CS 170: Algorithms
CS 170: Algorithms
CS 170: Algorithms
CS 170: Algorithms
CS 170: Algorithms
Dijkstra's Direct Inductive Proof.

Know distance to $S$.

Inv: $d(v) - \text{length of path through } S$.

Smallest $d(v)$ is correct, add to $S$.

Updating neighbors of $v$ enforces Inv.

Bellman-Ford: Negative weights.

Dijkstra doesn't work.

Update edge $(u, v)$: $d(v) = \min(d(v), d(u) + l(u, v))$.

Update all edges $|V| - 1$ times.

Paths of length $k$ correct after iteration $k$.

DAG: linearize and process vertices in order.

Updates of edges in order along path.
Dijkstra’s Direct Inductive Proof.
Know distance to $S$.
  Inv: $d(v)$ - length of path through $S$.
  Smallest $d(v)$ is correct, add to $S$
  Updating neighbors of $v$ enforces Inv.
Dijkstra’s Direct Inductive Proof.
Know distance to \( S \).
Inv: \( d(v) \) - length of path through \( S \).
Smallest \( d(v) \) is correct, add to \( S \)
Updating neighbors of \( v \) enforces Inv.

Bellman-Ford: Negative weights.
Dijkstra doesn’t work.
Update edge \((u, v)\): \( d(v) = \min(d(v), d(u) + l(u, v)) \).
Update all edges \(|V| – 1 \) times.
Paths of length \( k \) correct after iteration \( k \).
Dijkstra’s Direct Inductive Proof.
Know distance to $S$.
Inv: $d(v)$ - length of path through $S$.
Smallest $d(v)$ is correct, add to $S$
Updating neighbors of $v$ enforces Inv.

Bellman-Ford: Negative weights.
Dijkstra doesn’t work.
Update edge $(u, v)$: $d(v) = \min(d(v), d(u) + l(u, v))$.
Update all edges $|V| - 1$ times.
Paths of length $k$ correct after iteration $k$.

DAG:
linearize and process vertices in order.
Updates of edges in order along path.
Dijkstra’s Algorithm.

\[ \text{foreach } v : d(v) = \infty. \]
Dijkstra’s Algorithm.

\textbf{foreach} \textit{v}: \(d(v) = \infty\).

\(d(s) = 0\).
Dijkstra's Algorithm.

foreach \( v \): \( d(v) = \infty \).
\( d(s) = 0 \).
Q.Insert(s,0)
Dijkstra’s Algorithm.

foreach $v$: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)

While $u = Q.DeleteMin()$:
  foreach edge $(u, v)$:
Dijkstra’s Algorithm.

\[
\text{foreach } v: d(v) = \infty.
\]
\[
d(s) = 0.
\]
\[
\text{Q.Insert(s,0)}
\]
\[
\textbf{While } u = \text{Q.DeleteMin}():
\]
\[
\quad \text{foreach edge } (u, v):
\]
\[
\quad \quad \textbf{if } d(v) > d(u) + l(u, v):
\]
\[
\quad \quad \quad d(v) = d(u) + l(u, v)
\]
\[
\quad \quad \text{Q.InsertOrDecreaseKey(v,d(v))}
\]
Dijkstra’s Algorithm.

\[
\text{foreach } v: \quad d(v) = \infty.
\]
\[
d(s) = 0.
\]
\[
\text{Q.Insert(s,0)}
\]
\[
\text{While } \ u = \text{Q.DeleteMin():}
\]
\[
\text{foreach edge } (u, v):
\]
\[
\text{if } \quad d(v) > d(u) + l(u, v):
\]
\[
\quad d(v) = d(u) + l(u, v)
\]
\[
\quad \text{Q.InsertOrDecreaseKey}(v, d(v))
\]

Runtime:
Dijkstra’s Algorithm.

foreach v: \( d(v) = \infty \).
d(s) = 0.
Q.Insert(s,0)

While u = Q.DeleteMin():
    foreach edge (u, v):
        if \( d(v) > d(u) + l(u, v) \):
            \( d(v) = d(u) + l(u, v) \)
            Q.InsertOrDecreaseKey(v,d(v))

Runtime:
| \(|V| \) DeleteMins. |
Dijkstra’s Algorithm.

foreach \( v \): \( d(v) = \infty \).
\( d(s) = 0 \).
Q.Insert(s,0)

While \( u = Q.DeleteMin() \):
  foreach edge \( (u, v) \):
    if \( d(v) > d(u) + l(u, v) \):
      \( d(v) = d(u) + l(u, v) \)
      Q.InsertOrDecreaseKey(v,d(v))

Runtime:
\(|V|\) DeleteMins.
\(|V|\) Inserts.
Dijkstra’s Algorithm.

\[
\text{foreach } v: \; d(v) = \infty.
\]
\[
d(s) = 0.
\]
\[
\text{Q.Insert}(s,0)
\]
\[
\text{While } u = \text{Q.DeleteMin}():
\]
\[
\text{foreach } \text{edge } (u, v):
\]
\[
\quad \text{if } d(v) > d(u) + l(u,v):
\]
\[
\quad \quad d(v) = d(u) + l(u,v)
\]
\[
\quad \quad \text{Q.InsertOrDecreaseKey}(v,d(v))
\]

