Lecture 14: Shortest Paths

Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000syllabus

Business

Still working on midterm grading – should be done soon!

Homework 3 is out, due Monday at midnight Boston time

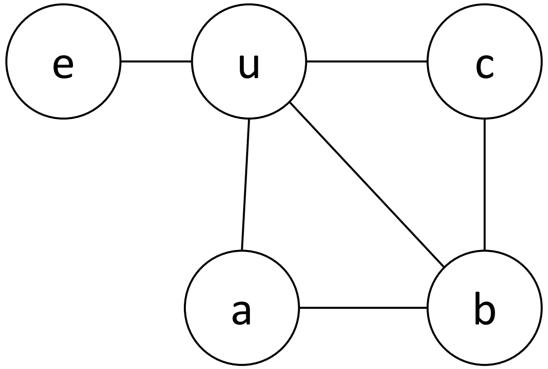
Last time: Betweenness Centrality

Betweenness centrality is used as a proxy for the importance of a node in facilitating connections between other nodes.

For node u, betweenness is measured as the ratio of shortest paths between all other pairs of nodes (s, t) that u lies on. Formally:

$$B(u) = \sum_{s \neq t \neq u} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

Where σ_{st} is the number of shortest paths between nodes s and t and $\sigma_{st}(u)$ is the number of those shortest paths that include u.



Last time: How do we compute shortest paths?

To compute betweenness centrality, we need to compute shortest paths for all pairs of nodes.

Rather than jumping straight there, let’s first solve a simpler problem: finding the length of the shortest path from a single node s to all other nodes in the graph (called the *single source shortest path* problem)

We can use BFS!

Last time: Single Source Shortest Paths with BFS

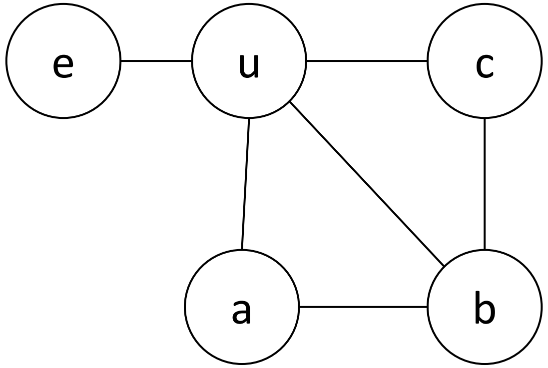
`dist`[v] stores the current estimate of the distance between our source node s and the node v, initialized to infinity

We walk along every edge of the graph and check whether the distance currently stored for v could be made shorter by routing through u

If yes, we update the distance, and store u as the *predecessor* (similar to parent) of v in the shortest path

Once BFS is done, we have the shortest path length from s to every node v stored in `dist[`v`]` and we can recover a shortest path for any node by following pred back to s

```
SSSP-BFS($s$):
  dist[$u$]← $\infty$ for all $u \in V$
  pred[$u$]← $null$ for all $u \in V$
  dist[$s$]← $0$
  Q ← $s$
  While Q is not empty:
    $u$ ← Pull(Q)
    For $v \in Neighbors(u)$:
      If dist[$v$] > dist[$u$] + 1:
        dist[$v$] = dist[$u$] + 1
        pred[$v$] = $u$
        Push($Q$,$v$)
```



Last time: Single Source Shortest Paths with BFS

`dist`$[v]$ stores the current estimate of the distance between our source node s and the node v, initialized to infinity

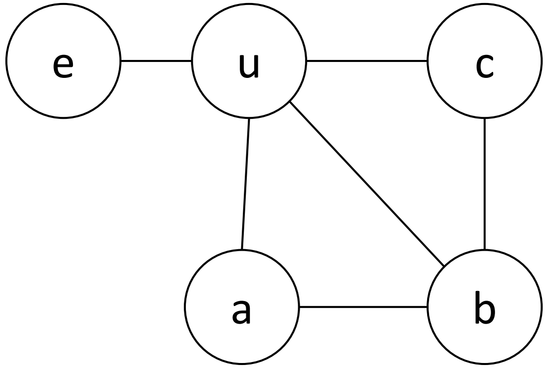
We walk along every edge of the graph and check whether the distance currently stored for v could be made shorter by routing through u

If yes, we update the distance, and store u as the *predecessor* (similar to parent) of v in the shortest path

Once BFS is done, we have the shortest path length from s to every node v stored in `dist`$[v]$ and **we can recover a shortest path for any node by following pred back to s**

```
SSSP-BFS($s$):
  dist[$u$]← $\infty$ for all $u \in V$
  pred[$u$]← $null$ for all $u \in V$
  dist[$s$]← $0$
  Q ← $s$
  While Q is not empty:
    $u$ ← Pull(Q)
    For $v \in Neighbors(u)$:
      If dist[$v$] > dist[$u$] + 1:
        dist[$v$] = dist[$u$] + 1
        pred[$v$] = $u$
        Push($Q,v$)
```

When we compute betweenness centrality, we will need all of the paths! How?



Recovering all paths from SSSP-BFS

A simple modification allows us to store all of the possible shortest paths.

We just need to adjust our pred[u] data structure to store a *list* of predecessors, rather than just one!

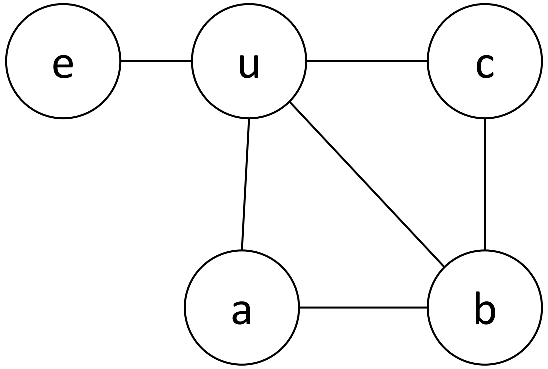
```
SSSP-BFS($s$):
  dist[$u$]$\leftarrow \infty$ for all $u \in V$
  pred[$u$]$\leftarrow null$ for all $u \in V$
  dist[$s$]$\leftarrow 0$
  Q $\leftarrow s$
  While Q is not empty:
    $u \leftarrow$ Pull(Q)
    For $v \in Neighbors(u)$:
      If dist[$v$] > dist[$u$] + 1:
        dist[$v$] = dist[$u$] + 1
        pred[$v$] = $u$
        Push($Q$,$v$)
```

Recovering all paths from SSSP-BFS

A simple modification allows us to store all of the possible shortest paths.

