

CSCI 256 Lecture 8:
Single-Source Shortest Paths

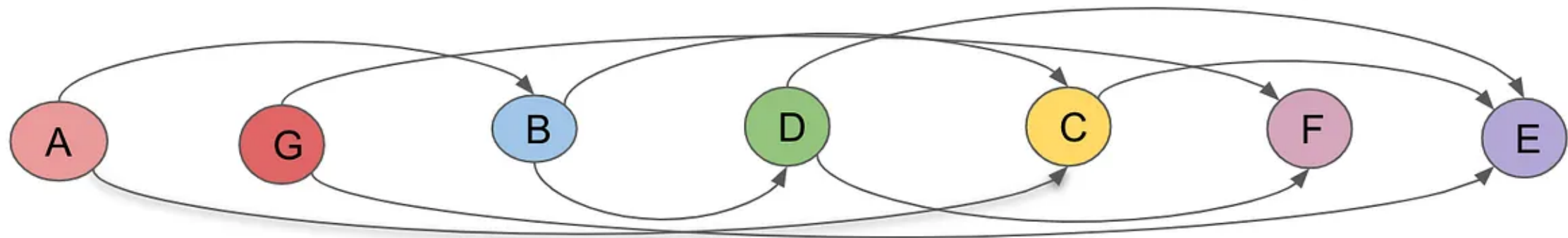
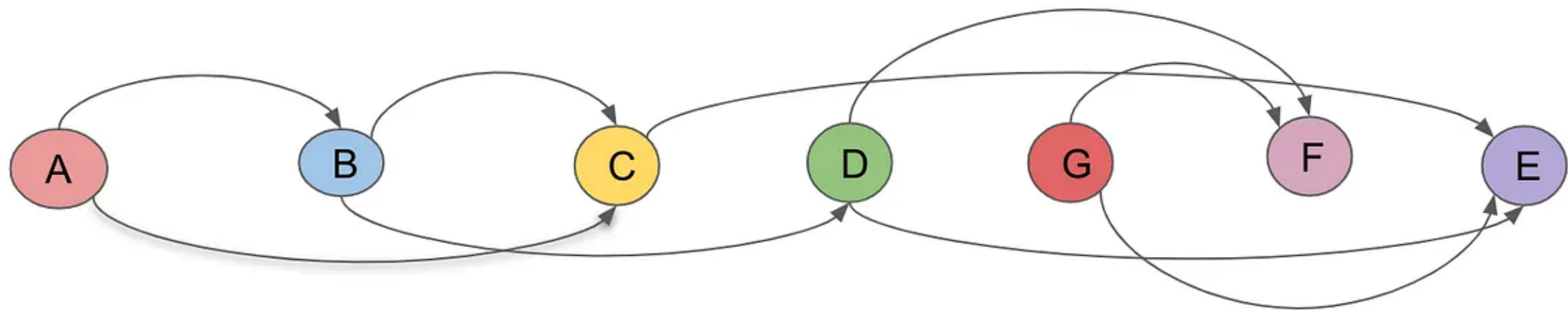
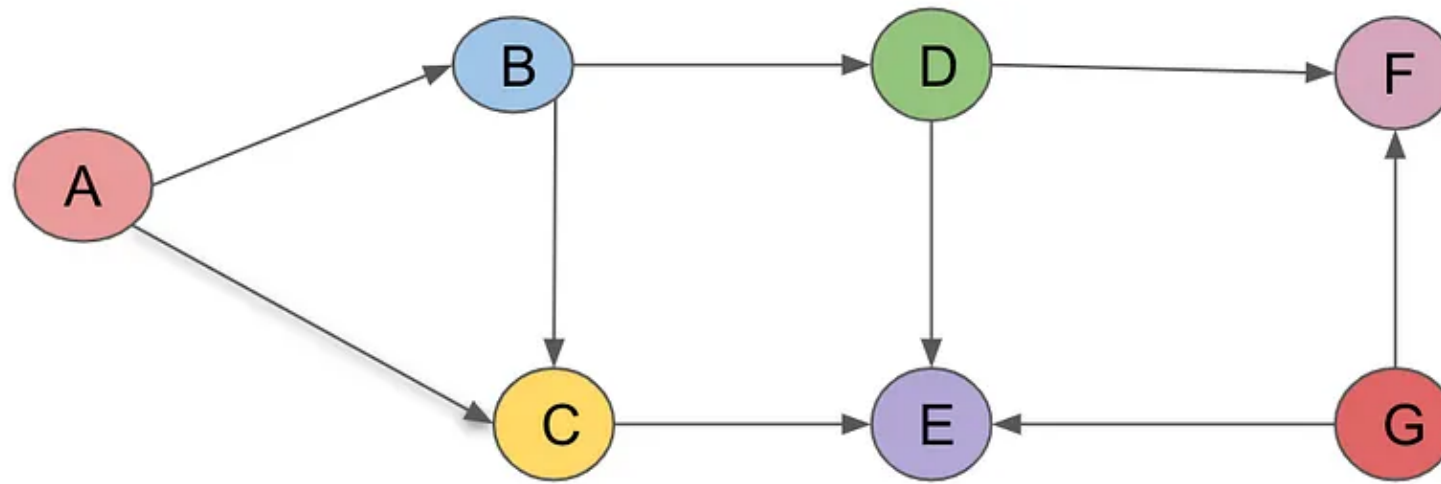
Shikha Singh

Announcements

- Hand in **Exercise #6**, Pick up **Exercise #7**
- Assignment 1 graded feedback returned
- **Assignment 2** due this Wed at 10 pm
- First in-class test is on Mon, Oct 5
 - Will cover topics up until Assignment 2 and Exercise 5
 - Chapter 1.1, 2.1-2.3 and 3 in KT
 - Stable matching, Asymptotics, Graphs, BFS, DFS and their applications, directed graphs and topsort
 - All content up to but not including Dijkstra's
 - Practice test will be released soon
- Office hours: 1.30-3 pm today!

Last Time and Today

- Discussed connectivity in undirected and directed graph
- Definition of topological ordering
- Today: topological ordering algorithm
 - Connection to DFS
- Move on to shortest path in weighted graphs
 - Extend BFS to Dijkstra's



Topological Sort

Topological Sorting Algorithm

```
TopologicalSorting(G)  $\triangleleft$   $G = (V, E)$  is a DAG
    while V is not empty do
        Find a vertex  $v \in V$  with  $\text{indeg}(v) = 0$ 
        # if such a vertex does not exist:
        #   report that G has a cycle
        Add v to the front of L
        Delete v (and its edges) from G
```

Analysis:

- Correctness?
- Running time?

Finding a Topological Ordering

Claim. Every DAG has a vertex with in-degree zero.

Proof. [By contradiction] Suppose every vertex has an incoming edge.

- Pick any vertex v , there must be an edge (u, v) .
- Walk backwards following these incoming edges for each vertex
- After $n + 1$ steps, we must have visited some vertex w twice (why?)
- Nodes between two successive visits to w form a cycle ($\Rightarrow \Leftarrow$) ■

Correctness. In-degree zero vertex is also safe to put first

- Cannot contribute to any "back edges"
- Can extend this idea inductively to a full proof

Topological Sorting Algorithm

Analysis (Correctness). Base case: $n = 1$, no edges, the vertex itself forms topological ordering

- Suppose algorithm is correct for any DAG with less than n vertices
- Consider an arbitrary DAG on n vertices
 - Must contain a vertex v with in-degree 0 (we proved it)
 - Delete v and its outgoing edges to create G'
 - G' is a DAG with less than n vertices and thus has a topological ordering (say $u_1, u_2, \dots, u_{n-1}$)
- Then $v, u_1, u_2, \dots, u_{n-1}$ is a valid topological ordering of G because v has no incoming edges from $u_1, \dots, u_{n-1}$ ■

Topological Sorting Algorithm

Running time

TopologicalSorting(G) $\triangleleft$ $G = (V, E)$ is a DAG

while V is not empty do

Find a vertex $v \in V$ with $\text{indeg}(v) = 0$

if such a vertex does not exist:
report that G has a cycle

Add v to the front of L

Delete v (and its edges) from G

Takes $O(\text{outdegree}(v))$ time

How do we do this fast?

Running time

- **Idea:** Maintain an in-degree count array $ID[u]$ for each vertex u and a queue (or stack) S of in-degree 0 vertices
- Updates: When v is deleted, decrement $ID[u]$ for each neighbor u ; if $ID[u] = 0$, add u to S :
 - $O(\text{outdegree}(v))$ time to decrement neighbor in-degrees
 - $O(1)$ time to check and add to S
- Total time for previous step over all vertices:
 - $\sum_{v \in V} O(\text{outdegree}(v)) = O(n + m)$ time
- Topological sorting takes $O(n + m)$ time: linear time!

DFS Application: TopSort

Topological Ordering by DFS

- **Idea:** Vertex to finish first in DFS: *sink* in the graph
- Call DFS and maintain finish times of all vertices
 - Finish(u): time DFS(v) completed for all neighbors of u
- As long as there are unmarked vertices, repeat DFS from them
- Return the list of vertices **in reverse order of finish times**
 - Vertex finished last will be first in topological ordering
- **Claim.** If a DAG G contains an edge $u \rightarrow v$, then the finish time of u must be larger than the finish time of v .
 - u is finished only after all its outgoing neighbors are finished

Topological Ordering: DFS

- Can use DFS to find cycle and if it doesn't exist, return topological order

DFS–Cycle(u):

Set status of u to marked # discovered u

for each edges (u, v):

if v's status is unmarked:

DFS(v)

else if v is marked but not finished

report a cycle!

mark u finished # place u at the front of L

done exploring neighbors of u

Extending BFS:

Shortest Path in Weighted Graphs

BFS Tree in Undirected Graphs

- In an undirected graph BFS tree from root s gives the shortest path tree from s to all nodes reachable from s
- Here the distance/cost of path is **the number of edges**
- What if there was a "cost/weight" associated to traverse an edge?
 - Some edges are cheaper than others to take
 - Maps traffic in a road or computer network