Runtime:
\[
|V| \text{ DeleteMins.}
\]
\[
|V| \text{ Inserts.}
\]
\[
\leq |E| \text{ DecreaseKeys.}
\]
Dijkstra’s Algorithm.

```plaintext
foreach v: \(d(v) = \infty\).
d(s) = 0.
Q.Insert(s,0)
While u = Q.DeleteMin():
    foreach edge \((u, v)\):
        if \(d(v) > d(u) + l(u, v)\):
            \(d(v) = d(u) + l(u, v)\)
            Q.InsertOrDecreaseKey(v,d(v))
```

**Runtime:**

|\(|V|\) | DeleteMins. |
|\(|V|\) | Inserts. |
\(\leq |E|\) DecreaseKeys.
Dijkstra's Algorithm.

\[
\textbf{foreach } v: \quad d(v) = \infty.
\]

\[
d(s) = 0.
\]

\[
\text{Q.Insert}(s, 0)
\]

\textbf{While } u = \text{Q.DeleteMin}():

\[
\textbf{foreach} \text{ edge } (u, v):
\]

\[
\textbf{if} \quad d(v) > d(u) + l(u, v):
\]

\[
d(v) = d(u) + l(u, v)
\]

\[
\text{Q.InsertOrDecreaseKey}(v, d(v))
\]

\textbf{Runtime:}

\[
\text{|V| DeleteMins.}
\]

\[
\text{|V| Inserts.}
\]

\[
\leq |E| \text{ DecreaseKeys.}
\]

Binary heap: \( O((|V| + |E|) \log |V|) \)
Alt Proof.

Dijkstra:

Know distance to processed nodes, \( R \).

Add node closest to \( s \) outside of \( R \).

Closest node \( v \) has path...

\[ s \ldots \]

\[ u \]

\[ v \]

in \( R \).

Since \( v \) is closest node not in \( R \).

\[ \text{d}(u) \] correct by induction.

\[ \text{d}(u) \] corresponds to the length of a shortest path.

\[ \text{d}(v) \leq \text{d}(u) + l(u, v) \]

Since \( u \) was processed by Algorithm.

\[ \text{d}(v) \] corresponds to length of path.

Set by some \( u' \), which corresponds to path by induction plus an edge \( (u', v') \).

Thus, when \( v \) is added to \( \text{d}(v) \) is correct.

Corresponds to the length of the shortest path.
Alt Proof.

Dijkstra:
"Know distance to processed nodes, $R$.”
Alt Proof.

Dijkstra:
"Know distance to processed nodes, \( R \)."
"Add node \( v \) closest to \( s \) outside of \( R \)."

Closest node \( v \) has path...
\( s \)
\( \cdots \)
\( u \)
\( \cdots \)
\( v \) in \( R \).

Since \( v \) is closest node not in \( R \).

\[ d(u) \]
correct by induction.

\[ d(v) \leq d(u) + l(u, v) \]
Since \( u \) was processed by Algorithm.

\[ d(v) \]
corresponds to length of path.

Set by some \( u \), which corresponds to path by induction plus an edge \( (u', v') \).

Thus, when \( v \) is added to \( d(v) \) is correct.

Corresponds to the length of the shortest path.
Alt Proof.

Dijkstra:
"Know distance to processed nodes, $R$.”
"Add node $v$ closest to $s$ outside of $R$."

Closest node $v$ has path...

![Diagram](image)

$d(u)$ correct by induction.
$d(v) \leq d(u) + l(u, v)$.

Since $u$ was processed by Algorithm.
$d(v)$ corresponds to length of path.
Set by some $u', v'$, which corresponds to path by induction plus an edge $(u', v')$.

Thus, when $v$ is added to $d(v)$ is correct.
Corresponds to the length of the shortest path.
Alt Proof.

Dijkstra:
"Know distance to processed nodes, $R$.”
"Add node $v$ closest to $s$ outside of $R$.”

Closest node $v$ has path...

$u$ in $R$. 
Alt Proof.

Dijkstra:
"Know distance to processed nodes, $R$.”
"Add node $v$ closest to $s$ outside of $R$.”

Closest node $v$ has path...

$u$ in $R$. Since $v$ is closest node not in $R$. 

\begin{tikzpicture}
  \node[shape=circle,draw=blue] (1) at (0,0) {s};
  \node[shape=circle,draw=blue] (2) at (1,0) {u};
  \node[shape=circle,draw=blue] (3) at (2,0) {v};
  \draw (1) -- (2);
  \draw (2) -- (3);
\end{tikzpicture}
Dijkstra:
"Know distance to processed nodes, $R$."
"Add node $v$ closest to $s$ outside of $R$."

Closest node $v$ has path...

$u$ in $R$. Since $v$ is closest node not in $R$. $d(u)$ correct by induction.
Alt Proof.

Dijkstra:
"Know distance to processed nodes, $R$.”
"Add node $v$ closest to $s$ outside of $R$.”

Closest node $v$ has path...

```
 s ··· u ··· v
```

$u$ in $R$. Since $v$ is closest node not in $R$.
$d(u)$ correct by induction.
$d(u)$ corresponds to the length of a shortest path.
$d(v) \leq d(u) + l(u, v)$.
Alt Proof.

Dijkstra:
"Know distance to processed nodes, \( R \)."
"Add node \( v \) closest to \( s \) outside of \( R \)."