We just need to adjust our pred$[u]$ data structure to store a *list* of predecessors, rather than just one!

```
SSSP-BFS($s$):
  dist[$u$]$\leftarrow \infty$ for all $u \in V$
  pred[$u$]$\leftarrow null$ for all $u \in V$
  dist[$s$]$\leftarrow 0$
  Q $\leftarrow s$
  While Q is not empty:
    $u \leftarrow$ Pull(Q)
    For $v \in Neighbors(u)$:
      If dist[$v$] > dist[$u$] + 1:
        dist[$v$] = dist[$u$] + 1
        pred[$v$] = [$u$]
        Push($Q$,$v$)
      Else If dist[$v$] = dist[$u$] + 1:
        Append $u$ to pred[$v$]
        Push($Q$,$v$)
```



All Pairs Shortest Paths with BFS

We have an algorithm for computing shortest paths from a single source node to every other node

We need the shortest paths for *all pairs* of nodes (the *all pairs shortest paths* problem)

One option: Just run SSSP-BFS from every node!

```
APSP-BFS($G = (V, E)$):
  For $v \in V$:
    SSSP-BFS($v$)
```

Running time

For each of n nodes, we run a full BFS. BFS runs in $O(n + m)$ time. Therefore we have $O(n(n + m))$, or $O(n^2 + nm)$.

All Pairs Shortest Paths with BFS

We have an algorithm for computing shortest paths from a single source node to every other node

We need the shortest paths for *all pairs* of nodes (the *all pairs shortest paths* problem)

One option: Just run SSSP-BFS from every node!

```
APSP-BFS-Paths($G = (V, E)$):
  For $s \in V$:
    SSSP-BFS($s$) // fills pred[$u$] $\forall_u$
    For $t \in V$:
      If $s \neq t$ and $s > t$:
        paths[s,t] ← RecoverPaths($s$,$t$)
```

Running time

For each of n nodes, we run a full BFS. BFS runs in $O(n+m)$ time. Therefore we have $O(n(n+m))$, or $O(n^2+nm)$.

RecoverPaths is $O(n)$, meaning the doubly nested loop is $O(n^3)$, regardless of SSSP-BFS.

Betweenness Centrality

Now we can compute betweenness centrality:

$$B(u) = \sum_{s \neq t \neq u} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

Where σ_{st} is the number of shortest paths between nodes s and t and $\sigma_{st}(u)$ is the number of those shortest paths that include u.

```
Betweenness($G$):
  paths ← APSP-BFS-Paths( $G$ )
  For $u \in V$:
    For $s \in V$:
      For $t \in V$:
        if $s \neq t$:
          $denominator \leftarrow$ |paths[s,t]|
          $numerator \leftarrow$ |paths[s,t] that contain $u$|
          $b[u] \mathrel{+}= \frac{numerator}{denominator}$

  Return $b$
```

Betweenness Centrality

Now we can compute betweenness centrality:

$$B(u) = \sum_{s \neq t \neq u} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

Where σ_{st} is the number of shortest paths between nodes s and t and $\sigma_{st}(u)$ is the number of those shortest paths that include u.

```
Betweenness($G$):
    paths ← APSP-BFS-Paths($G$)
    For $u \in V$:
        For $s \in V$:
            For $t \in V$:
                if $s \neq t$:
                    $denominator \leftarrow$ |paths[s,t]|
                    $numerator \leftarrow$ |paths[s,t] that contain $u$|
                    $b[u] \mathrel{+}= \frac{numerator}{denominator}$

    Return $b$
```

Question: Does all of this work on directed graphs?

Betweenness Centrality

Now we can compute betweenness centrality:

$$B(u) = \sum_{s \neq t \neq u} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

Where σ_{st} is the number of shortest paths between nodes s and t and $\sigma_{st}(u)$ is the number of those shortest paths that include u.

```
Betweenness($G$):
    paths ← APSP-BFS-Paths( $G$ )
    For $u \in V$:
        For $s \in V$:
            For $t \in V$:
                if $s \neq t$:
                    $denominator \leftarrow |$paths[s,t]$|$
                    $numerator \leftarrow |$paths[s,t] that contain $u|$
                    $b[u] \mathrel{+}= \frac{numerator}{denominator}$

    Return $b$
```

Question: Does all of this work on directed graphs?

Yes! With some modification to APSP-BFS-Paths to account for when a path does not exist (dist[s,t]=∞)

What about weighted graphs?

So far, we have only considered unweighted graphs, or equivalently graphs with uniform weights.

We may want to find shortest paths in a *weighted graph* $G = (V, E, W)$ where W is a set of weights corresponding to the edges, e.g. $\mathrm{W} = (u, v, w)$ where w is a nonnegative integer for all $(\mathrm{u}, \mathrm{v}) \in E$.

Generalizing SSSP-BFS: Best First Search

We can modify our BFS based algorithm to take edge weight into account

The distance corresponding to a path between two nodes is now the sum of the edge weights along the path

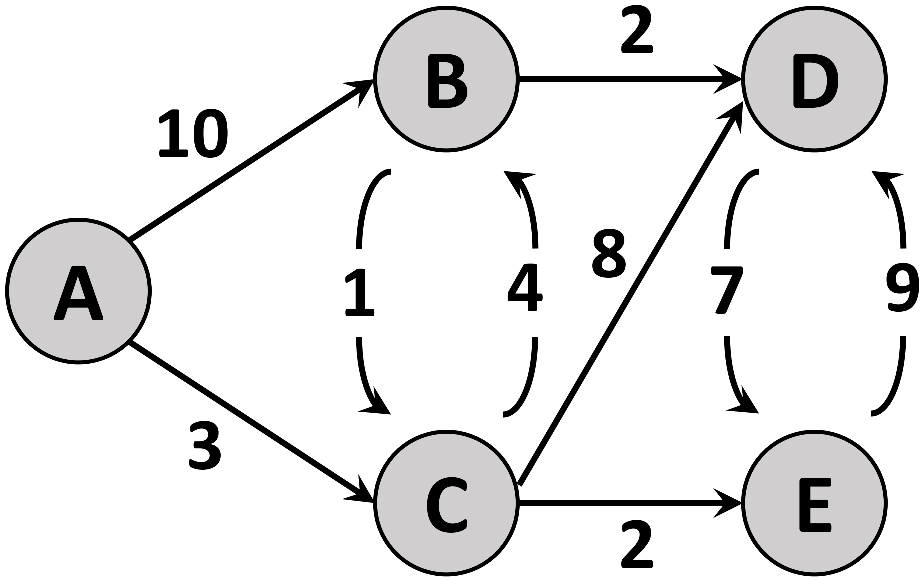
Modification requires taking a “global” view of the graph – the next step in any traversal algorithm involves choosing an edge to follow, we will choose it in a smarter way.