Single-Source Shortest Paths

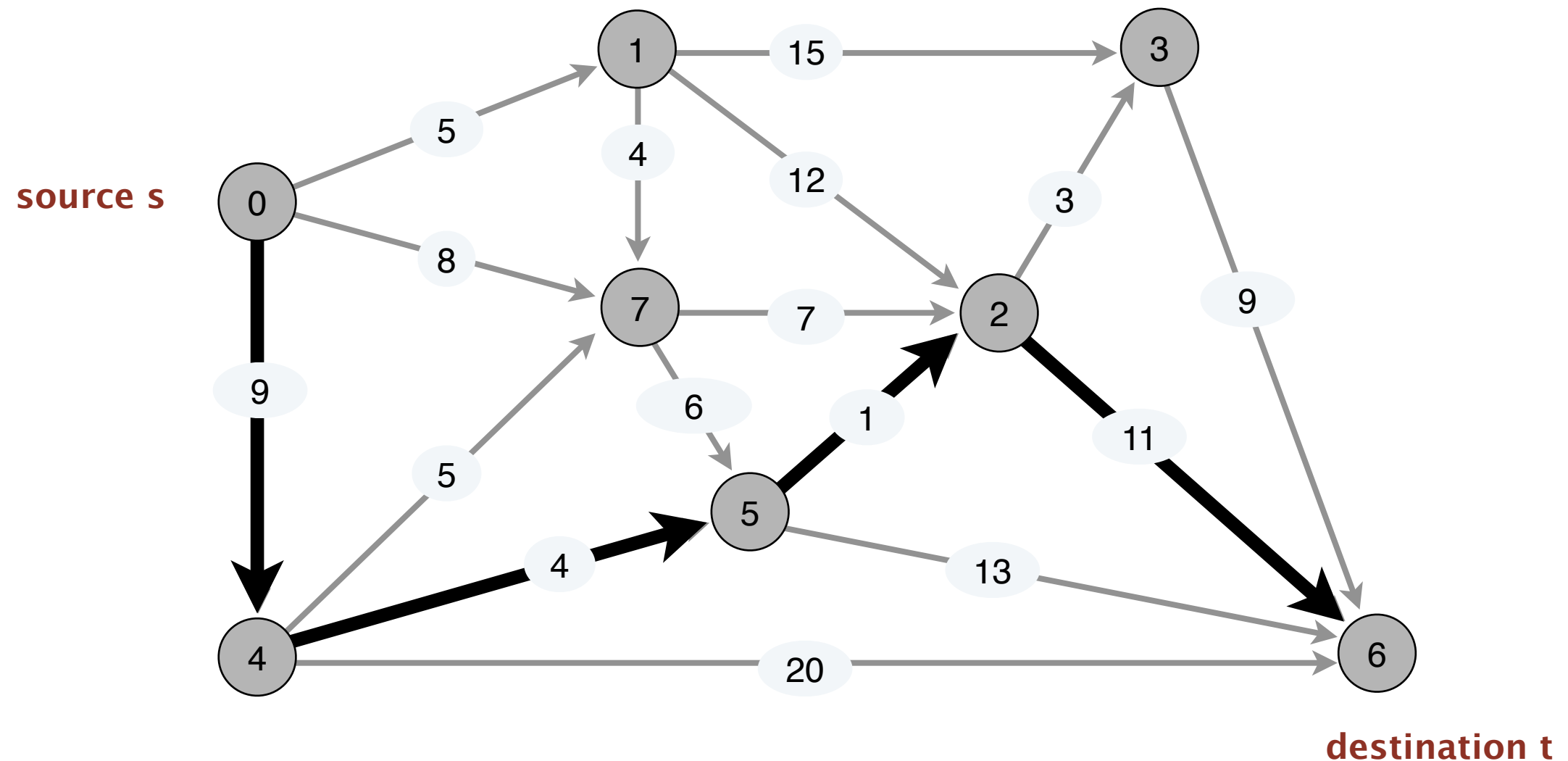
Problem. Given a directed graph $G = (V, E)$ with **non-negative edge weights**: that is, each edge $e \in E$ has a weight $w(e) \geq 0$ and a vertex s , find the shortest path from s to all vertices in V .

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

- What if there is only one path from s to t ?
- We only have a choice when multiple paths exist

Shortest Path from s to t

Many possible $s - t$ paths



shortest path distance = 9 + 4 + 1 + 11 = 25

Shortest Paths Applications

- Map routing
- Path planning and navigation
- Image segmentation and resizing
- Typesetting in LaTeX
- Scheduling, routing of operators
- Internet routing
- It is so important that we will revisit shortest paths when we study dynamic programming!

Dijkstra's Algorithm

- Extends BFS to weighted graphs
- Computes the shortest path tree from s to all vertices
- Dijkstra's algorithm maintains a **shortest path tree** rooted at s
 - Maintains a "best-so-far" distance estimate $d(u)$ of the path from s to a node u not yet in the tree
 - It greedily selects the next vertex to add to the tree based on lowest distance estimate
- First example of a **greedy optimization algorithm!**
 - Locally optimal decisions

Dijkstra's Algorithm

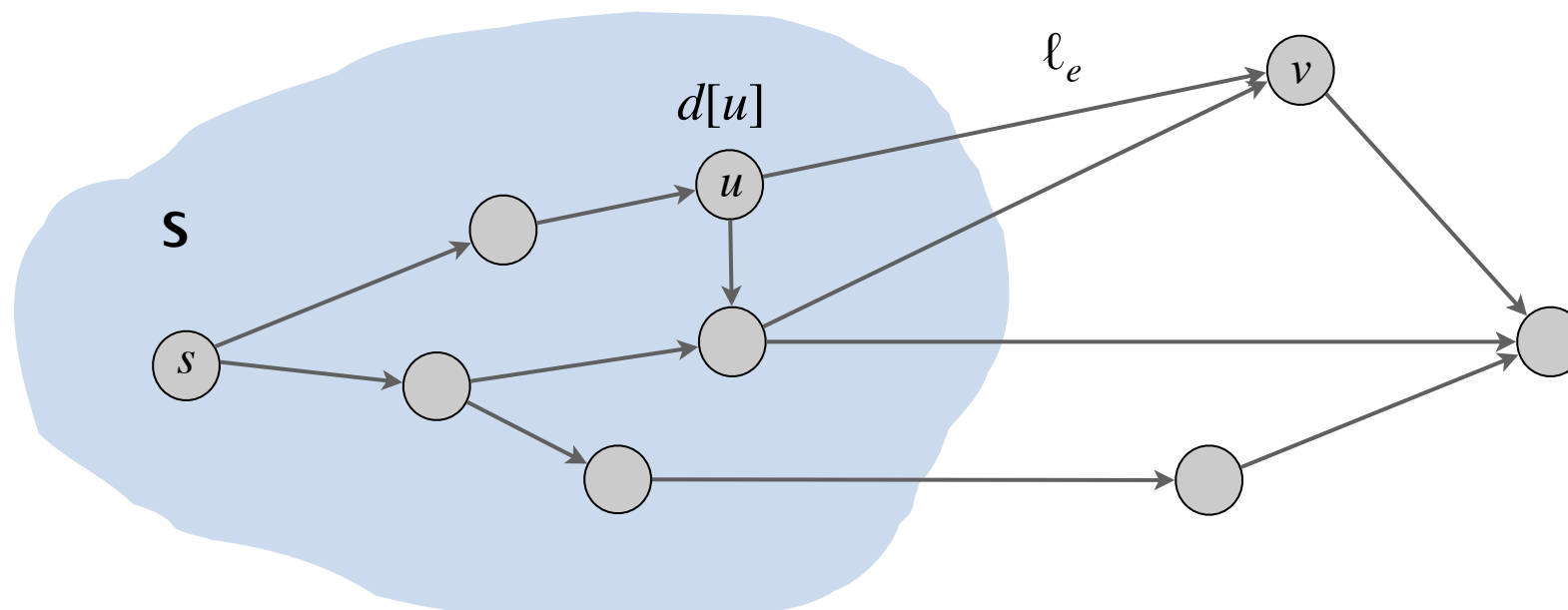
Greedy approach. Maintain a set of explored nodes S for which algorithm has already found $d[u] = \text{shortest } s \rightsquigarrow u \text{ path distance}$

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

which minimizes $\min_{e=(u,v):u \in S}$

$$d[u] + \ell_e$$

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



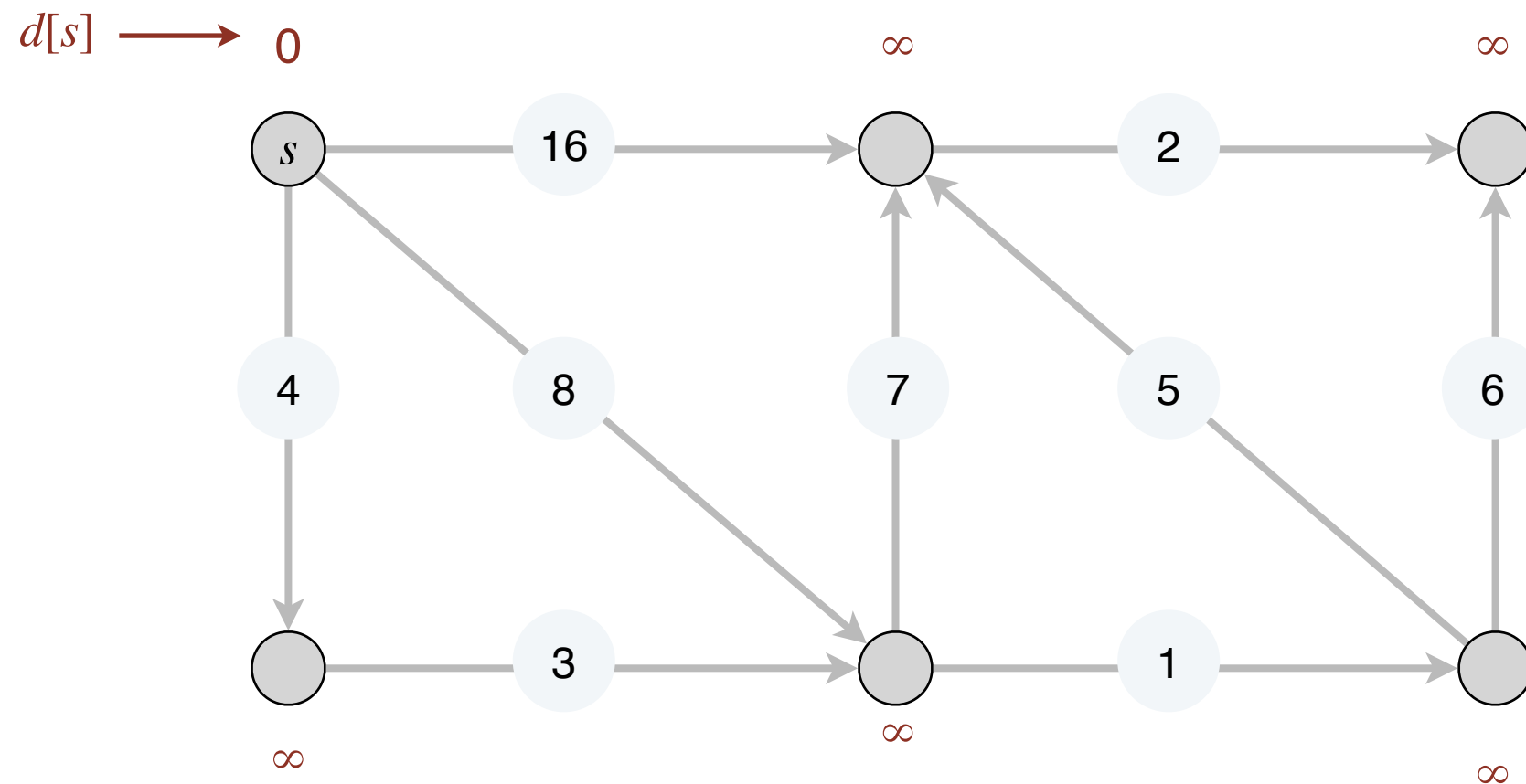
Dijkstra's Algorithm

```
procedure Dijkstra(G, s)      # G = (V,E) is connected
    S = {s}, d[s] ← 0        # initialize
    for all neighbors v of s:
        d[v] ← w(s, v)
    for all non-neighbors v of s:
        d[v] ← ∞ # no path found
    while S not equal to V:
        select v from V - S with min d[v]
        Add v to S
        for each neighbor u of v in V - S:
            if d[v] + w(v, u) < d[u]: # better path found
                d[u] = d[v] + w(v, u) # update estimate
```