Closest node \( v \) has path...
\[
\text{s} \quad \cdots \quad \text{u} \quad \text{v}
\]

\( u \) in \( R \). Since \( v \) is closest node not in \( R \).
\( d(u) \) correct by induction.
\( d(u) \) corresponds to the length of a shortest path.
\( d(v) \leq d(u) + l(u, v) \). Since \( u \) was processed by Algorithm.
Alt Proof.

Dijkstra:
"Know distance to processed nodes, $R$.”
"Add node $v$ closest to $s$ outside of $R$.”

Closest node $v$ has path...

$u$ in $R$. Since $v$ is closest node not in $R$.
$d(u)$ correct by induction.
$d(u)$ corresponds to the length of a shortest path.
$d(v) \leq d(u) + l(u, v)$. Since $u$ was processed by Algorithm.
$d(v)$ corresponds to length of path.
Alt Proof.

Dijkstra:
"Know distance to processed nodes, $R$.”
"Add node $v$ closest to $s$ outside of $R$.”

Closest node $v$ has path...

\[ s \rightarrow \cdots \rightarrow u \rightarrow v \]

$u$ in $R$. Since $v$ is closest node not in $R$.
$d(u)$ correct by induction.

$d(u)$ corresponds to the length of a shortest path.
$d(v) \leq d(u) + l(u, v)$. Since $u$ was processed by Algorithm.
$d(v)$ corresponds to length of path.

Set by some $u$, which corresponds to path by induction plus an edge $(u', v')$. 
Alt Proof.

Dijkstra:

"Know distance to processed nodes, \( R \)."
"Add node \( v \) closest to \( s \) outside of \( R \)."

Closest node \( v \) has path...
\[
\text{s} \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \text{u} \quad \text{v}
\]

\( u \) in \( R \). Since \( v \) is closest node not in \( R \).
\( d(u) \) correct by induction.

\( d(u) \) corresponds to the length of a shortest path.
\( d(v) \leq d(u) + l(u, v) \). Since \( u \) was processed by Algorithm.
\( d(v) \) corresponds to length of path.

Set by some \( u \), which corresponds to path by induction plus an edge \((u', v')\).

Thus, when \( v \) is added to \( d(v) \) is correct.
Alt Proof.

Dijkstra:
"Know distance to processed nodes, $R$."
"Add node $v$ closest to $s$ outside of $R$."

Closest node $v$ has path...

\[
\begin{array}{c}
  s \\
  \cdots \cdots \\
  u \\
  \cdots \cdots
  v \\
\end{array}
\]

$u$ in $R$. Since $v$ is closest node not in $R$.
$d(u)$ correct by induction.

$d(u)$ corresponds to the length of a shortest path.
$d(v) \leq d(u) + l(u, v)$. Since $u$ was processed by Algorithm.
$d(v)$ corresponds to length of path.

Set by some $u$, which corresponds to path by induction plus an edge $(u', v')$.

Thus, when $v$ is added to $d(v)$ is correct.
Corresponds to the length of the shortest path.
Negative edges.

Notice: argument for Dijkstra breaks for negative edges.
Negative edges.

Notice: argument for Dijkstra breaks for negative edges. For example.
Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

Dijkstra:
Negative edges.

Notice: argument for Dijkstra breaks for negative edges.
For example.

Dijkstra:
Process s:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

Dijkstra:
Process $s$: Set $d(a) = 1, d(b) = 2, d(c) = 3$. 
Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

Dijkstra:
Process $s$: Set $d(a) = 1$, $d(b) = 2$, $d(c) = 3$.
Process $a$, $d(a) = 1$: 
Notice: argument for Dijkstra breaks for negative edges. For example.

Dijkstra:
Process \(s\): Set \(d(a) = 1, d(b) = 2, d(c) = 3\).
Process \(a\), \(d(a) = 1\): No outgoing edges.
Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

Dijkstra:
Process \( s \): Set \( d(a) = 1, d(b) = 2, d(c) = 3 \).
Process \( a \), \( d(a) = 1 \): No outgoing edges.
Process \( b \), \( d(b) = 2 \):
Notice: argument for Dijkstra breaks for negative edges. For example.

Dijkstra:
Process $s$: Set $d(a) = 1$, $d(b) = 2$, $d(c) = 3$.
Process $a$, $d(a) = 1$: No outgoing edges.
Process $b$, $d(b) = 2$: $d(a)$ still set to 1.
Negative edges.

Notice: argument for Dijkstra breaks for negative edges. For example.

Dijkstra:
Process s: Set $d(a) = 1$, $d(b) = 2$, $d(c) = 3$.
Process $a$, $d(a) = 1$: No outgoing edges.
Process $b$, $d(b) = 2$: $d(a)$ still set to 1.
Process $c$, $d(c) = 3$:
Notice: argument for Dijkstra breaks for negative edges.

For example.

Dijkstra:
Process $s$: Set $d(a) = 1$, $d(b) = 2$, $d(c) = 3$.
Process $a$, $d(a) = 1$: No outgoing edges.
Process $b$, $d(b) = 2$: $d(a)$ still set to 1.
Process $c$, $d(c) = 3$: Set $d(b) = 0$. 
Negative edges.

Notice: argument for Dijkstra breaks for negative edges.
For example.

Dijkstra:
Process $s$: Set $d(a) = 1$, $d(b) = 2$, $d(c) = 3$.
Process $a$, $d(a) = 1$: No outgoing edges.
Process $b$, $d(b) = 2$: $d(a)$ still set to 1.
Process $c$, $d(c) = 3$: Set $d(b) = 0$.
But, can’t process $b$, again!!!
Negative edges.