Dijkstra’s Algorithm: choose the minimum distance edge to try to update next using a priority queue

Dijkstra’s algorithm is an example of a “best first search” approach to graph traversal: we have some criteria (known as a heuristic) for choosing a good next node, so we use it.

```
SSSP-Dijkstra($s$):
    dist[$u$]$\leftarrow \infty$ for all $u \in V$
    pred[$u$]$\leftarrow null$ for all $u \in V$
    dist[$s$]$\leftarrow 0$
    Q $\leftarrow (s, 0)$
    While Q is not empty:
        $u \leftarrow$ PullMinimum(Q)
        For $v \in Neighbors(u)$:
            If dist[$v$] > dist[$u$] + 1:
                dist[$v$] = dist[$u$] + 1
                pred[$v$] = [$u$]
                PushOrReplace($Q$,$v$,dist[$v$])
            Else If dist[$v$] = dist[$u$] + 1:
                Append $u$ to pred[$v$]
                PushOrReplace($Q$,$v$,dist[$v$])
```

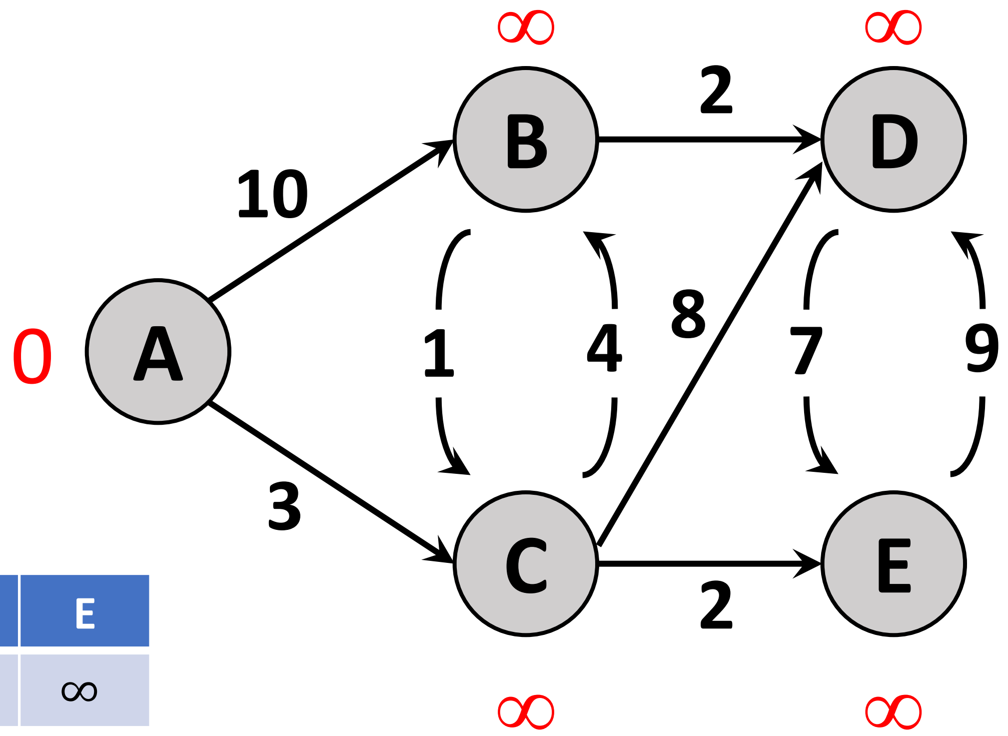
Dijkstra’s Algorithm: Demo



Example from Jon Ullman

Dijkstra’s Algorithm: Demo