Dijkstra's Demo

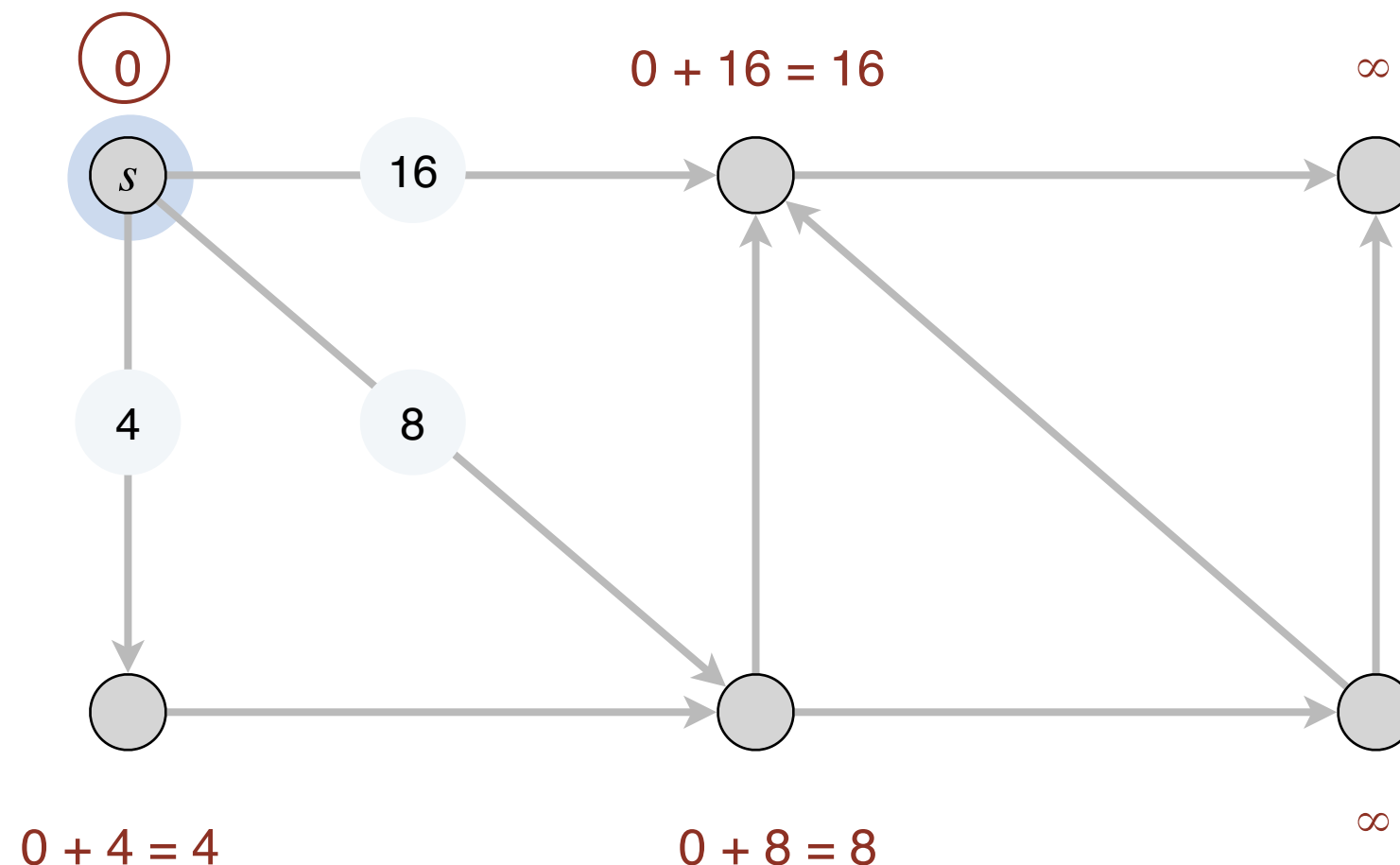
Initialization.

- For all $v \neq s$: $d[v] \leftarrow \infty$.
- For all $v \neq s$: $pred[v] \leftarrow null$.
- $S \leftarrow \emptyset$ and $d[s] \leftarrow 0$.



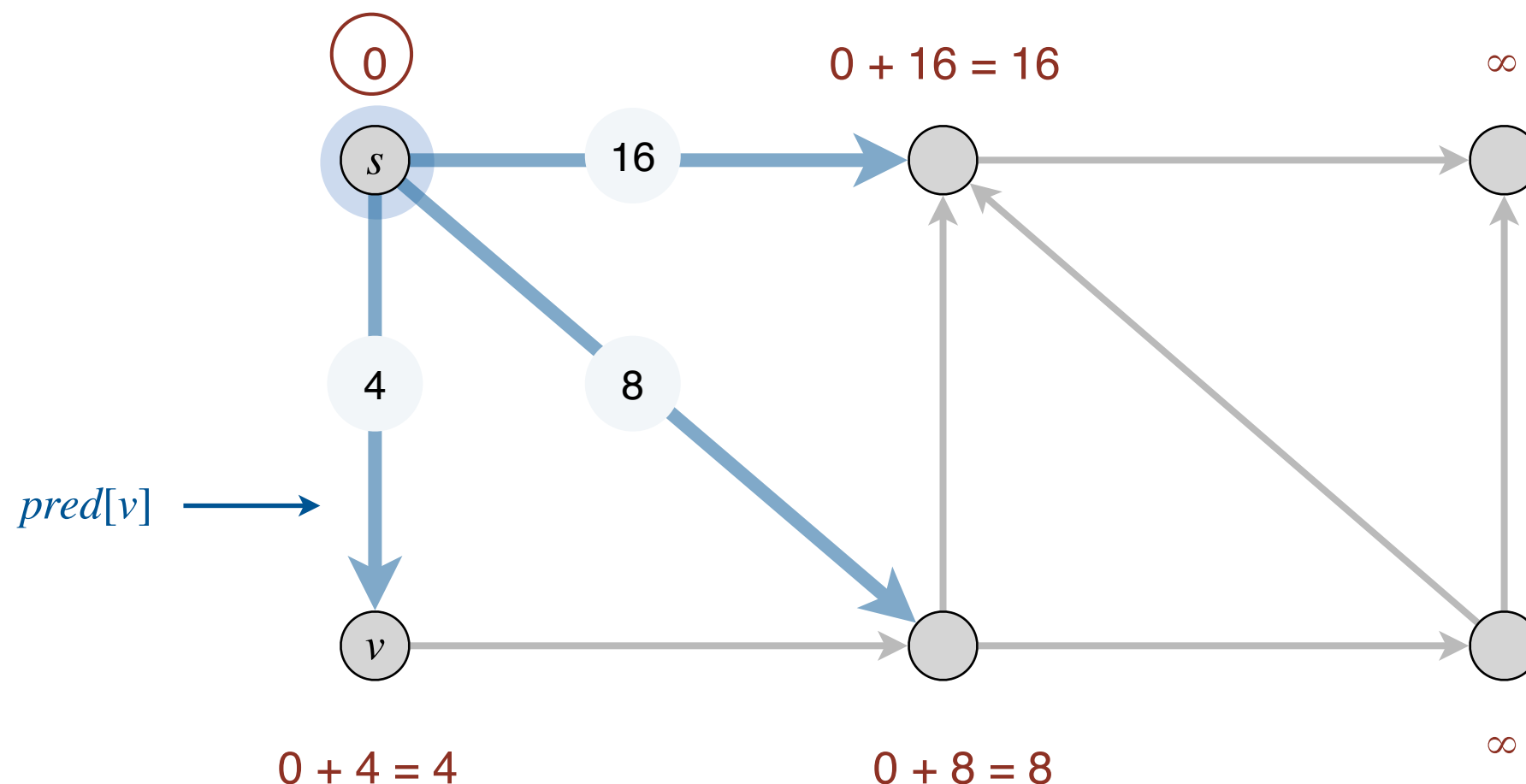
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



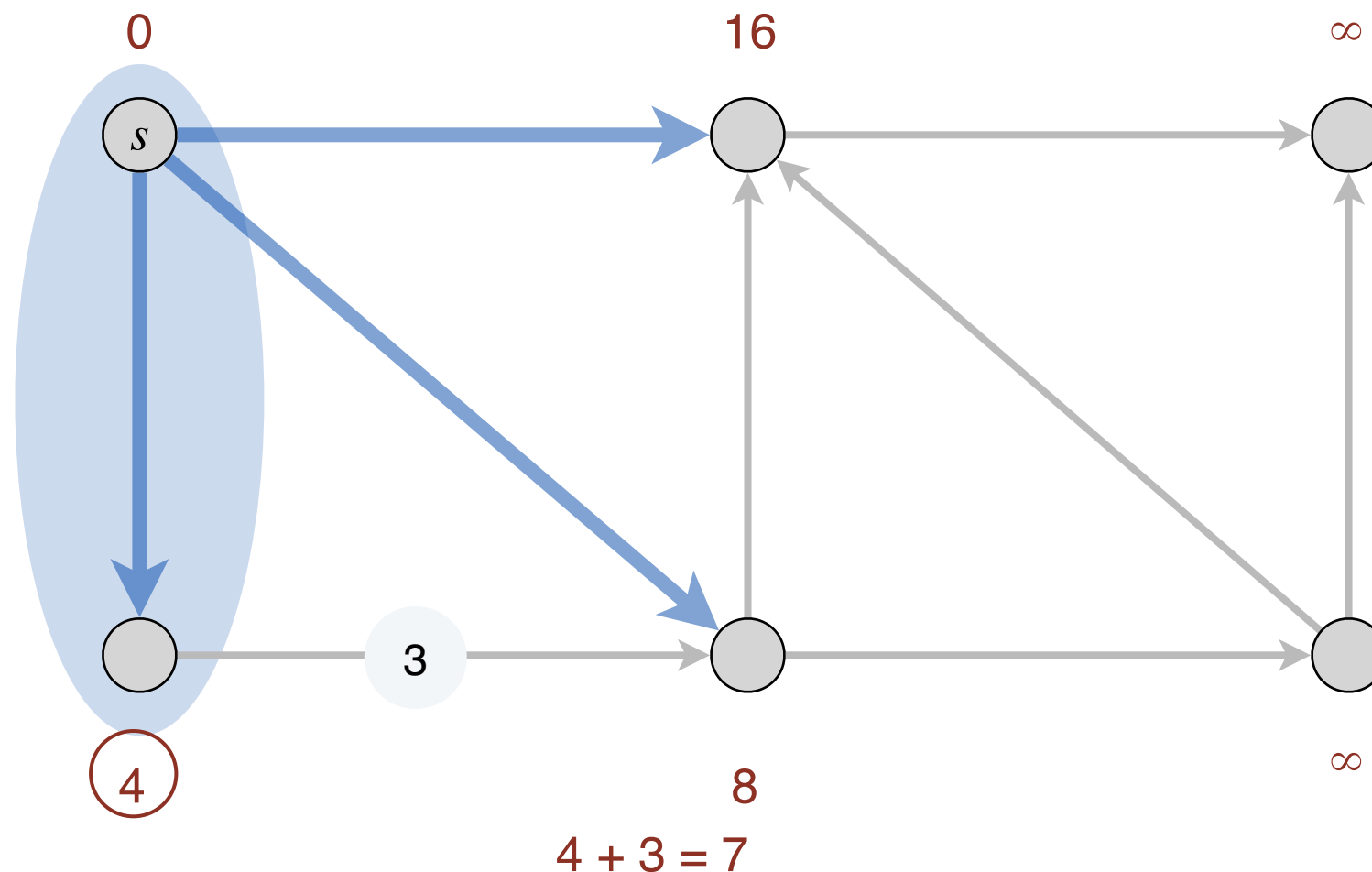
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