Notice: argument for Dijkstra breaks for negative edges.
For example.

Dijkstra:
Process $s$: Set $d(a) = 1$, $d(b) = 2$, $d(c) = 3$.
Process $a$, $d(a) = 1$: No outgoing edges.
Process $b$, $d(b) = 2$: $d(a)$ still set to 1.
Process $c$, $d(c) = 3$: Set $d(b) = 0$.
But, can’t process $b$, again!!!
$d(a)$ still 1.
Notice: argument for Dijkstra breaks for negative edges. For example.

Dijkstra:
Process $s$: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process $a$, $d(a) = 1$: No outgoing edges.
Process $b$, $d(b) = 2$: $d(a)$ still set to 1.
Process $c$, $d(c) = 3$: Set $d(b) = 0$.
But, can’t process $b$, again!!!
$d(a)$ still 1. Should be 0
Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

Dijkstra:
Process $s$: Set $d(a) = 1$, $d(b) = 2$, $d(c) = 3$.
Process $a$, $d(a) = 1$: No outgoing edges.
Process $b$, $d(b) = 2$: $d(a)$ still set to 1.
Process $c$, $d(c) = 3$: Set $d(b) = 0$.
But, can’t process $b$, again!!!
$d(a)$ still 1. Should be 0
Problem: $d(b)$ was incorrect when processed due to negative edge.
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

Update/Bellman-Ford.

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless.

$u_1 u_2 \cdots u_k t$

If update ($s, u_1$), then ($u_1, u_2$), \ldots, and finally ($u_k, t$).

It's all good!

How???

Update!

Here, there, everywhere...

Bellman-Ford: do $n-1$ times, update all edges.

Correctness: After $i$th loop, $d(v)$ is correct for $v$ with $i$ edge shortest paths.

Time: $O(|V||E|)$
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node,
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.
Update/Bellman-Ford.

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + I (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) \(d(v)\) is correct if \(u\) is second to last node on shortest path, and \(d(u)\) correct.
2) Never makes \(d(v)\) too small.
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..
Update/Bellman-Ford.

```python
def update ((u, v)):
    dist(v) = \min \ (dist(v), dist(u) + l(u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

How???
Update!
Here, there, everywhere...

Bellman-Ford: do $n-1$ times, update all edges.

Correctness: After $i$th loop, $d(v)$ is correct for $v$ with $i$ edge shortest paths.

Time: $O(|V||E|)$
**Update/Bellman-Ford.**

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

**Properties:**

1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

![Diagram](image)

If update $(s, u_1)$,
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

If update $(s, u_1)$, then $(u_1, u_2)$,
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless.

If update $(s, u_1)$, then $(u_1, u_2)$, ...
Update/Bellman-Ford.

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

If update $(s, u_1)$, then $(u_1, u_2)$, ... and finally $(u_k, t)$. 
Update/Bellman-Ford.

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

![Graph diagram](image)

If update $(s, u_1)$, then $(u_1, u_2)$, \ldots and finally $(u_k, t)$. It's
**Update/Bellman-Ford.**

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless.

If update $(s, u_1)$, then $(u_1, u_2)$, ... and finally $(u_k, t)$. It's all
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) \( d(v) \) is correct if \( u \) is second to last node on shortest path, and \( d(u) \) correct.
2) Never makes \( d(v) \) too small. Harmless.

If update \((s, u_1)\), then \((u_1, u_2)\), \ldots and finally \((u_k, t)\). It's all good!
Update/Bellman-Ford.

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

If update $(s, u_1)$, then $(u_1, u_2)$, ... and finally $(u_k, t)$. It's all good!

How???
Update/Bellman-Ford.

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

![Graph](image)

If update $(s, u_1)$, then $(u_1, u_2)$, ... and finally $(u_k, t)$. It's all good! How??? Update!
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) \( d(v) \) is correct if \( u \) is second to last node on shortest path, and \( d(u) \) correct.
2) Never makes \( d(v) \) too small. Harmless..

If update \((s, u_1)\), then \((u_1, u_2)\), \ldots \text{ and finally } (u_k, t). \text{ It's all good!}

How???: Update! Here,
Update/Bellman-Ford.

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) \(d(v)\) is correct if \(u\) is second to last node on shortest path, and \(d(u)\) correct.
2) Never makes \(d(v)\) too small. Harmless..

![Graph](image)

If update \((s, u_1)\), then \((u_1, u_2)\), ... and finally \((u_k, t)\). It's all good!

How???

Update! Here, there,
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) \( d(v) \) is correct if \( u \) is second to last node on shortest path, and \( d(u) \) correct.
2) Never makes \( d(v) \) too small. Harmless.

\[ s \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_k \rightarrow t \]

If update \((s, u_1)\), then \((u_1, u_2)\), \ldots and finally \((u_k, t)\). It’s all good!
How??? Update! Here, there, everywhere...
**Update/Bellman-Ford.**

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

**Properties:**
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

![Graph](image)

If update $(s, u_1)$, then $(u_1, u_2)$, … and finally $(u_k, t)$. **It’s all good!**

*How??? Update! Here, there, everywhere...*

**Bellman-Ford:**