Initialize

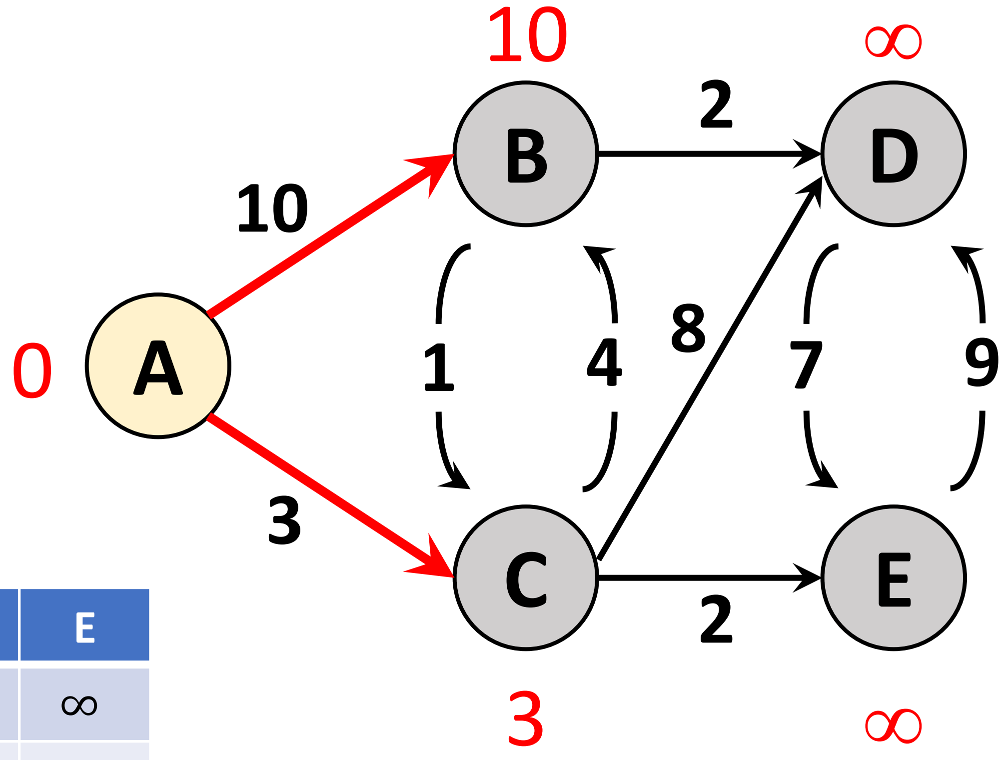


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞

$S = \{\}$

Dijkstra’s Algorithm: Demo

Explore A

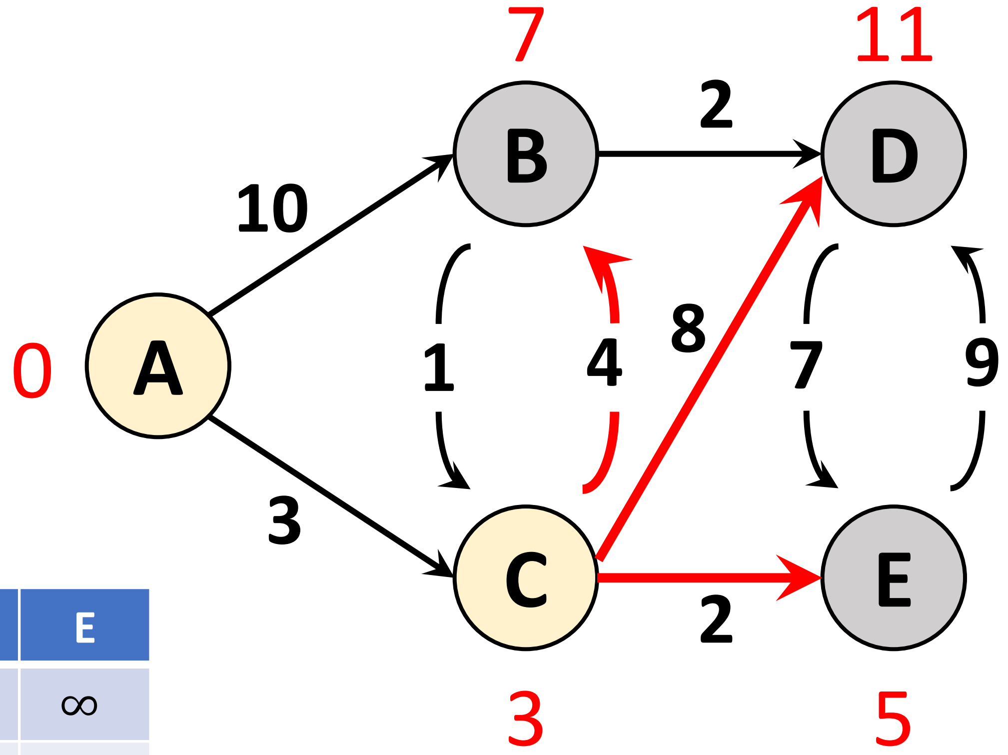


	A	B	C	D	E
$d_0(u)$	**0**	∞	∞	∞	∞
$d_1(u)$	**0**	**10**	**3**	∞	∞

$$S = \{A\}$$

Dijkstra’s Algorithm: Demo

Explore C

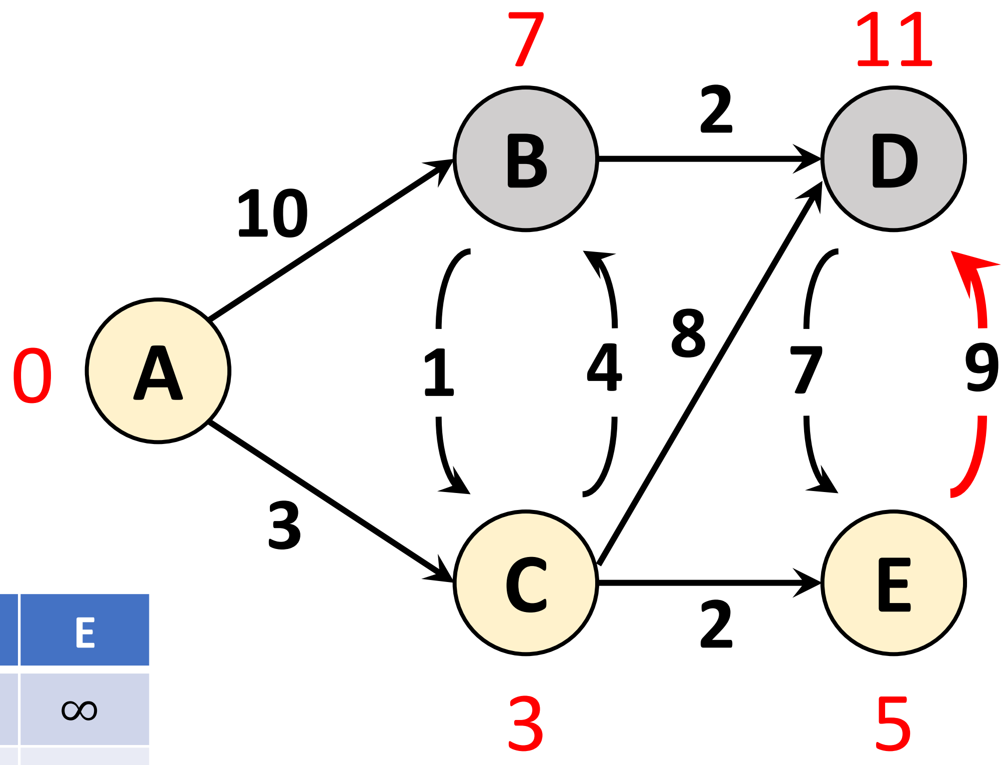


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5

$$S = \{A, C\}$$

Dijkstra’s Algorithm: Demo

Explore E

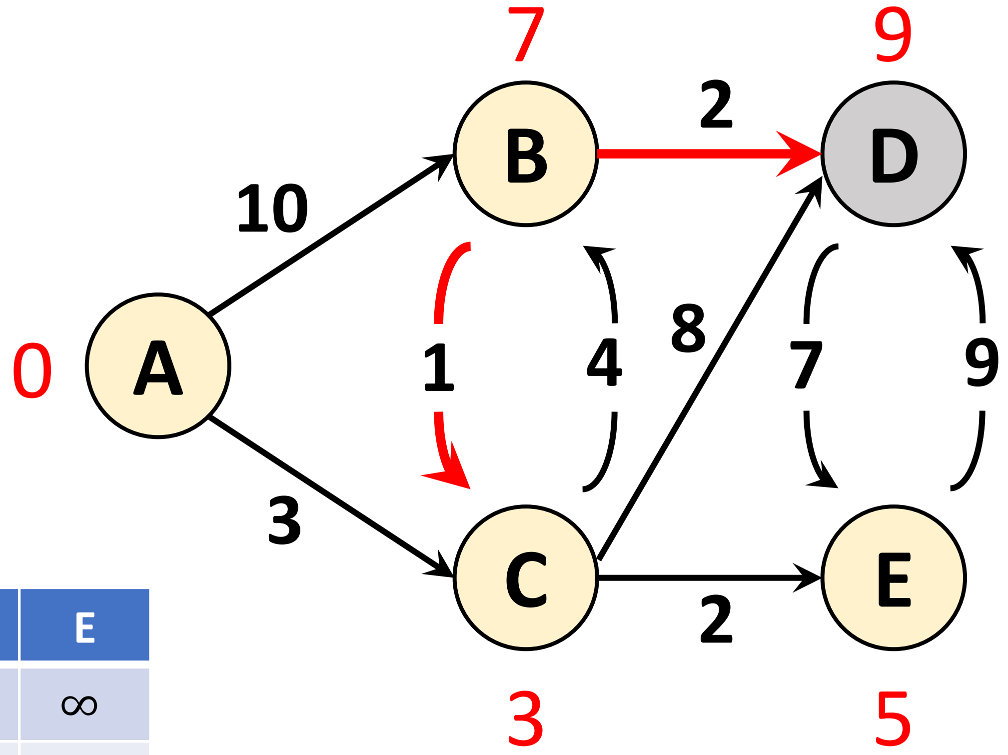


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5

$$S = \{A, C, E\}$$

Dijkstra’s Algorithm: Demo

Explore B

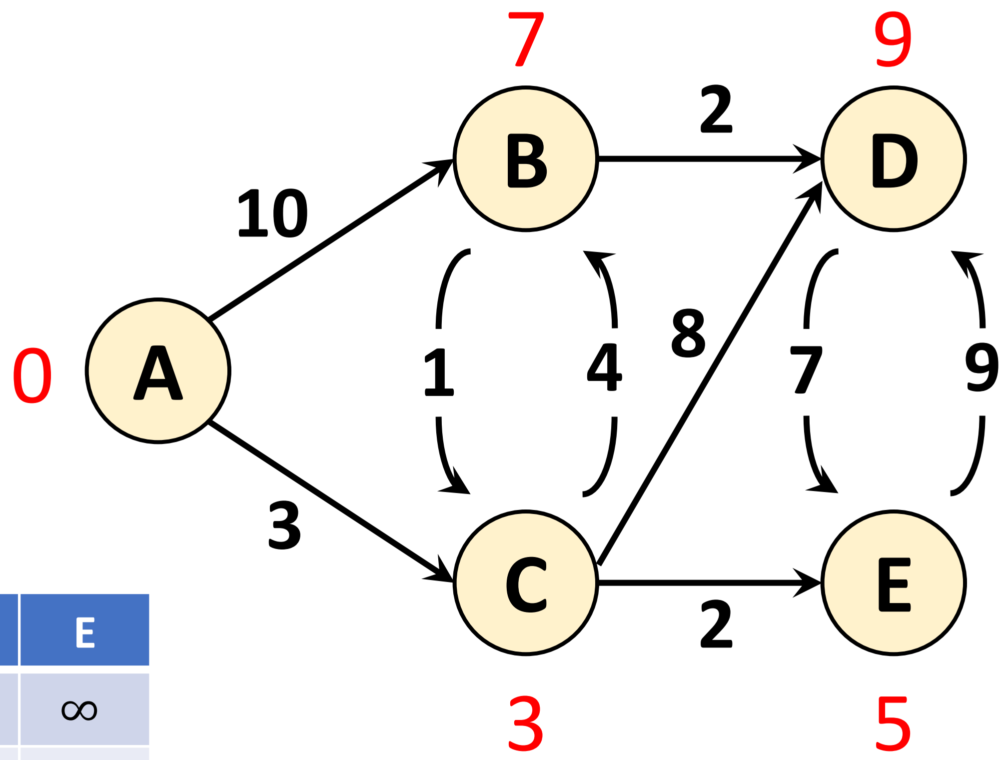


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

$$S = \{A, C, E, B\}$$

Dijkstra’s Algorithm: Demo

Don’t need to explore D

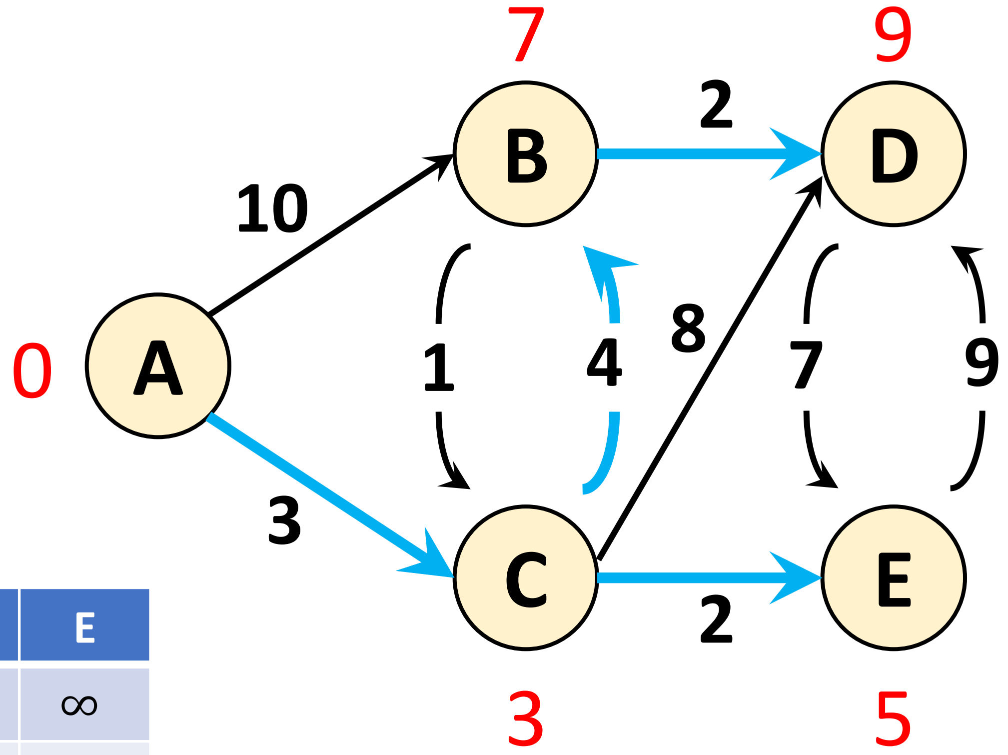


	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

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

