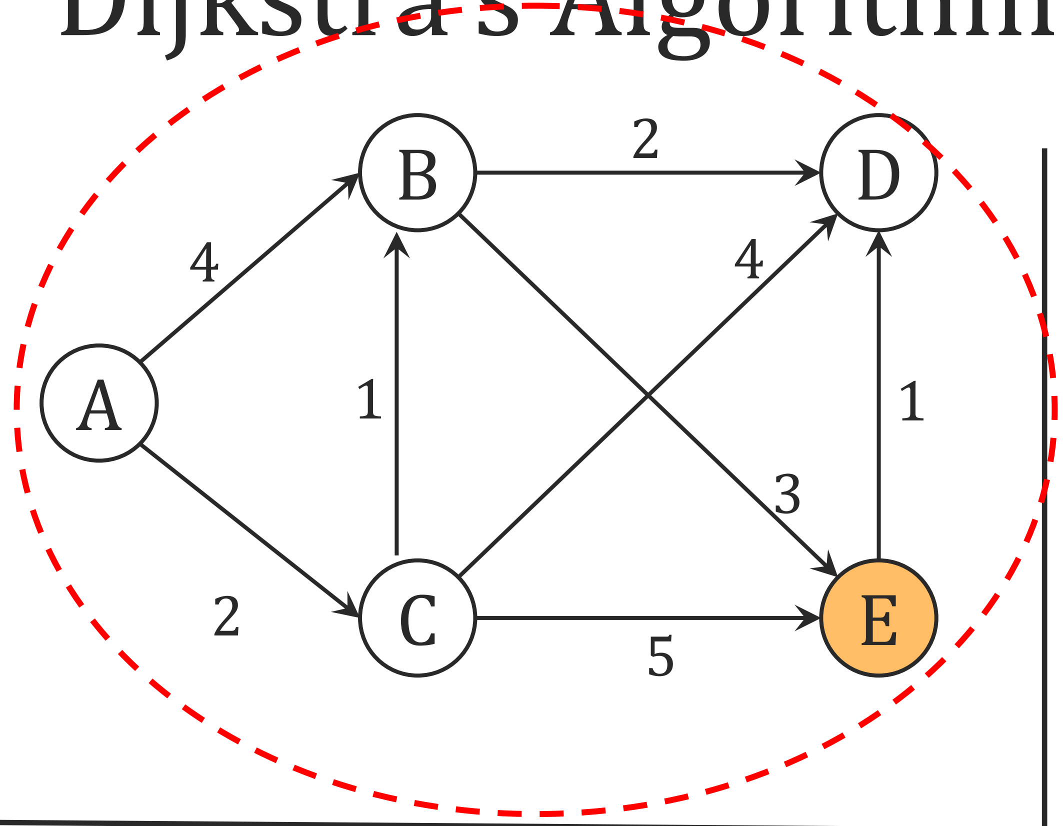
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	~~B~~	~~C~~	~~D~~	E
$dist$	0	3	2	5	6

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

$dist[s] = 0$

$U = V$ $\quad // K = V \setminus U$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

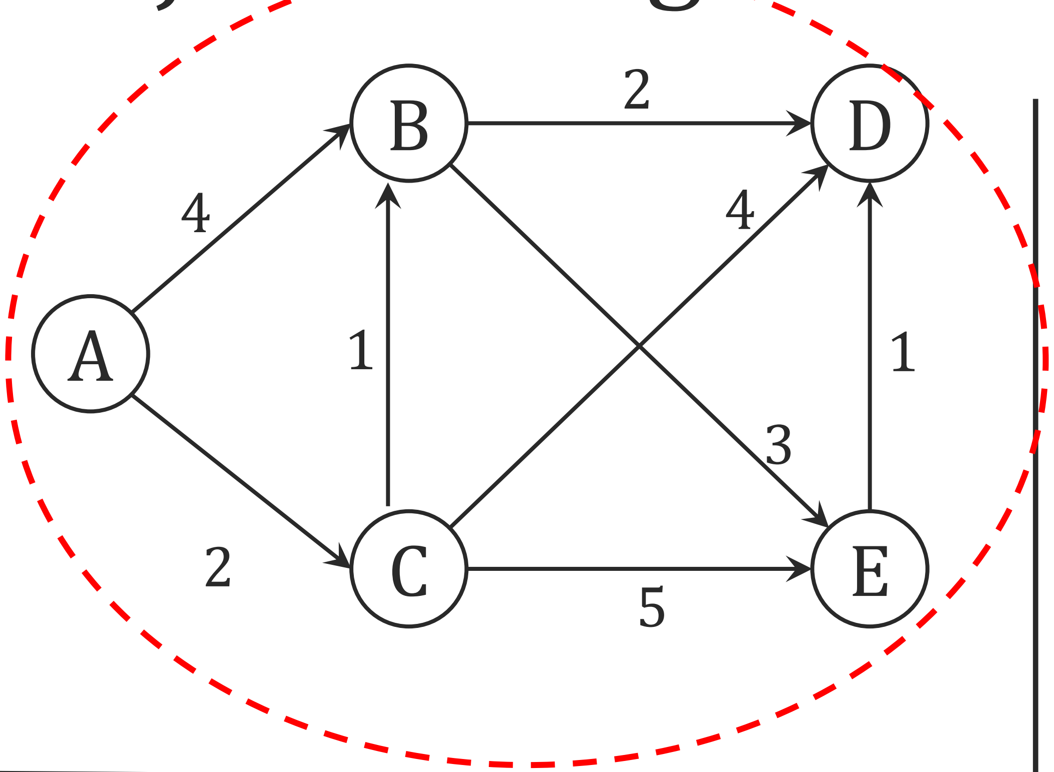
$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] \cancel{<} dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