```python
Bellman-Ford:
```
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) d(v) is correct if u is second to last node on shortest path, and d(u) correct.
2) Never makes d(v) too small. Harmless..

If update (s, u₁), then (u₁, u₂), … and finally (u_k, t). It’s all good!

How?? Update! Here, there, everywhere...

Bellman-Ford:
    do n – 1 times,
**Update/Bellman-Ford.**

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) \(d(v)\) is correct if \(u\) is second to last node on shortest path, and \(d(u)\) correct.
2) Never makes \(d(v)\) too small. Harmless..

If update \((s, u_1)\), then \((u_1, u_2)\), \ldots and finally \((u_k, t)\). It's all good!

How?? Update! Here, there, everywhere...

Bellman-Ford:
- do \(n - 1\) times,
  - update all edges.

**Correctness:**
After \(i\)th loop, \(d(v)\) is correct for \(v\) with \(i\) edge shortest paths.

**Time:** \(O(|V|\cdot|E|)\)
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) \( d(v) \) is correct if \( u \) is second to last node on shortest path, and \( d(u) \) correct.
2) Never makes \( d(v) \) too small. Harmless..

If update \((s, u_1), \) then \((u_1, u_2), \ldots\) and finally \((u_k, t)\). It’s all good!

How??? Update! Here, there, everywhere...

Bellman-Ford:
    do \( n - 1 \) times,
    update all edges.

Correctness: After \( i \)th loop,
**Update/Bellman-Ford.**

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

![Diagram of shortest path](image)

If update $(s, u_1)$, then $(u_1, u_2)$, ... and finally $(u_k, t)$. *It’s all good!*

How??! Update! Here, there, everywhere...

Bellman-Ford:
- do $n – 1$ times,
  - update all edges.

**Correctness:** After $i$th loop, $d(v)$ is correct for $v$ with $i$ edge shortest paths.
Update/Bellman-Ford.