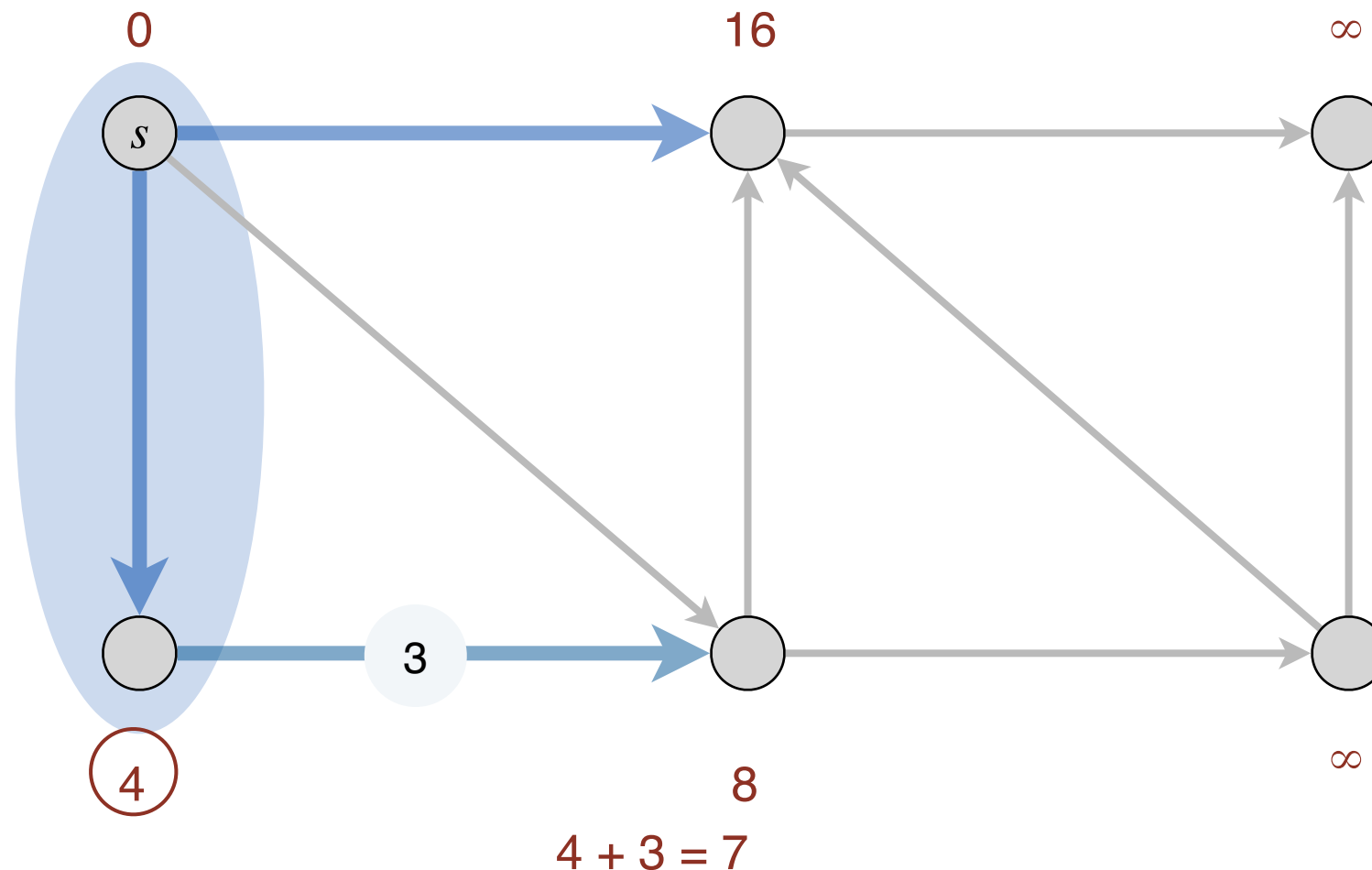
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



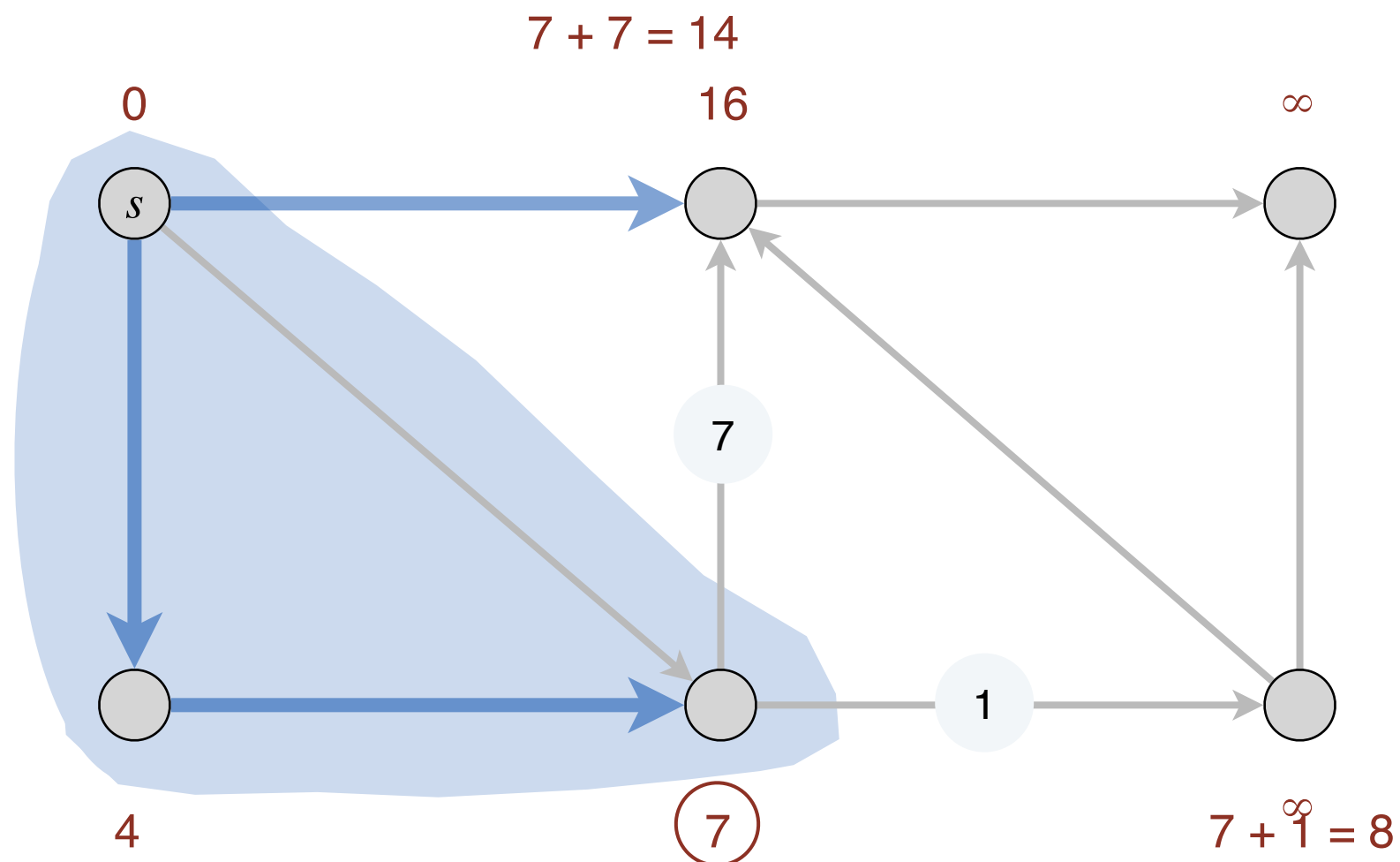
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
$$d[v] \leftarrow d[u] + \ell_e$$
$$pred[v] \leftarrow e \text{ (remember path to } v)$$



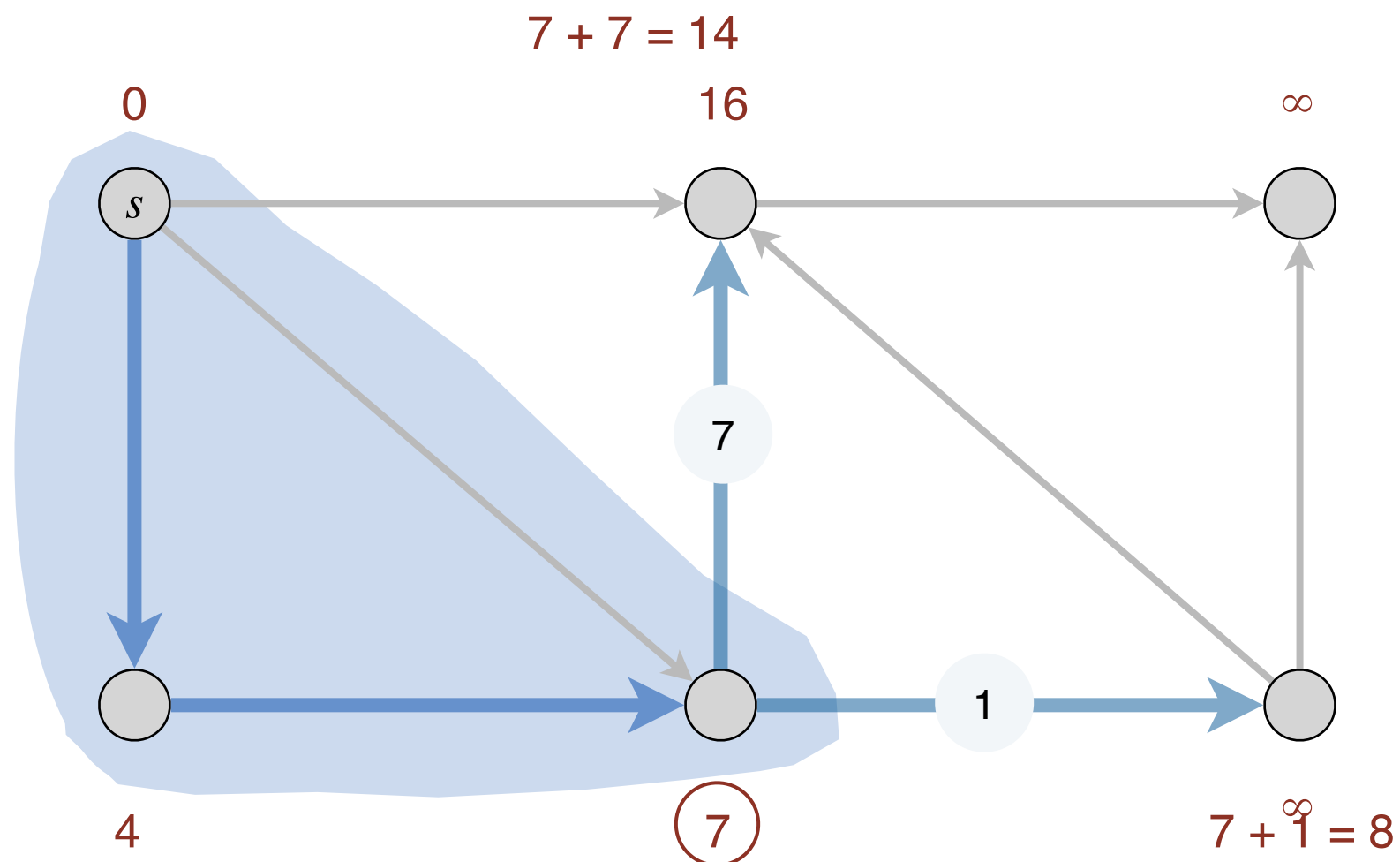
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