Dijkstra's Algorithm



U	~~A~~	~~B~~	~~C~~	~~D~~	~~E~~
$dist$	0	3	2	5	6

$dijkstra(G, s)$

  int array $dist(n)$ // initialize to all ∞

  $dist[s] = 0$

  $U = V$ $// \ K = V \setminus U$

  While U is not empty

    $v \leftarrow$ node in U with smallest $dist[v]$

    $U \leftarrow U \setminus v$

    for all $(v, w) \in E$

      If $dist[w] \cancel{<} dist[v] + \ell(v, w)$

        $dist[w] = dist[v] + \ell(v, w)$

What is the shortest path?

The dist data structure is keeping track of the distances.

But what about the actual path from s to all nodes?

$Update(v, w)$ routine:

"If" condition, finds a shorter path, S to v to w.

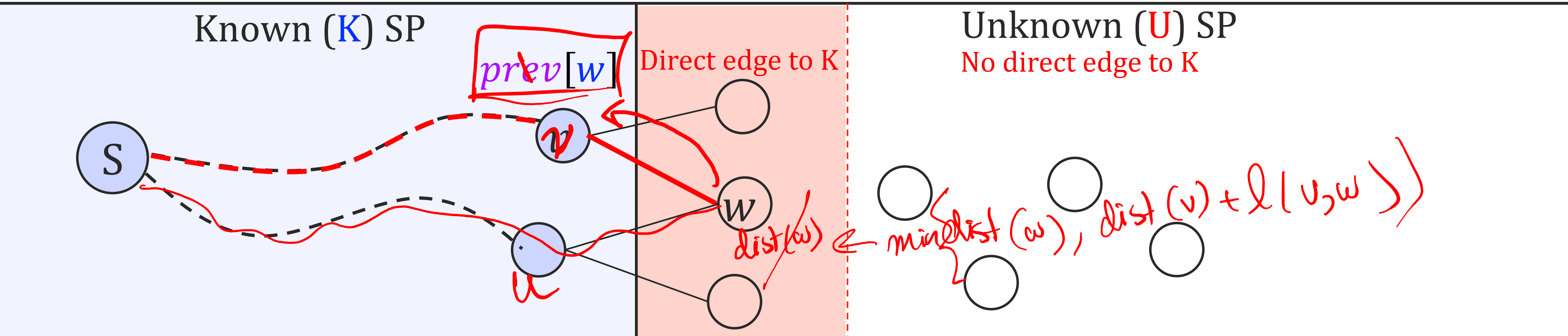
Keep track of the incoming edge, in $prev[w]$.