```python
def update ((u, v)):
    dist(v) = min (dist(v), dist(u) + l (u,v)).
```

In Dijkstra: Process closest unprocessed node, update neighbors.

Properties:
1) $d(v)$ is correct if $u$ is second to last node on shortest path, and $d(u)$ correct.
2) Never makes $d(v)$ too small. Harmless..

If update $(s, u_1)$, then $(u_1, u_2)$, ... and finally $(u_k, t)$. It's all good!

How??? Update! Here, there, everywhere...

Bellman-Ford:
  - do $n - 1$ times,
    - update all edges.

**Correctness:** After $i$th loop, $d(v)$ is correct for $v$ with $i$ edge shortest paths.
Time: $O(|V||E|)$
Example.
Example.
Example.
Example.
Example.
Example.
Example.
Example.
Example.
Careful: Negative Cycles.

Bellman-Ford: $d(v) \leq \text{length of i edge shortest path}$. 

Why?

No cycles!

Why?

Cycle only adds length ______. Not necessarily with negative edges.

When won’t it?

If there is a negative cycle. After $n$ iterations, some distance changes, there must be negative cycle!
Careful: Negative Cycles.

Bellman-Ford: \( d(v) \leq \text{length of } i \text{ edge shortest path.} \)
Assumes length of shortest path is at most \( n - 1 \).
Careful: Negative Cycles.

Bellman-Ford: $d(v) \leq$ length of $i$ edge shortest path. Assumess length of shortest path is at most $n - 1$.
Why?
Careful: Negative Cycles.

Bellman-Ford: \( d(v) \leq \) length of \( i \) edge shortest path. Assumes length of shortest path is at most \( n-1 \).

Why? No cycles!
Careful: Negative Cycles.

Bellman-Ford: \( d(v) \leq \) length of \( i \) edge shortest path.
Assumes length of shortest path is at most \( n - 1 \).
Why? No cycles!
Why?
Careful: Negative Cycles.

Bellman-Ford: $d(v) \leq \text{length of } i \text{ edge shortest path.}$
Assumes length of shortest path is at most $n - 1.$
Why? No cycles!
Why? Cycle only adds length
Careful: Negative Cycles.

Bellman-Ford: $d(v) \leq$ length of $i$ edge shortest path. Assumess length of shortest path is at most $n-1$.

Why? No cycles!

Why? Cycle only adds length??
Careful: Negative Cycles.

Bellman-Ford: $d(v) \leq$ length of $i$ edge shortest path. Assumes length of shortest path is at most $n - 1$.

Why? No cycles!
Why? Cycle only adds length $????$.

Not necessarily with negative edges.
Careful: Negative Cycles.

Bellman-Ford: \( d(v) \leq \text{length of } i \text{ edge shortest path} \).

Assumes length of shortest path is at most \( n - 1 \).

Why? No cycles!

Why? Cycle only adds length \( 
\).

Not necessarily with negative edges.

When won’t it?
Careful: Negative Cycles.

Bellman-Ford: \( d(v) \leq \text{length of } i \text{ edge shortest path.} \)
Assumes length of shortest path is at most \( n-1 \).
Why? No cycles!
Why? Cycle only adds length \( ???? \).
   Not necessarily with negative edges.
When won’t it?
   If there is a negative cycle.
Careful: Negative Cycles.

Bellman-Ford: $d(v) \leq$ length of $i$ edge shortest path. Assumes length of shortest path is at most $n - 1$.

Why? No cycles!

Why? Cycle only adds length ????.

  Not necessarily with negative edges.

When won’t it?

  If there is a negative cycle.

After $n$ iterations, some distance changes, there must be negative cycle!
Example: negative cycle
Example: negative cycle
Example: negative cycle
Example: negative cycle
Example: negative cycle

Graph:
- Node S connected to A with weight 10
- Node A connected to B with weight 5
- Node B connected to C with weight -10
- Node C has a weight of 5
- Node S has a weight of 9
- Node B has a weight of 15
Example: negative cycle
Example: negative cycle
Property on cycle.

For negative cycle, $C$ where $l(C) < 0$, where $v \in C$, $d(v) < \infty$, there exist edge $(u, v) \in C$, where $d(v) > d(u) + l(u, v)$. 

But, $\sum_{e = (u, v) \in C} d(v) - d(u) = 0$ since each vertex appears once positively and once negatively in the sum. Contradiction. 

Negative cycle $\Rightarrow$ update after $n$ iterations of Bellman-Ford.
For negative cycle, $C$ where $l(C) < 0$, where $v \in C$, $d(v) < \infty$, there exist edge $(u, v) \in C$, where $d(v) > d(u) + l(u, v)$.

If not, $d(v) \leq d(u) + l(u, v)$ or $d(v) - d(u) \leq l(u, v)$ for all edges in the cycle.
Property on cycle.

For negative cycle, $C$ where $l(C) < 0$, where $v \in C$, $d(v) < \infty$, there exist edge $(u, v) \in C$, where $d(v) > d(u) + l(u, v)$.

If not, $d(v) \leq d(u) + l(u, v)$ or $d(v) - d(u) \leq l(u, v)$ for all edges in the cycle.

Thus, $\sum_{e=(u,v)\in C} d(v) - d(u) \leq \sum_{e\in C} l(u,v) = l(C) < 0$. 

But, $\sum_{e=(u,v)\in C} d(v) - d(u) = 0$ since each vertex appears once positively and once negatively in the sum. Contradiction.

Negative cycle $\Rightarrow$ update after $n$ iterations of Bellman-Ford.
Property on cycle.

For negative cycle, $C$ where $l(C) < 0$, where $v \in C$, $d(v) < \infty$, there exist edge $(u, v) \in C$, where $d(v) > d(u) + l(u, v)$.

If not, $d(v) \leq d(u) + l(u, v)$ or $d(v) - d(u) \leq l(u, v)$ for all edges in the cycle.

Thus, $\sum_{e=(u, v) \in C} d(v) - d(u) \leq \sum_{e \in C} l(u, v) = l(C) < 0$.

But, $\sum_{e=(u, v) \in C} d(v) - d(u) = 0$ since each vertex appears once positively and once negatively in the sum.
For negative cycle, $C$ where $l(C) < 0$, where $v \in C$, $d(v) < \infty$, there exist edge $(u, v) \in C$, where $d(v) > d(u) + l(u, v)$.

If not, $d(v) \leq d(u) + l(u, v)$ or $d(v) - d(u) \leq l(u, v)$ for all edges in the cycle.

Thus, $\sum_{e=(u,v) \in C} d(v) - d(u) \leq \sum_{e\in C} l(u, v) = l(C) < 0$.

But, $\sum_{e=(u,v) \in C} d(v) - d(u) = 0$ since each vertex appears once positively and once negatively in the sum.

Contradiction.
For negative cycle, $C$ where $l(C) < 0$, where $v \in C$, $d(v) < \infty$, there exist edge $(u, v) \in C$, where $d(v) > d(u) + l(u, v)$.

If not, $d(v) \leq d(u) + l(u, v)$ or $d(v) - d(u) \leq l(u, v)$ for all edges in the cycle.