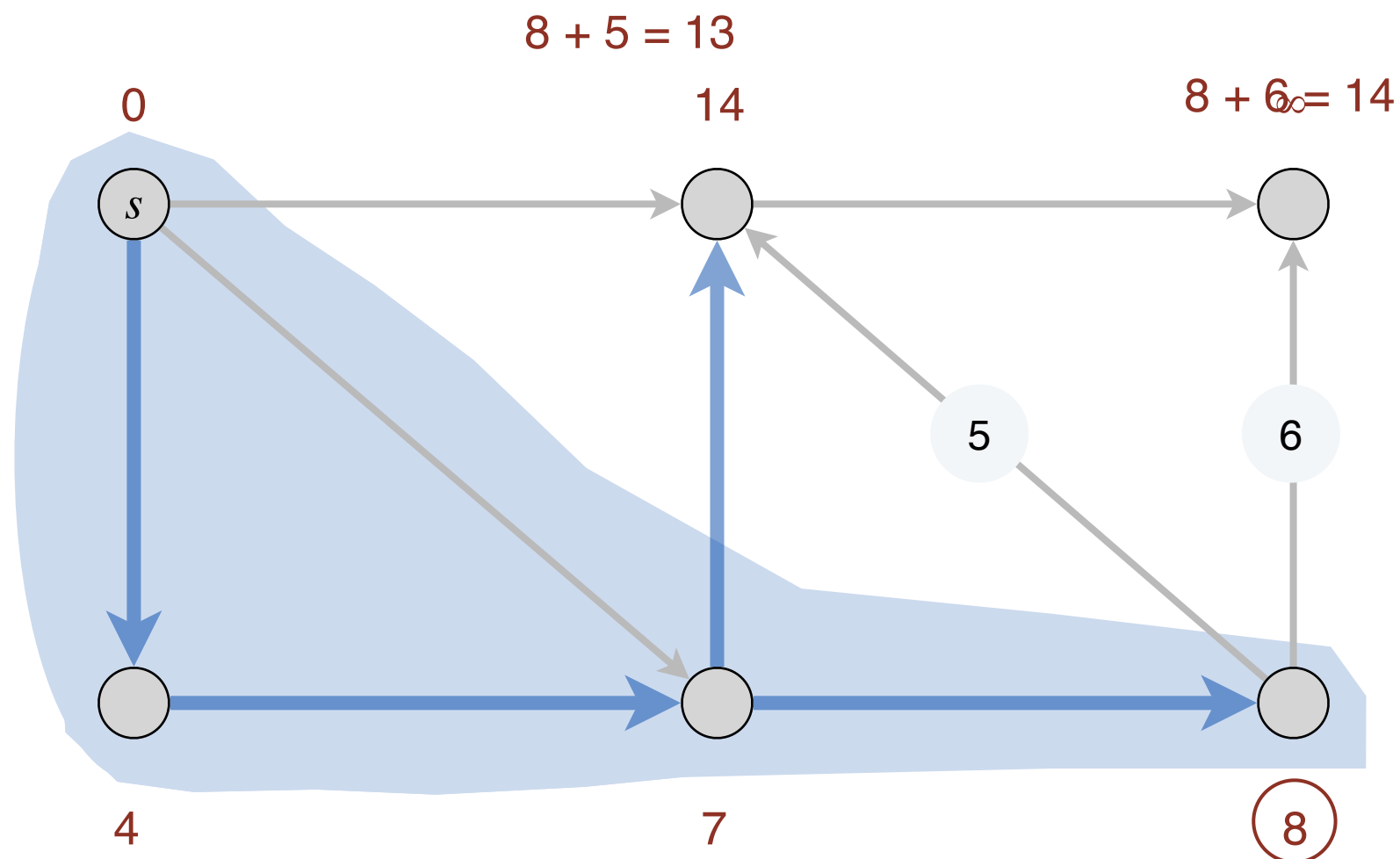
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



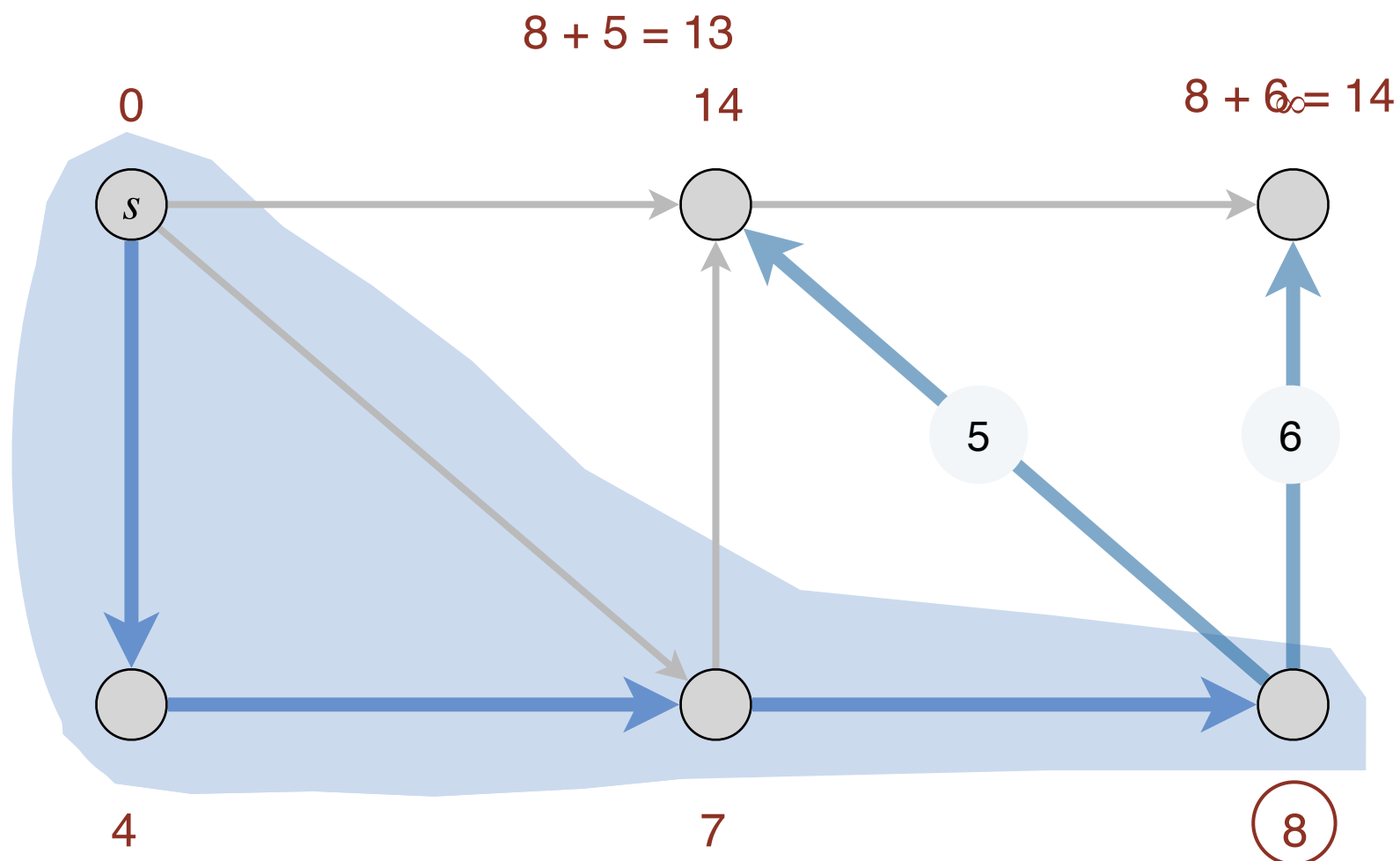
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



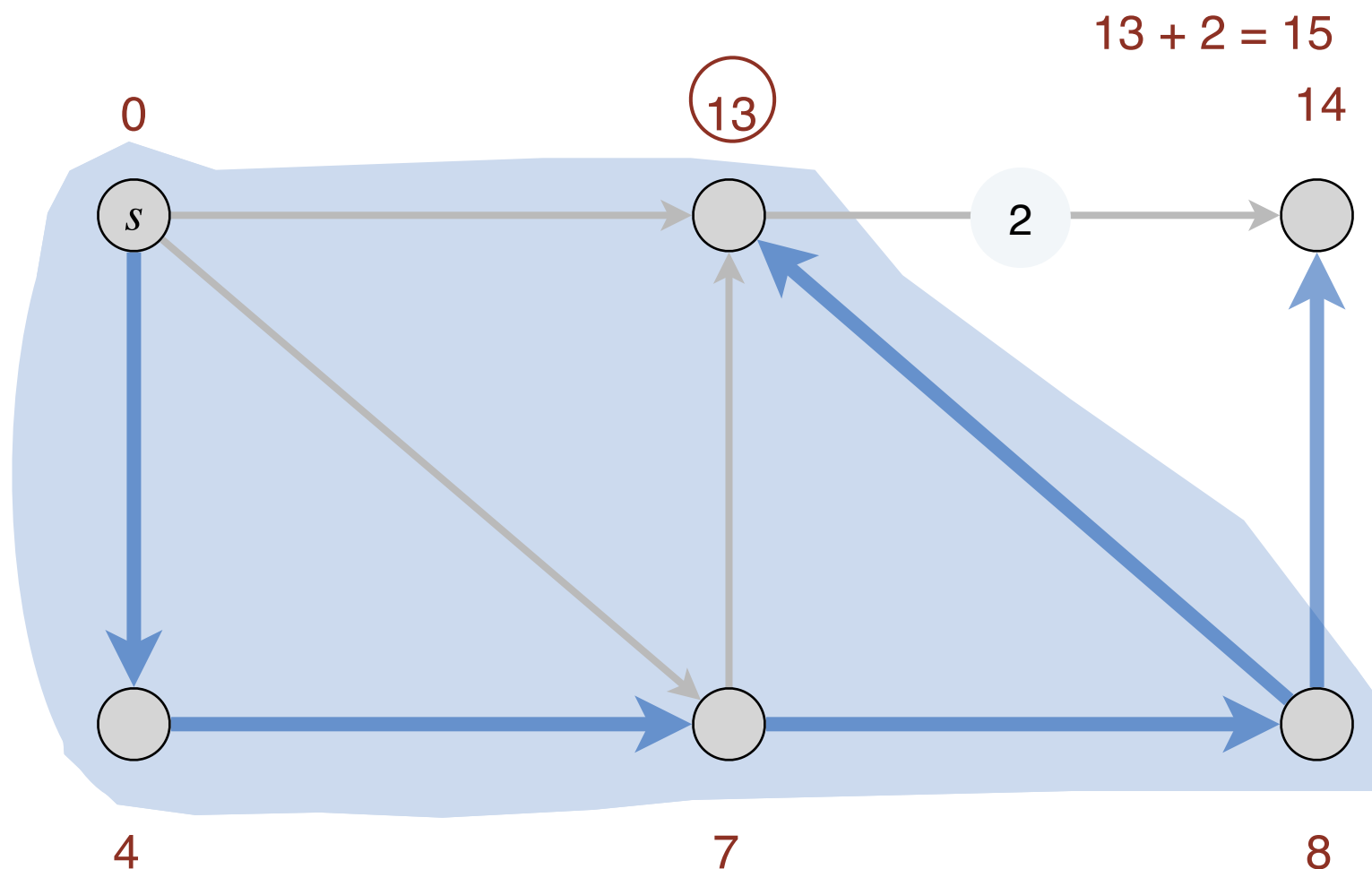
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
$$d[v] \leftarrow d[u] + \ell_e$$
$$pred[v] \leftarrow e \text{ (remember path to } v)$$