$Update(v, w)$

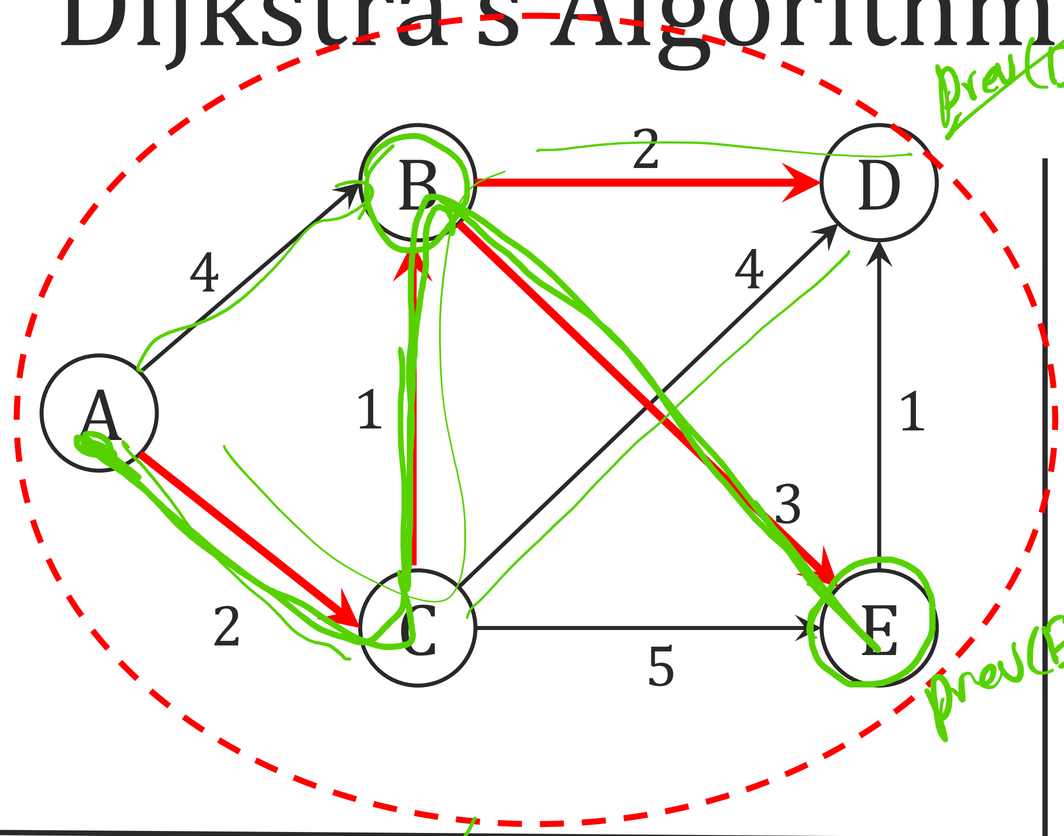
If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

$prev[w] = v$



Dijkstra's Algorithm



~~prev(D)=C~~
prev(D)=B
prev(E)=B

$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

array $prev(n)$ // initialize to all *nil*

// $K = V \setminus U$

$dist[s] = 0$,

$U = V$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] \cancel{<} > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

$\boxed{prev[w] = v}$

U	~~A~~	~~B~~	~~C~~	~~D~~	~~E~~
$dist$	0	~~4~~ 3	2	~~6~~ 5	6

Runtime of Dijkstra

Depends on the data structure used for keeping track of U's distances.

Priority Queue

node

A, B, C, D . . . .

numerical val.

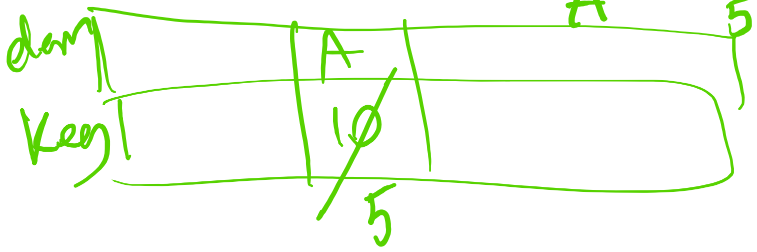
dist(A), dist(B) . . . .

- Insert(elem, key)
- DeleteMin finds the elem with lowest dist.
- DeleteKey (elem, key)

$\left.\begin{array}{l}\text{Insert(elem, key)}\\ \text{DeleteMin}\end{array}\right\}$ n calls