Dijkstra's Algorithm: Demo

Maintain parent pointers so we can find the shortest paths



	A	B	C	D	E
$d_0(u)$	0	∞	∞	∞	∞
$d_1(u)$	0	10	3	∞	∞
$d_2(u)$	0	7	3	11	5
$d_3(u)$	0	7	3	11	5
$d_4(u)$	0	7	3	9	5

Dijkstra: Why does it work?

At the beginning, we have only that the distance from s to itself is 0, which is true by assumption.

```
SSP-Dijkstra($s$):
  dist[$u$]$\leftarrow \infty$ for all $u \in V$
  pred[$u$]$\leftarrow$ $null$ for all $u \in V$
  dist[$s$]$\leftarrow 0$
  Q $\leftarrow$ $(s, 0)$
  While Q is not empty:
    $u \leftarrow$ PullMinimum(Q)
    For $v \in Neighbors(u)$:
      If dist[$v$] > dist[$u$] + 1:
        dist[$v$] = dist[$u$] + 1
        pred[$v$] = [$u$]
        PushOrReplace($Q$,$v$,dist[$v$])
      Else If dist[$v$] = dist[$u$] + 1:
        Append $u$ to pred[$v$]
        PushOrReplace($Q$,$v$,dist[$v$])
```

Dijkstra: Why does it work?

At the beginning, we have only that the distance from s to itself is 0, which is true by assumption.