Thus, $\sum_{e=(u,v)\in C} d(v) - d(u) \leq \sum_{e\in C} l(u, v) = l(C) < 0$.

But, $\sum_{e=(u,v)\in C} d(v) - d(u) = 0$ since each vertex appears once positively and once negatively in the sum.

Contradiction.
Property on cycle.

For negative cycle, $C$ where $l(C) < 0$, where $v \in C$, $d(v) < \infty$, there exist edge $(u, v) \in C$, where $d(v) > d(u) + l(u, v)$.

If not, $d(v) \leq d(u) + l(u, v)$ or $d(v) - d(u) \leq l(u, v)$ for all edges in the cycle.

Thus, $\sum_{e=(u,v)\in C} d(v) - d(u) \leq \sum_{e\in C} l(u, v) = l(C) < 0$.

But, $\sum_{e=(u,v)\in C} d(v) - d(u) = 0$ since each vertex appears once positively and once negatively in the sum.

Contradiction.

Negative cycle $\implies$ update after $n$ iterations of Bellman-Ford.
DAG

Dijkstra:

Directed graph with positive edge lengths.

$O((m+n) \log n)$ time or a bit better.

Bellman-Ford:

Directed graph with arbitrary edge lengths.

$O(nm)$ time.

Also $O(nm)$ time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle?

Possible?

Not possible.

Not possible.

No cycles at all!

DAG:

Remember ...

the Alamo!

Remember ...

linearization.

Inverse post-ordering!

Remember ...

Goliad!

Remember ...

updating along path makes it all good.

Shortest path for DAG:

linearize/topological sort process nodes (and update neighbors in order.)
DAG

Dijkstra: Directed graph with positive edge lengths.
DAG

Dijkstra: Directed graph with positive edge lengths. 
\[ O((m + n) \log n) \] time
DAG

Dijkstra: Directed graph with positive edge lengths. 
\[ O((m + n) \log n) \] time or a bit better.
DAG

Dijkstra: Directed graph with positive edge lengths. \( O((m + n) \log n) \) time or a bit better.

Bellman-Ford:
DAG

Dijkstra: Directed graph with positive edge lengths. 
$O((m + n) \log n)$ time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
DAG

Dijkstra: Directed graph with positive edge lengths. 
\[ O((m + n) \log n) \] time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. 
\[ O(nm) \] time.
DAG

Dijkstra: Directed graph with positive edge lengths. $O((m + n) \log n)$ time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. $O(nm)$ time.
Also $O(nm)$ time to detect negative cycle.
DAG

Dijkstra: Directed graph with positive edge lengths. 
\(O((m + n) \log n)\) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
\(O(nm)\) time.
Also \(O(nm)\) time to detect negative cycle.

Directed acyclic graphs?
DAG

Dijkstra: Directed graph with positive edge lengths.
\[ O((m + n) \log n) \] time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
\[ O(nm) \] time.
Also \( O(nm) \) time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle?
DAG

Dijkstra: Directed graph with positive edge lengths.
\[ O((m + n) \log n) \] time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
\[ O(nm) \] time.
Also \[ O(nm) \] time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible?
DAG

Dijkstra: Directed graph with positive edge lengths. 
\(O((m + n) \log n)\) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. 
\(O(nm)\) time.
Also \(O(nm)\) time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?
DAG

Dijkstra: Directed graph with positive edge lengths. \( O((m + n) \log n) \) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. \( O(nm) \) time.
Also \( O(nm) \) time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible.
DAG

Dijkstra: Directed graph with positive edge lengths. $O((m + n) \log n)$ time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. $O(nm)$ time.
   Also $O(nm)$ time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!
DAG

Dijkstra: Directed graph with positive edge lengths.
\[ O((m + n) \log n) \] time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
\[ O(nm) \] time.
Also \[ O(nm) \] time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
DAG

Dijkstra: Directed graph with positive edge lengths.
\[ O((m + n) \log n) \] time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
\[ O(nm) \] time.
Also \[ O(nm) \] time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
Remember ...
**DAG**

Dijkstra: Directed graph with positive edge lengths. 
$O((m + n) \log n)$ time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. 
$O(nm)$ time. 
Also $O(nm)$ time to detect negative cycle.

Directed acyclic graphs? 
Negative Cycle? Possible? Not possible? 
Not possible. No cycles at all!

DAG: 
Remember ...the Alamo!
DAG

Dijkstra: Directed graph with positive edge lengths. 
$O((m + n) \log n)$ time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. 
$O(nm)$ time. 
Also $O(nm)$ time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
Remember ...the Alamo!
Remember ...
DAG

Dijkstra: Directed graph with positive edge lengths. 
$O((m + n) \log n)$ time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. 
$O(nm)$ time. 
Also $O(nm)$ time to detect negative cycle.

Directed acyclic graphs?
Negative Cycle? Possible? Not possible?
Not possible. No cycles at all!

DAG:
Remember ...the Alamo!
Remember ...linearization.
DAG

Dijkstra: Directed graph with positive edge lengths.
\(O((m + n) \log n)\) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
\(O(nm)\) time.
Also \(O(nm)\) time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
Remember ...the Alamo!
Remember ...linearization. Inverse post-ordering!
DAG

Dijkstra: Directed graph with positive edge lengths. \( O((m + n) \log n) \) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. \( O(nm) \) time.
  Also \( O(nm) \) time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
  Remember ...the Alamo!
  Remember ...linearization. Inverse post-ordering!
DAG

Dijkstra: Directed graph with positive edge lengths. $O((m + n) \log n)$ time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. $O(nm)$ time. Also $O(nm)$ time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