$\left.\text{DecreaseKey (elem, key)}\right\}$ m calls

A 5



$dijkstra(G, s)$

int array $dist(n)$ // initialize to all ∞

array $prev(n)$ // initialize to all nil

$dist[s] = 0,$ $\quad$ $// K = V \setminus U$

$U = \text{V}$

While U is not empty

$v \leftarrow$ node in U with smallest $dist[v]$

$U \leftarrow U \setminus v$

for all $(v, w) \in E$

If $dist[w] > dist[v] + \ell(v, w)$

$dist[w] = dist[v] + \ell(v, w)$

$prev[w] = v$

Priority Queues and Dijkstra

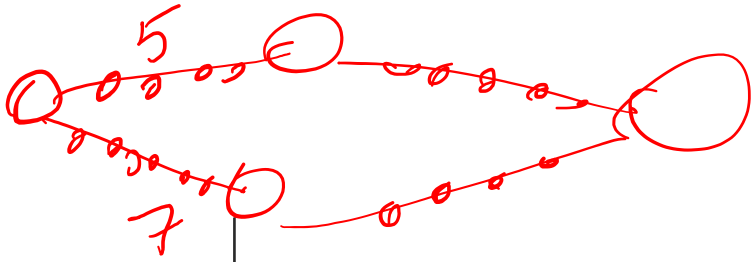
$O(n) \times (\text{Insert} + \text{Delete})$
$+ m \times \text{Decrease}.$

Implementation	$\underbrace{\text{Insert} \quad \text{DeleteMin}}_{n}$		DecreaseKey	Dijkstra's Runtime
Array	$O(1)$	$O(n)$	$O(1)$	$O(n^2 + m) = O(n^2)$

Priority Queues and Dijkstra

Implementation	Insert	DeleteMin	DecreaseKey	Dijkstra's Runtime
Array	$O(1)$	$O(n)$	$O(1)$	$O(n^2 + m) = O(n^2)$
Binary heap	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O\big((n + m)\log(n)\big)$

Priority Queues and Dijkstra



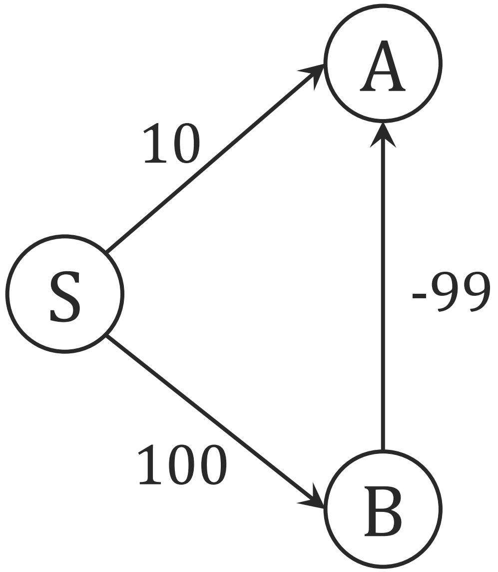
Implementation	**Insert**	**DeleteMin**	**DecreaseKey**	**Dijkstra's Runtime**
Array	$O(1)$	$O(n)$	$O(1)$	$O(n^2 + m) = O(n^2)$
Binary heap	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O\big((n+m)\log(n)\big)$
Fibonacci heap	$O(1)$	$O(\log(n))$	$O(1)$	$O(n\log(n) + m)$

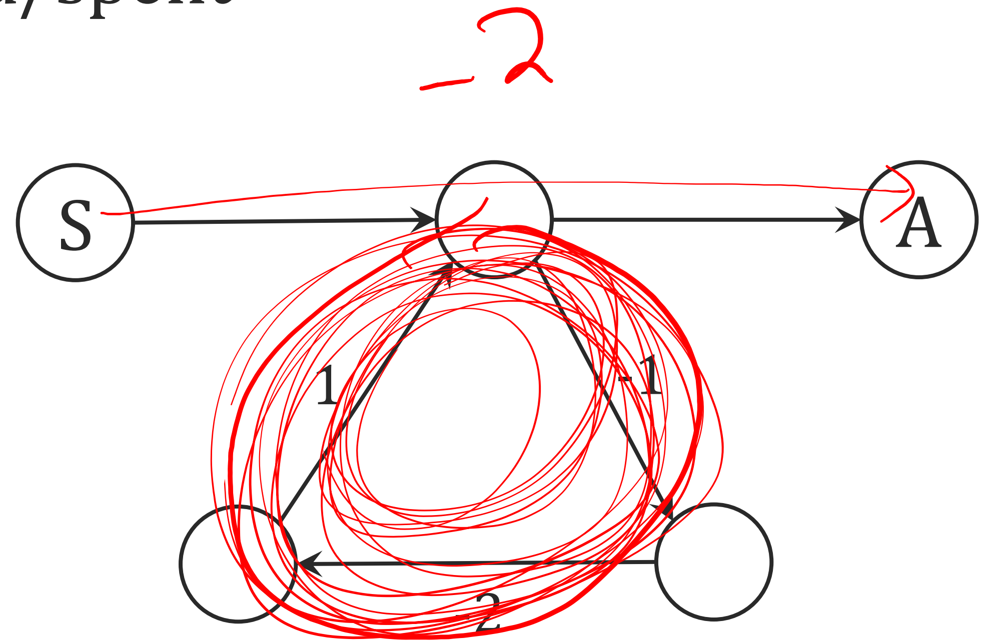
Best known Dijsktra's runtime (2004): $O(n \log\log(n) + m)$