First, we explore the neighborhood of s and set all distances to its neighbors correctly.

```
SSSP-Dijkstra($s$):
   dist[$u$]$\leftarrow \infty$ for all $u \in V$
   pred[$u$]$\leftarrow null$ for all $u \in V$
   dist[$s$]$\leftarrow 0$
   Q $\leftarrow (s, 0)$
   While Q is not empty:
      $u \leftarrow$ PullMinimum(Q)
      For $v \in Neighbors(u)$:
         If dist[$v$] > dist[$u$] + 1:
            dist[$v$] = dist[$u$] + 1
            pred[$v$] = [$u$]
            PushOrReplace($Q$,$v$,dist[$v$])
         Else If dist[$v$] = dist[$u$] + 1:
            Append $u$ to pred[$v$]
            PushOrReplace($Q$,$v$,dist[$v$])
```

Dijkstra: Why does it work?

At the beginning, we have only that the distance from s to itself is 0, which is true by assumption.

First, we explore the neighborhood of s and set all distances to its neighbors correctly.

Then we choose another node, call it v_1, and correctly set all of the distances from $s \rightarrow v_1 \rightarrow t$, where $t \in Neighbors(v_1)$.

```
SSSP-Dijkstra($s$):
    dist[$u$]$\leftarrow \infty$ for all $u \in V$
    pred[$u$]$\leftarrow null$ for all $u \in V$
    dist[$s$]$\leftarrow 0$
    Q $\leftarrow (s, 0)$
    While Q is not empty:
        $u \leftarrow$ PullMinimum(Q)
        For $v \in Neighbors(u)$:
            If dist[$v$] > dist[$u$] + 1:
                dist[$v$] = dist[$u$] + 1
                pred[$v$] = [$u$]
                PushOrReplace($Q$,$v$,dist[$v$])
            Else If dist[$v$] = dist[$u$] + 1:
                Append $u$ to pred[$v$]
                PushOrReplace($Q$,$v$,dist[$v$])
```

Dijkstra: Why does it work?

At the beginning, we have only that the distance from s to itself is 0, which is true by assumption.

First, we explore the neighborhood of s and set all distances to its neighbors correctly.

Then we choose another node, call it v_1, and correctly set all of the distances from $s \to v_1 \to t$, where $t \in Neighbors(v_1)$.

At the i^{th} node, we set the correct distances from $s \rightsquigarrow v_i \rightsquigarrow t$, where $t \in Neighbors(v_i)$.

```
SSP-Dijkstra($s$):
    dist[$u$]$\leftarrow \infty$ for all $u \in V$
    pred[$u$]$\leftarrow null$ for all $u \in V$
    dist[$s$]$\leftarrow 0$
    Q $\leftarrow (s, 0)$
    While Q is not empty:
        $u \leftarrow$ PullMinimum(Q)
        For $v \in Neighbors(u)$:
            If dist[$v$] > dist[$u$] + 1:
                dist[$v$] = dist[$u$] + 1
                pred[$v$] = [$u$]
                PushOrReplace($Q$,$v$,dist[$v$])
            Else If dist[$v$] = dist[$u$] + 1:
                Append $u$ to pred[$v$]
                PushOrReplace($Q$,$v$,dist[$v$])
```

Dijkstra: Why does it work?

At the beginning, we have only that the distance from s to itself is 0, which is true by assumption.

First, we explore the neighborhood of s and set all distances to its neighbors correctly.

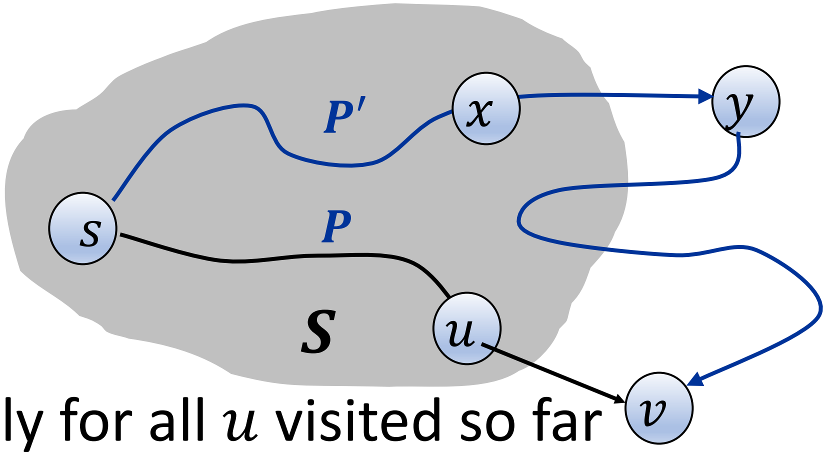
Then we choose another node, call it v_1, and correctly set all of the distances from $s \rightarrow v_1 \rightarrow t$, where $t \in Neighbors(v_1)$.

At the i^{th}node, we set the correct distances from $s \rightsquigarrow v_i \rightsquigarrow t$, where $t \in Neighbors(v_i)$.

Invariant: After we explore the i^{th} node, dist[u] is set correctly for all u visited so far

```
SSSP-Dijkstra($s$):
    dist[$u$]$\leftarrow \infty$ for all $u \in V$
    pred[$u$]$\leftarrow null$ for all $u \in V$
    dist[$s$]$\leftarrow 0$
    Q $\leftarrow (s, 0)$
    While Q is not empty:
        $u \leftarrow$ PullMinimum(Q)
        For $v \in Neighbors(u)$:
            If dist[$v$] > dist[$u$] + 1:
                dist[$v$] = dist[$u$] + 1
                pred[$v$] = [$u$]
                PushOrReplace($Q$,$v$,dist[$v$])
            Else If dist[$v$] = dist[$u$] + 1:
                Append $u$ to pred[$v$]
                PushOrReplace($Q$,$v$,dist[$v$])
```

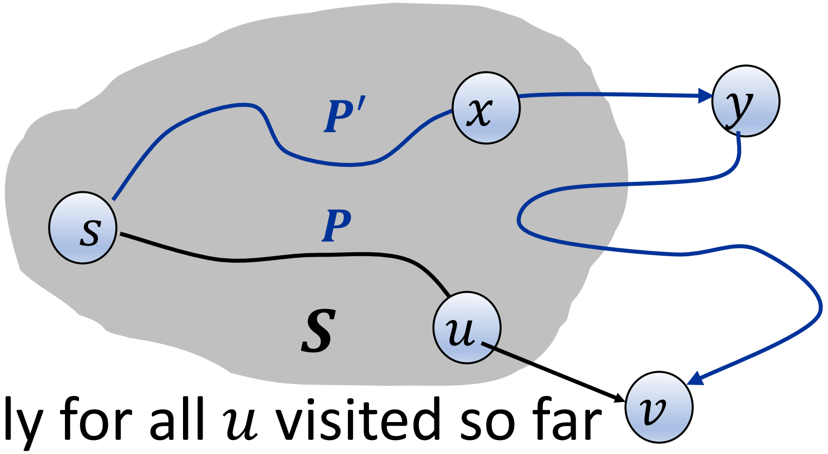
Dijkstra: Why does it work?