  Remember ...the Alamo!
  Remember ...linearization. Inverse post-ordering!

Remember ...
Dag

Dijkstra: Directed graph with positive edge lengths. 
$O((m + n) \log n)$ time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
$O(nm)$ time.
Also $O(nm)$ time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
Remember ... the Alamo!
Remember ... linearization. Inverse post-ordering!

Remember ... Goliad!
DAG

Dijkstra: Directed graph with positive edge lengths.  
\( O((m + n)\log n) \) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.  
\( O(nm) \) time.  
Also \( O(nm) \) time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
  Remember ...the Alamo!
  Remember ...linearization. Inverse post-ordering!

Remember ...Goliad!
Remember ...
DAG

Dijkstra: Directed graph with positive edge lengths. 
\(O((m + n) \log n)\) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. 
\(O(nm)\) time. 
Also \(O(nm)\) time to detect negative cycle.

Directed acyclic graphs?
Negative Cycle? Possible? Not possible?
Not possible. No cycles at all!

DAG:
Remember ...the Alamo!
Remember ...linearization. Inverse post-ordering!

Remember ...Goliad!
Remember ...updating along path makes it all good.
DAG

Dijkstra: Directed graph with positive edge lengths.
\( O((m + n) \log n) \) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
\( O(nm) \) time.
Also \( O(nm) \) time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
Remember ...the Alamo!
Remember ...linearization. Inverse post-ordering!

Remember ...Goliad!
Remember ...updating along path makes it all good.

Shortest path for DAG:
DAG

Dijkstra: Directed graph with positive edge lengths. \( O((m + n) \log n) \) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths. \( O(nm) \) time.
Also \( O(nm) \) time to detect negative cycle.

Directed acyclic graphs?
Negative Cycle? Possible? Not possible?
Not possible. No cycles at all!

DAG:
Remember ...the Alamo!
Remember ...linearization. Inverse post-ordering!

Remember ...Goliad!
Remember ...updating along path makes it all good.

Shortest path for DAG:
linearize/topological sort
DAG

Dijkstra: Directed graph with positive edge lengths.
\( O((m+n)\log n) \) time or a bit better.

Bellman-Ford: Directed graph with arbitrary edge lengths.
\( O(nm) \) time.
Also \( O(nm) \) time to detect negative cycle.

Directed acyclic graphs?

Negative Cycle? Possible? Not possible?

Not possible. No cycles at all!

DAG:
Remember ...the Alamo!
Remember ...linearization. Inverse post-ordering!

Remember ...Goliad!
Remember ...updating along path makes it all good.

Shortest path for DAG:
linearize/topological sort
process nodes (and update neighbors in order.)
Every path looks like this in a topological order.
Every path looks like this in a topological order.
Every path looks like this in a topological order.

The vertices (and edges) along path are processed in order.
DAG

\[
\begin{align*}
\text{Process } a, & \quad d(a) = 0, \\
\text{Process } b, & \quad d(b) = 0, \\
\text{Process } c, & \quad d(c) = 3, \\
\text{Process } d, & \quad d(c) = 3, \\
\text{Process } e, & \quad d(c) = 3.
\end{align*}
\]
Process $s$, $d(s) = 0$: 

DAG
Process \( s, d(s) = 0 \): Updates \( d(c) = 3 \),
Process $s$, $d(s) = 0$: Updates $d(c) = 3$, $d(b) = 2$, $d(a) = 0$. 

DAG
Process $s$, $d(s) = 0$: Updates $d(c) = 3$, $d(b) = 2$, $d(a) = 1$. 
Process $s$, $d(s) = 0$: Updates $d(c) = 3$, $d(b) = 2$, $d(a) = 1$.
Process $c$, $d(c) = 3$: 

![DAG Diagram](image-url)
Process $s$, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process $c$, $d(c) = 3$: Update $d(b) = 0$. 

DAG
Process $s$, $d(s) = 0$: Updates $d(c) = 3$, $d(b) = 2$, $d(a) = 1$.
Process $c$, $d(c) = 3$: Update $d(b) = 0$.
Process $b$, $d(b) = 0$: 
Process $s$, $d(s) = 0$: Updates $d(c) = 3$, $d(b) = 2$, $d(a) = 1$.

Process $c$, $d(c) = 3$: Update $d(b) = 0$.

Process $b$, $d(b) = 0$: $d(a) = 0$. 
Process $s$, $d(s) = 0$: Updates $d(c) = 3$, $d(b) = 2$, $d(a) = 1$.
Process $c$, $d(c) = 3$: Update $d(b) = 0$.
Process $b$, $d(b) = 0$: $d(a) = 0$.
Process $a$, $d(a) = 0$. 
Process $s$, $d(s) = 0$: Updates $d(c) = 3$, $d(b) = 2$, $d(a) = 1$.
Process $c$, $d(c) = 3$: Update $d(b) = 0$.
Process $b$, $d(b) = 0$: $d(a) = 0$.
Process $a$, $d(a) = 0$.
Done.
Dijkstra's Direct Inductive Proof.

Know distance to $S$.

Inv: $d(v) - \text{length of path through } S$.

Smallest $d(v)$ is correct, add to $S$.

Updating neighbors of $v$ enforces Inv.

Bellman-Ford: Negative weights.

Dijkstra doesn't work.

Update edge $(u, v)$:

$$d(v) = \min(d(v), d(u) + l(u, v))$$

Update all edges $|V| - 1$ times.

Paths of length $k$ correct after iteration $k$.

DAG: linearize and process vertices in order.

Updates of edges in order along path.
Dijkstra’s Direct Inductive Proof.
Know distance to $S$.
Inv: $d(v)$ - length of path through $S$.
Smallest $d(v)$ is correct, add to $S$
Updating neighbors of $v$ enforces Inv.
Dijkstra’s Direct Inductive Proof.
Know distance to S.
Inv: \( d(v) \) - length of path through S.
Smallest \( d(v) \) is correct, add to S
Updating neighbors of \( v \) enforces Inv.

Bellman-Ford: Negative weights.
Dijkstra doesn’t work.
Update edge \((u, v)\): \( d(v) = \min(d(v), d(u) + l(u, v)) \).
Update all edges \( |V| - 1 \) times.
Paths of length \( k \) correct after iteration \( k \).
Dijkstra’s Direct Inductive Proof.
Know distance to $S$.
Inv: $d(v)$ - length of path through $S$.
Smallest $d(v)$ is correct, add to $S$
Updating neighbors of $v$ enforces Inv.

Bellman-Ford: Negative weights.
Dijkstra doesn’t work.
Update edge $(u, v)$: $d(v) = \min(d(v), d(u) + l(u, v))$.
Update all edges $|V| - 1$ times.
Paths of length $k$ correct after iteration $k$.

DAG:
linearize and process vertices in order.
Updates of edges in order along path.
Have a ...

Good Weekend!