Negative Weights

Sometimes there are negative weights on graphs:

- Instead of total cost, recording cost saved/spent





SSSP is well-defined if no cycle has negative length.

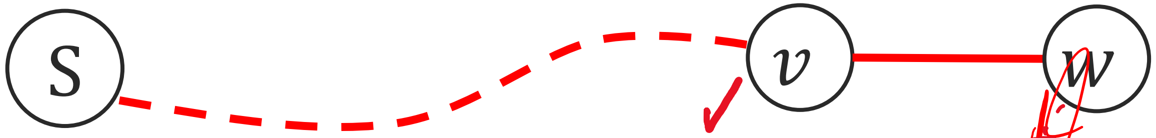
Reviewing the Update Method

1. Update is "safe": $dist[w]$ is an overestimate on the true SP length $d(s, w)$

→ At all times, $dist[w] \geq d(s, w)$ for all $w \in V$.

So S-v SP is also on the same path

2. Suppose the shortest $S - w$ path is the following and that $dist[v] = d(s, v)$.



→ Then, $update(v, w)$ will result in $dist[w] = d(s, w)$.

B/C $\min\{dist(w), \underbrace{dist(v) + \ell(v,w)}_{\text{true SP so } = d(s,w)}\}$

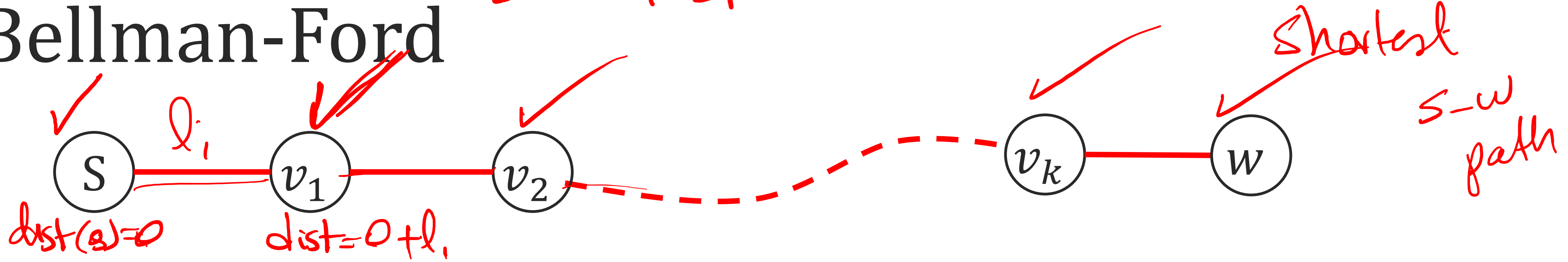
and by 1 $dist(w) \geq d(s,w)$ originally.

$Update(v, w)$

$$\boxed{\begin{aligned} &\text{If } dist[w] < dist[v] + \ell(v, w) \\ &\qquad dist[w] = dist[v] + \ell(v, w) \end{aligned}}$$

Bellman-Ford

• S-v_i SP is also on this path assume.



The following sequence, computes every node's distance from S correctly.

$$\boxed{update(s, v_1)\ \ldots update(v_1, v_2)\ \ldots update(v_2, v_3)\ \ldots\quad \ldots update(v_k, w)}$$

This sequence is a subsequence of iterating over all edges and updating each one, and repeating this $n-1$ times.

Bellman-Ford(G, s)

For i=1, , ..., $n-1$

For all $(u, v) \in E$

$update(u, v)$

Runtime of Bellman-Ford:

- $O(nm)$ updates
- Each update is $O(1)$.
- Best SSSP runtime for arbitrary edge weights.

Wrap up

BFS versus DFS!

BFS, Dijkstra, Bellman-Ford
→ All good for single-source shortest path problem.
→ Dijkstra handles positive weights, but less efficient than BFS
→ Bellman-Ford can handle negative weights, but less efficient than Dijkstra

Next time

- Greedy Algorithms