Invariant: After we explore the i^{th} node, dist$[u]$ is set correctly for all u visited so far

We want to prove that $d_i(v) = d_i(u) + w_{uv}$ is the shortest path from s to v if v is the next node in the priority queue. We showed this works for $i = 1$ and $i = 2$.

Dijkstra: Why does it work?

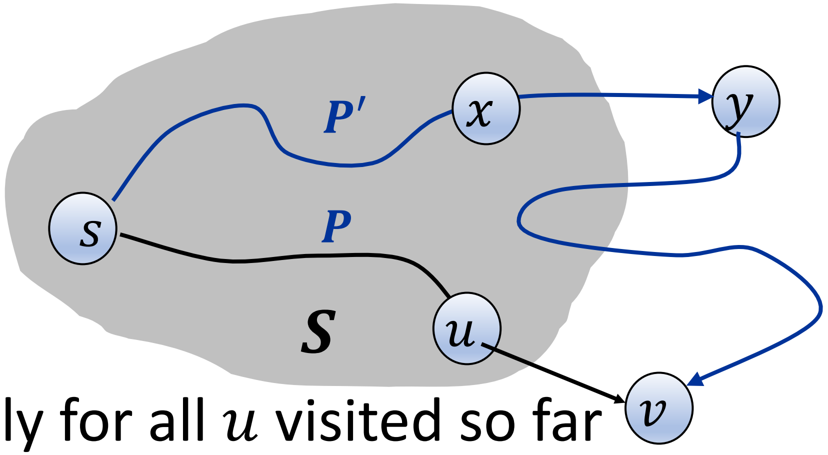


Invariant: After we explore the i^{th} node, dist[u] is set correctly for all u visited so far

We want to prove that $d_i(v) = d_i(u) + w_{uv}$ is the shortest path from s to v if v is the next node in the priority queue. We showed this works for $i = 1$ and $i = 2$.

Consider the picture above, which represents two possibilities for paths from s to some node v. The path P represents an actual shortest path, while $P`$ represents an alternative path assuming some node $y$ was actually a better next choice than $v$, meaning that $\ell\ (P`) < \ell(P)$. We have:

Dijkstra: Why does it work?



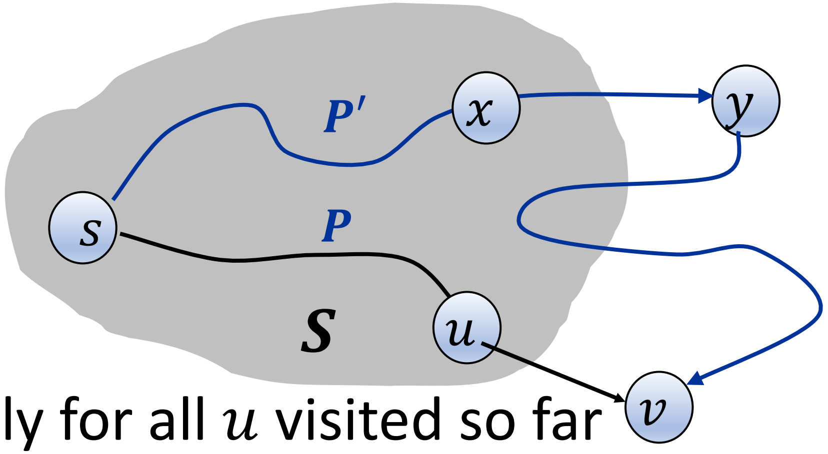
Invariant: After we explore the i^{th} node, dist[u] is set correctly for all u visited so far

We want to prove that $d_i(v) = d_i(u) + w_{uv}$ is the shortest path from s to v if v is the next node in the priority queue. We showed this works for $i = 1$ and $i = 2$.

Consider the picture above, which represents two possibilities for paths from s to some node v. The path P represents an actual shortest path, while $P`$ represents an alternative path assuming some node $y$ was actually a better next choice than $v$, meaning that $\ell\,(P`) < \ell(P)$. We have:

$$\ell(P`) = d_i(\mathrm{x}) + \mathrm{w}_{\mathrm{xy}} + \mathrm{w}_{\mathrm{yv}}$$

Dijkstra: Why does it work?



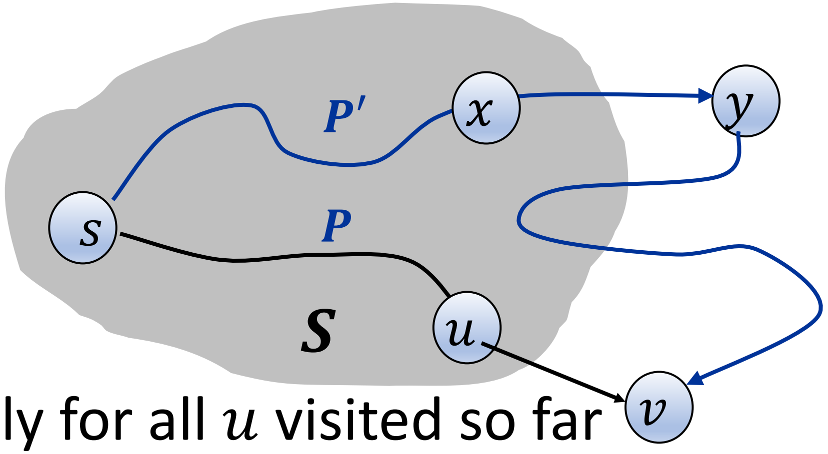
Invariant: After we explore the i^{th} node, dist[u] is set correctly for all u visited so far

We want to prove that $d_i(v) = d_i(u) + w_{uv}$ is the shortest path from s to v if v is the next node in the priority queue. We showed this works for $i = 1$ and $i = 2$.

Consider the picture above, which represents two possibilities for paths from s to some node v. The path P represents an actual shortest path, while P\` represents an alternative path assuming some node y was actually a better next choice than v, meaning that $\ell\,(P\grave{}) < \ell(P)$. We have:

$$\begin{aligned} \ell(P\grave{}) &= d_i(\mathrm{x}) + \mathrm{w_{xy}} + \mathrm{w_{yv}} \\ &\geq d_i(x) + w_{xy} \qquad \text{Since } w_{yv} \geq 0 \end{aligned}$$

Dijkstra: Why does it work?



Invariant: After we explore the i^{th} node, dist[u] is set correctly for all u visited so far

We want to prove that $d_i(v) = d_i(u) + w_{uv}$ is the shortest path from s to v if v is the next node in the priority queue. We showed this works for $i = 1$ and $i = 2$.