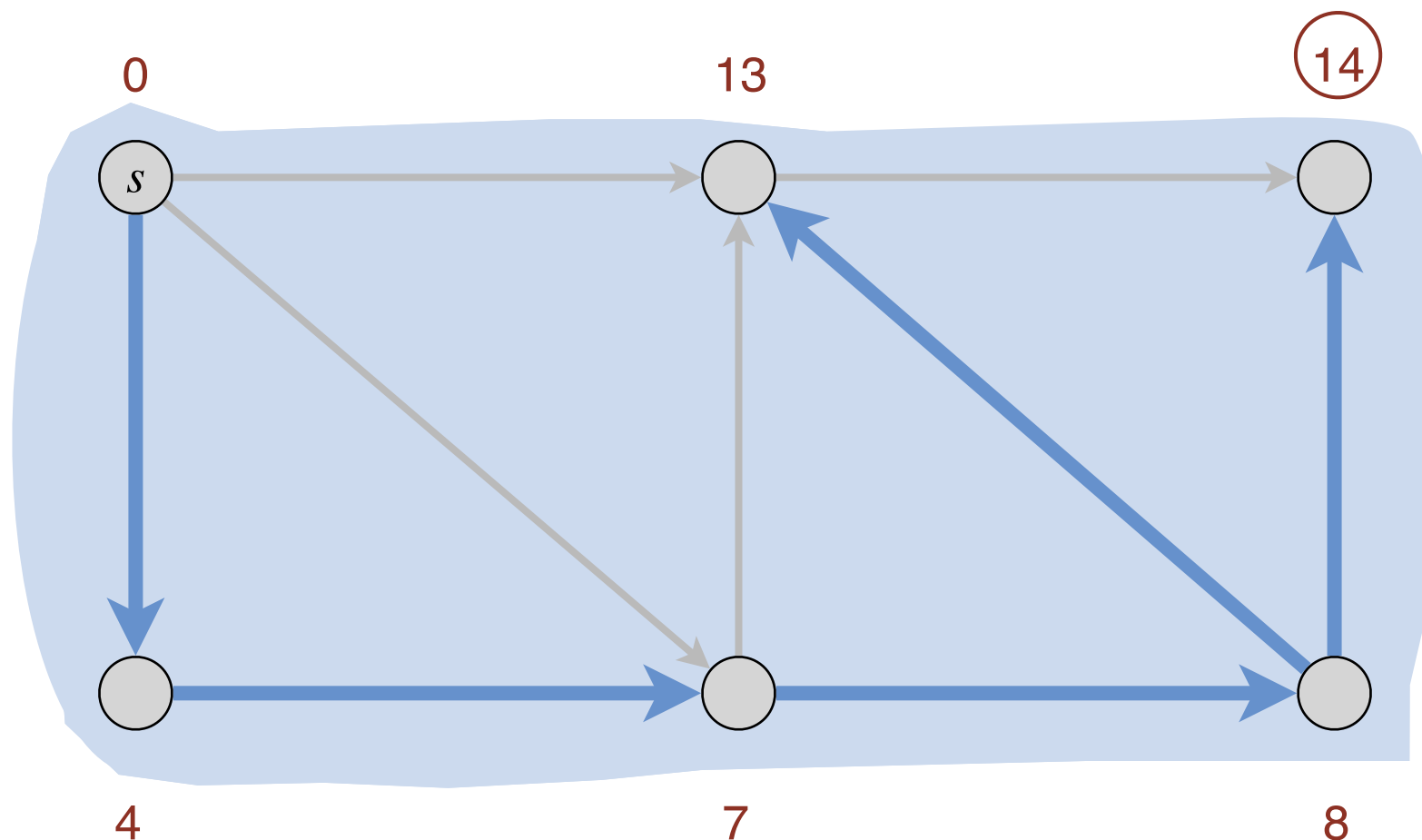
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



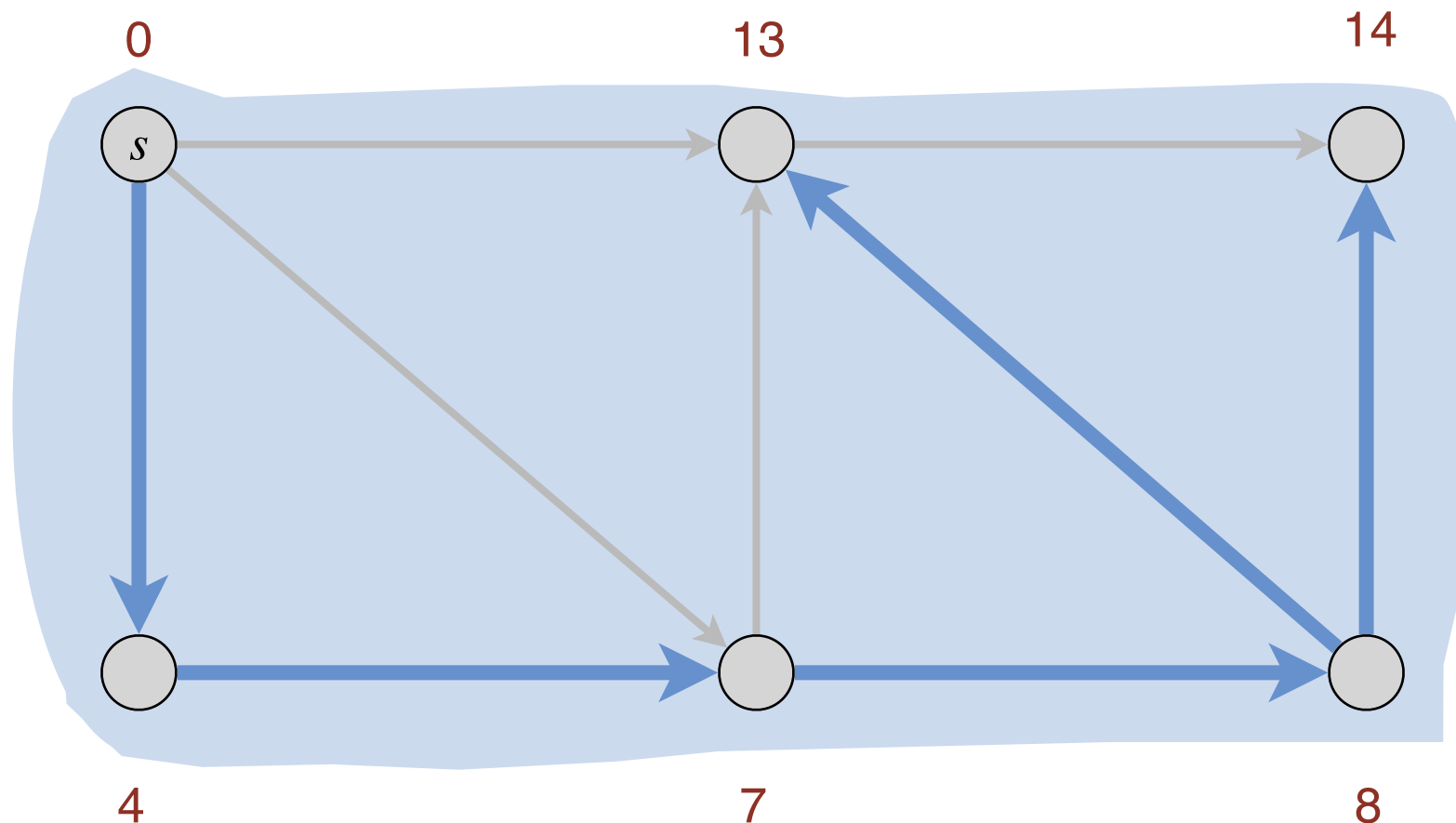
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



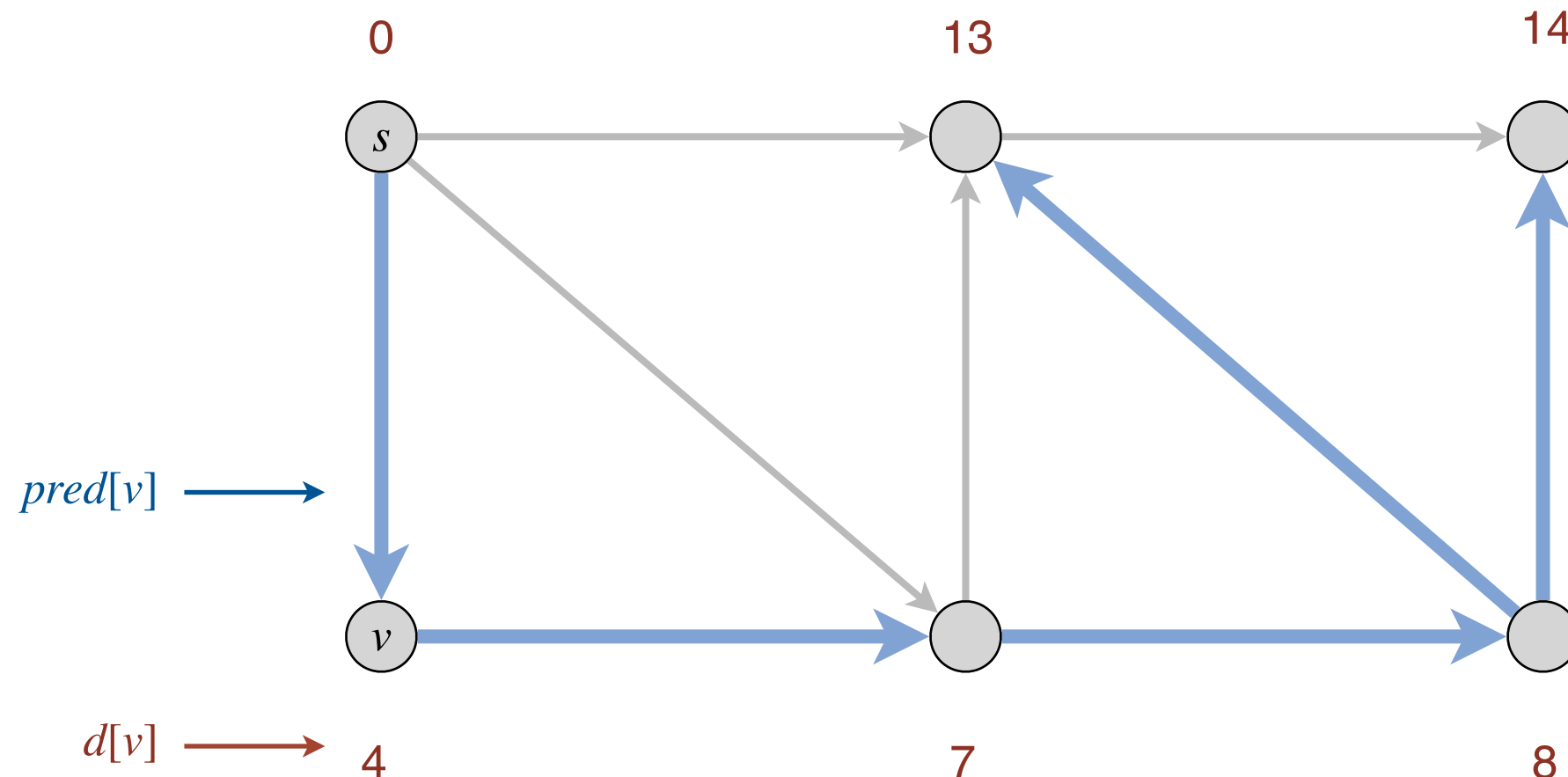
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S .
- For each edge $u \rightarrow v$, if $d[v] > d[u] + \ell_e$ then
 $d[v] \leftarrow d[u] + \ell_e$
 $pred[v] \leftarrow e$ (*remember path to v*)



Termination.