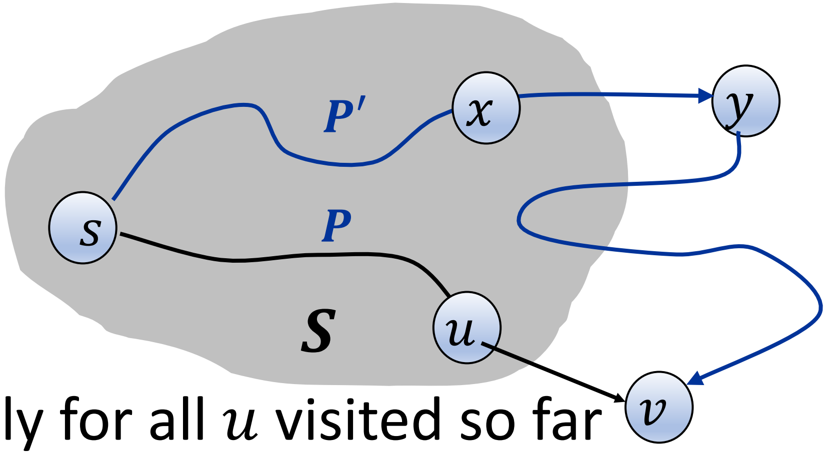
Consider the picture above, which represents two possibilities for paths from s to some node v. The path P represents an actual shortest path, while $P`$ represents an alternative path assuming some node $y$ was actually a better next choice than $v$, meaning that $\ell\ (P`) < \ell(P)$. We have:

$$\ell(P`) = d_i(\mathrm{x}) + \mathrm{w}_{\mathrm{xy}} + \mathrm{w}_{\mathrm{yv}}$$

$$\geq d_i(x) + w_{xy}$$ Since $w_{yv} \geq 0$

$$\geq d_i(y)$$ We know x is explored already

Dijkstra: Why does it work?



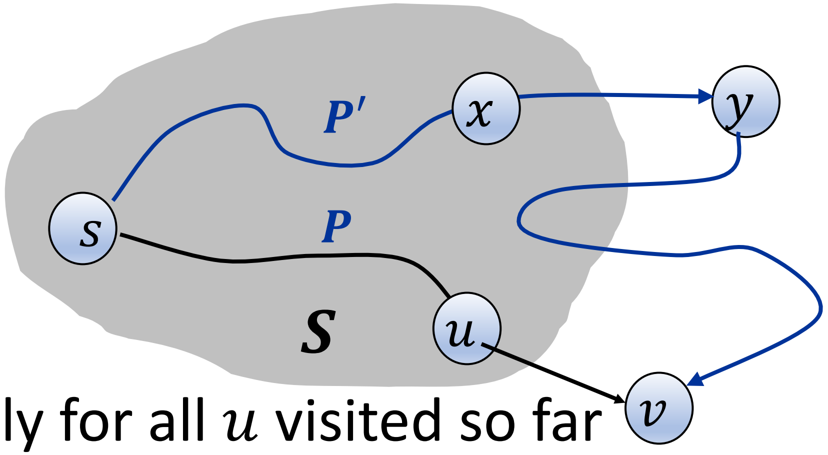
Invariant: After we explore the i^{th} node, dist[u] is set correctly for all u visited so far

We want to prove that $d_i(v) = d_i(u) + w_{uv}$ is the shortest path from s to v if v is the next node in the priority queue. We showed this works for $i = 1$ and $i = 2$.

Consider the picture above, which represents two possibilities for paths from s to some node v. The path P represents an actual shortest path, while $P`$ represents an alternative path assuming some node $y$ was actually a better next choice than $v$, meaning that $\ell\,(P`) < \ell(P)$. We have:

$$\begin{aligned}\ell(P`) &= d_i(\mathrm{x}) + \mathrm{w_{xy}} + \mathrm{w_{yv}} \\ &\geq d_i(x) + w_{xy} && \text{Since } w_{yv} \geq 0 \\ &\geq d_i(y) && \text{We know } x \text{ is explored already} \\ &\geq d_i(v) && \text{We chose } v \text{ to explore, not } y!\end{aligned}$$

Dijkstra: Why does it work?



Invariant: After we explore the i^{th} node, dist[u] is set correctly for all u visited so far

We want to prove that $d_i(v) = d_i(u) + w_{uv}$ is the shortest path from s to v if v is the next node in the priority queue. We showed this works for $i = 1$ and $i = 2$.

Consider the picture above, which represents two possibilities for paths from s to some node v. The path P represents an actual shortest path, while $P`$ represents an alternative path assuming some node $y$ was actually a better next choice than $v$, meaning that $\ell\ (P`) < \ell(P)$. We have:

$$\begin{aligned} \ell(P`) &= d_i(\mathrm{x}) + \mathrm{w_{xy}} + \mathrm{w_{yv}} \\ &\geq d_i(x) + w_{xy} && \text{Since } w_{yv} \geq 0 \\ &\geq d_i(y) && \text{We know } x \text{ is explored already} \\ &\geq d_i(v) && \text{We chose } v \text{ to explore, not } y\text{!} \\ &= \ell(P) && \text{So } \ell\ (P`) \geq \ell(P)\text{. Contradiction!} \end{aligned}$$

Dijkstra running time

Assuming our priority queue supports insertion, update, and extraction in $O(log\ E)$ time, this approach runs in $O(n + \log E)$

```
SSSP-Dijkstra($s$):
   dist[$u$]$\leftarrow \infty$ for all $u \in V$
   pred[$u$]$\leftarrow null$ for all $u \in V$
   dist[$s$]$\leftarrow 0$
   Q $\leftarrow (s, 0)$
   While Q is not empty:
      $u \leftarrow$ PullMinimum(Q)
      For $v \in Neighbors(u)$:
         If dist[$v$] > dist[$u$] + 1:
            dist[$v$] = dist[$u$] + 1
            pred[$v$] = [$u$]
            PushOrReplace($Q$,$v$,dist[$v$])
         Else If dist[$v$] = dist[$u$] + 1:
            Append $u$ to pred[$v$]
            PushOrReplace($Q$,$v$,dist[$v$])
```

Floyd-Warshall

What about applications where negative edgeweights make sense?

- Transactions
- Chemical reactions
- Changes over time

The *Floyd-Warshall* algorithm is a dynamic programming solution to solving the all-pairs-shortest-paths problem on weighted, directed graphs that have no *negative cycles*.

Floyd-Warshall

What about applications where negative edgeweights make sense?

- Transactions
- Chemical reactions
- Changes over time

The *Floyd-Warshall* algorithm is a dynamic programming solution to solving the all-pairs-shortest-paths problem on weighted, directed graphs that have no *negative cycles*.

(sub)homework 4: Read/watch about Floyd-Warshall and translate recursive definition and pseudocode into LaTeX!

- Will be concurrent with Homework 3 but due Tuesday at Midnight
- Released shortly after class
- Very easy LaTeX practice! Just translate something you are given into LaTex.

Next Time

Spanning trees and flow algorithms

Suggested Reading: Erickson Chapter 7 and Chapter 10 through 10.3

Keep working on homeworks, ask questions on Piazza, and have a great weekend!