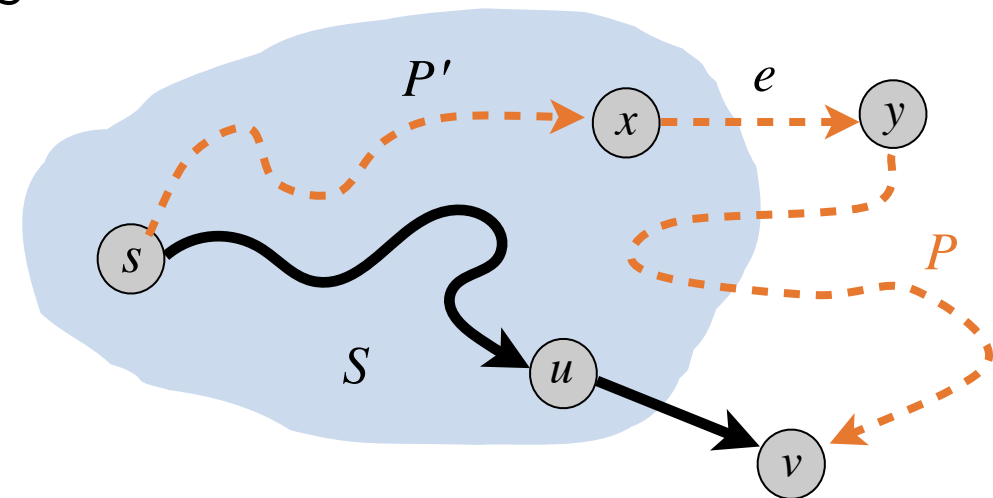
- $d[v]$ = length of a shortest $s \rightsquigarrow v$ path.
- $pred[v]$ = last edge on a shortest $s \rightsquigarrow v$ path.



Dijkstra's Correctness: High level

Main Idea: Algorithm has already computed shortest paths to every vertex that is in S

- Any time it grows S , it looks at all edges "crossing" the blue frontier
- All these edges have already contributed to updating d estimates
 - One end point is already in S !
 - When it was added, these edges were checked for updates
- Picking the node v with min d value means if there were multiple paths leaving the frontier and going to v , algorithm picks the lowest-cost distance one!



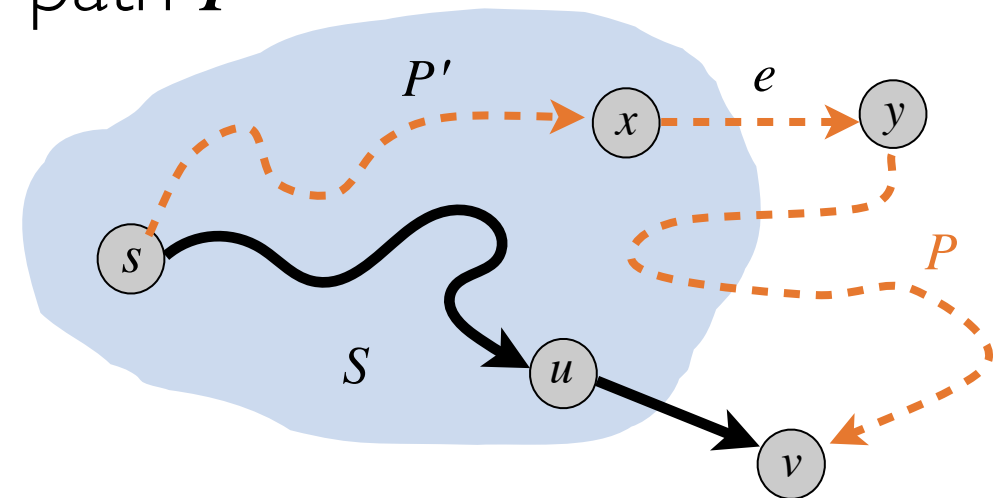
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [By induction on $k = |S|$]. Base case: $k = 1, S = \{s\}$ and $d[s] = 0$. Assume invariant holds for some $k > 1$.

We now grow S to size $k + 1$: let v be next node added to S

- Goal is to show that $d[v]$ is the shortest s - v path distance
- If this is the only s - v path, nothing to prove
- Now suppose there is an another path s - v path P
 - Need to show $w(P) \geq d[v]$



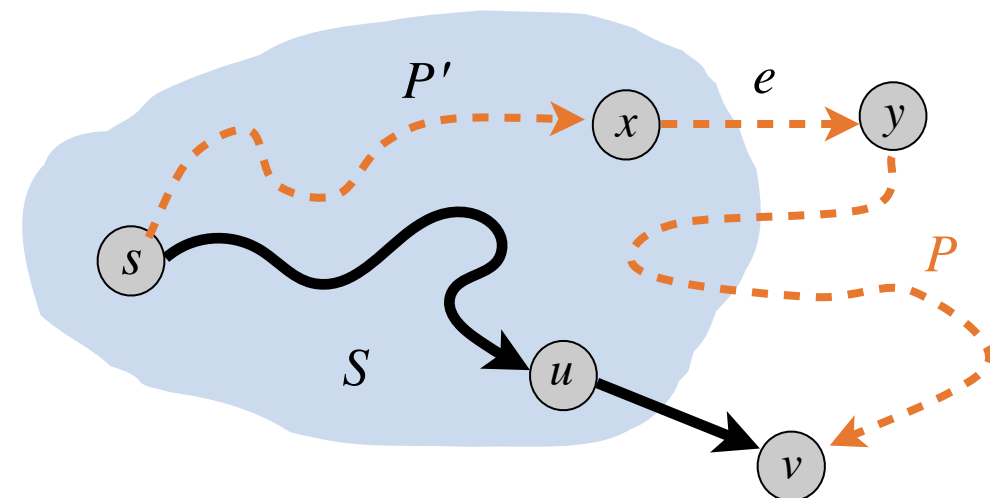
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [Continued] Consider another s - v path P

- Let $e = (x, y)$ be the first edge along P that leaves S
- At the $(k + 1)^{\text{st}}$ step Dijkstra chose v instead of y :

$$d[y] \geq d[v]$$

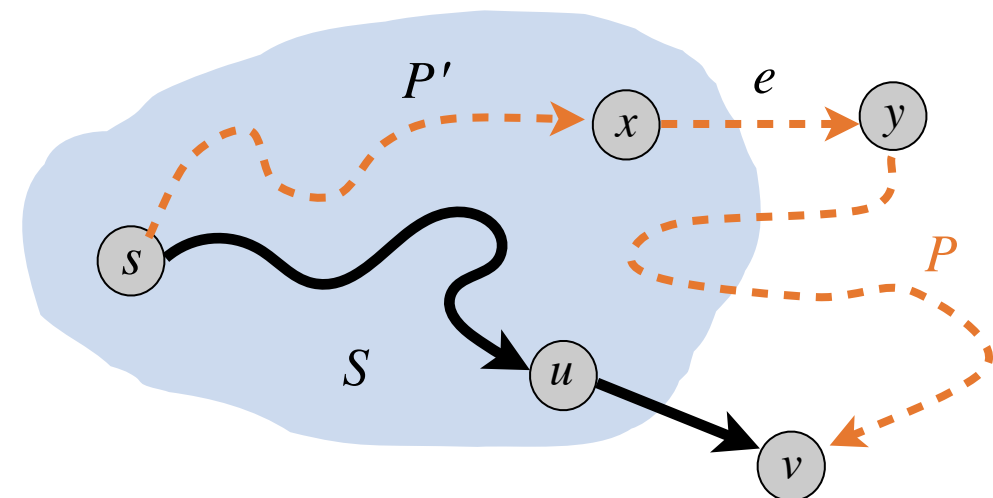


Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [Continued] Consider another s - v path P

- Let $e = (x, y)$ be the first edge along P that leaves S
- When x was added to S , $d[y]$ was updated: $d[x] + w_e \geq d[y]$



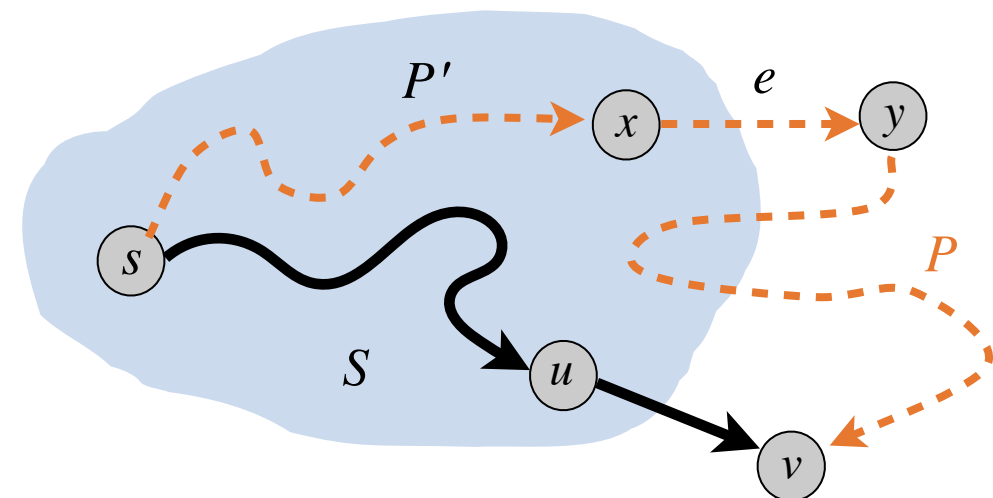
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [Continued] Consider another s - v path P

- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath in P from s to x
- Then, by the inductive hypothesis:

$$w(P') \geq d[x]$$



Dijkstra's Algorithm: Correctness

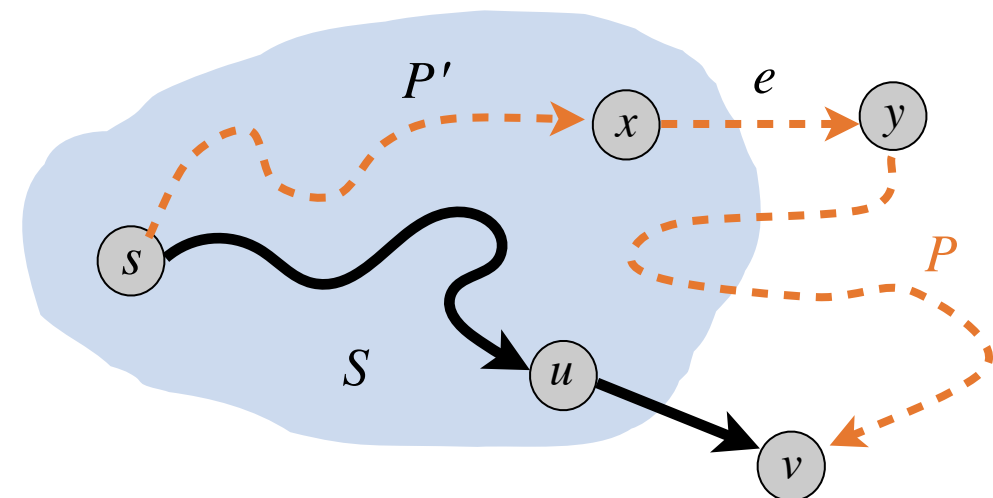
Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [Continued]

- Putting it all together we have:

Dijkstra chose to add v instead of y

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



Dijkstra's Algorithm: Correctness

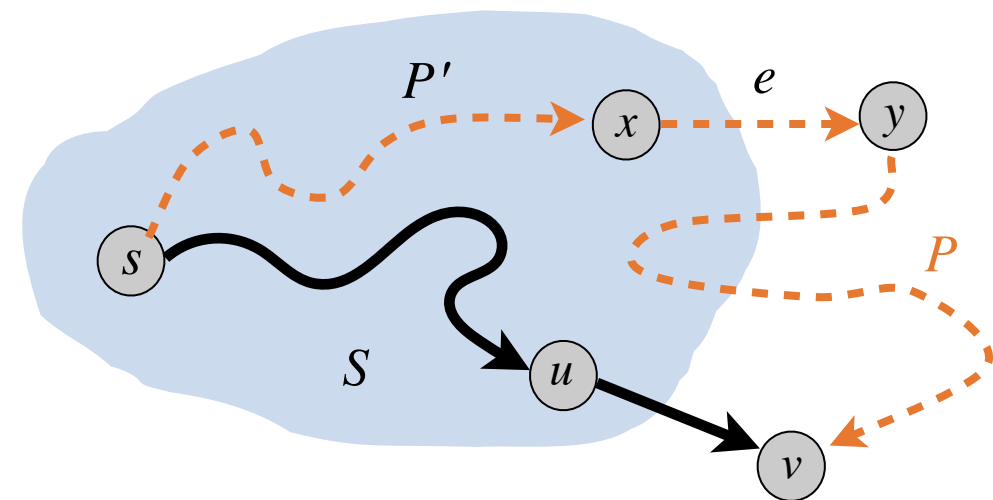
Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [Continued]

- Putting it all together we have:

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

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



Dijkstra's Algorithm: Correctness

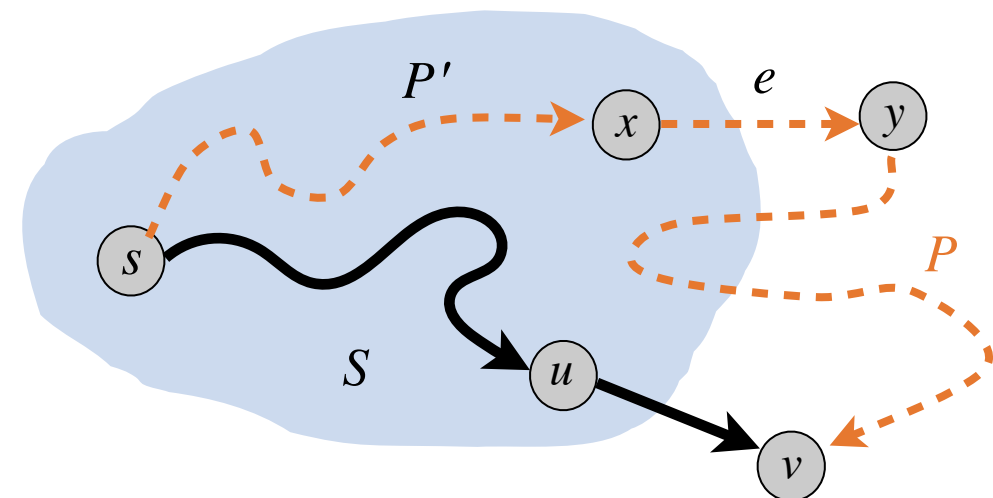
Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [Continued]

- Need to show $w(P) \geq d[v]$, what is the last step?

Inductive Hypothesis

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



Dijkstra's Algorithm: Correctness

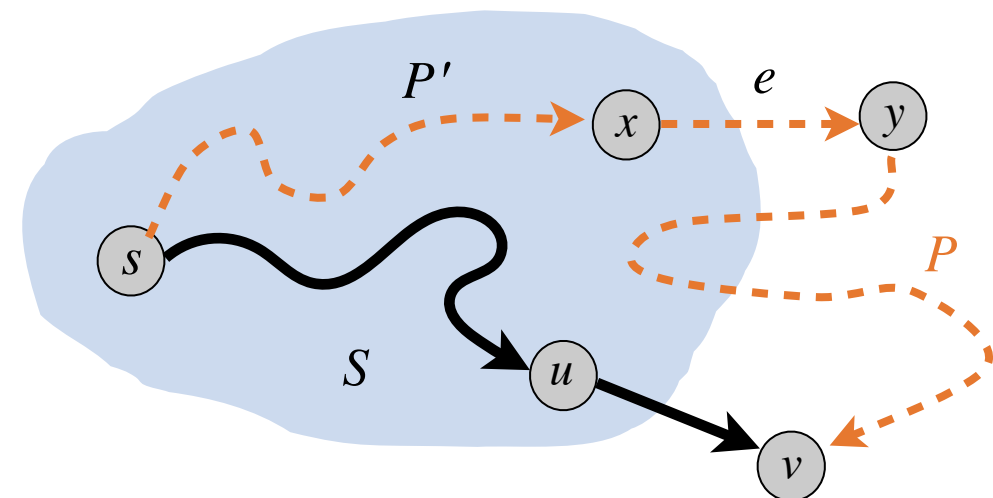
Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [Continued]

- P' is a subpath of P : non-negative weight means it cannot be shorter

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

Non-negative weights



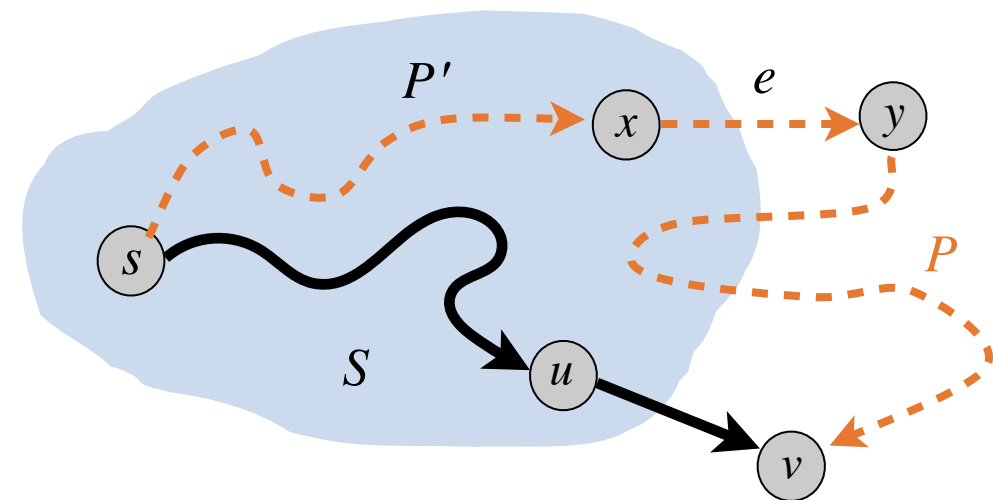
Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is the shortest s - u distance

Proof. [Continued]

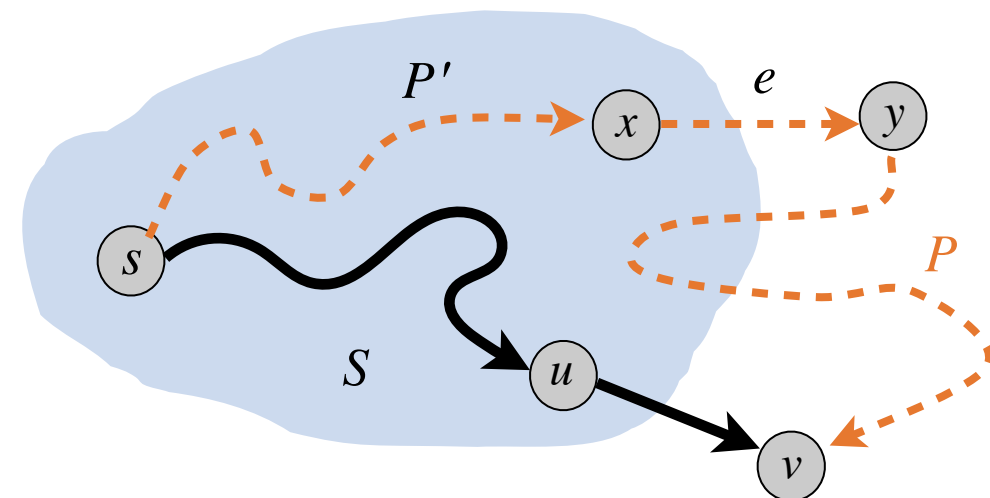
- Since P was an arbitrary path from s to v , we are done!

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



Negative Weights

- Dijkstra only works for non-negative weights
- When weights can be negative:
 - Need a dynamic programming approach (Bellman Ford)
 - Will revisit when we study dynamic programming



Dijkstra's Algorithm: Runtime

```
procedure Dijkstra( $G, s$ )      #  $G = (V, E)$  is connected
     $S = \{s\}, d[s] \leftarrow 0$   # initialize
    for all neighbors  $v$  of  $s$ :
         $d[v] \leftarrow w(s, v)$ 
    for all non-neighbors  $v$  of  $s$ :
         $d[v] \leftarrow \infty$ 
    while  $S$  not equal to  $V$ :
        select  $v$  from  $V - S$  with min  $d[v]$ 
        Add  $v$  to  $S$ 
        for each neighbor  $u$  of  $v$  in  $V - S$ :
            if  $d[v] + w(v, u) < d[u]$ : # better path
                 $d[u] = d[v] + w(v, u)$  # update
```

Every time a node is added to S , check if $d[]$ of all its neighbors need updating

Choose min d node

Dijkstra's Algorithm: Runtime

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

n iterations at most

Can do in $O(n)$ scan

$O(\text{outdegree}(v))$ time

Dijkstra's Algorithm: Runtime

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

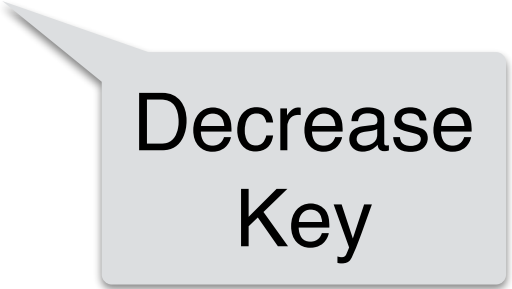
Can implement in $O(n(n + m))$ time

Faster Running Time

- Add at most n vertices to S : outer loop runs $\leq n$ times
- Within each iteration:
 - Find unvisited vertex v with min $d[v]$ among all v that are neighbors of elements in S
 - We could do this in $O(m)$ by checking all edges
 - Update $d[u]$ for each neighbor of v : $O(\text{outdegree}(v))$
 - Maintain set of visited S and unvisited vertices $V - S$:
 - Maintain a tree of edges $(v, (\text{pred}[v]))$
- Naively we can implement in $O(nm)$ time



Extract
Min



Decrease
Key