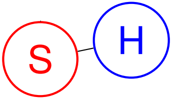
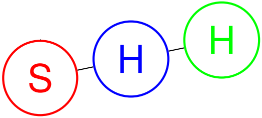
CS 170: Algorithms

CS 170: Algorithms

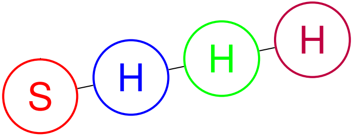
CS 170: Algorithms



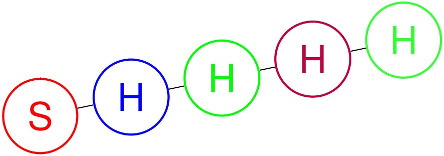
CS 170: Algorithms



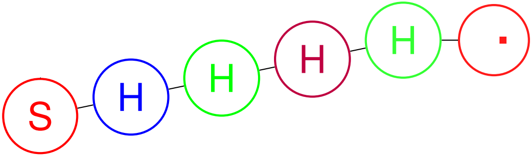
CS 170: Algorithms



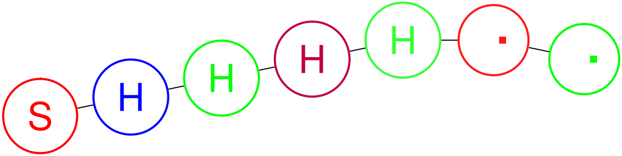
CS 170: Algorithms



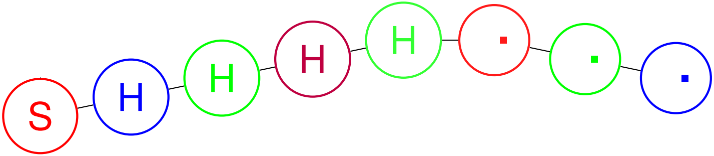
CS 170: Algorithms



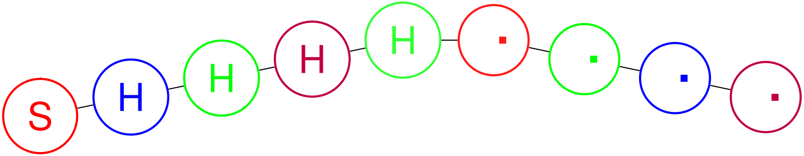
CS 170: Algorithms



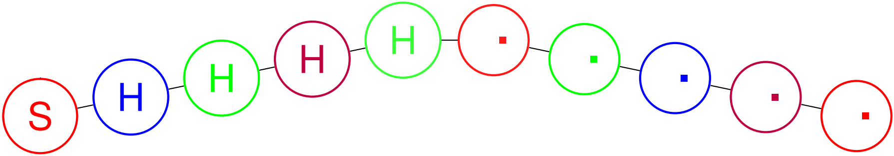
CS 170: Algorithms



CS 170: Algorithms



CS 170: Algorithms



Lecture in a minute

Lecture in a minute

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.

Lecture in a minute

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.

Lecture in a minute

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.

foreach v: $d(v) = \infty$.

Dijkstra's Algorithm.

foreach 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.

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))

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:

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.

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.

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.

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.

Binary heap: $O((|V| + |E|)\log|V|)$

Alt Proof.

Dijkstra:

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.”

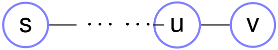
Alt Proof.

Dijkstra:

"Know distance to processed nodes, R."

"Add node v closest to s outside of R."

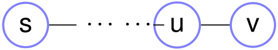
Closest node v has path...



Alt Proof.

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

Closest node *v* has path...

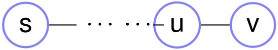


u in *R*.

Alt Proof.

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

Closest node v has path...

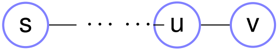


u in R. Since v is closest node not in R.

Alt Proof.

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

Closest node v has path...

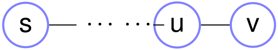


u in R. Since v is closest node not in R.
$d(u)$ correct by induction.

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

Alt Proof.

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

Closest node v has path...

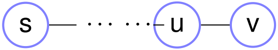
s $-\!\!-\; \cdots\;\; \cdots\!-$ u $-\!\!-$ v

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

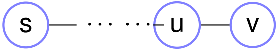
s — $\cdots$ $\cdots$ — u — v

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



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...

s — $\cdots$ $\cdots$ — 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.
$d(v) \leq d(u) + l(u, v)$. 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.

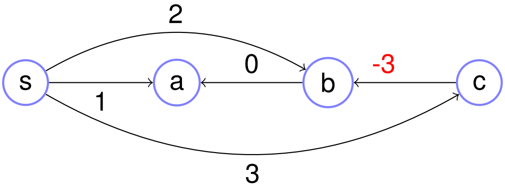
Notice: argument for Dijkstra breaks for negative edges.

For example.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

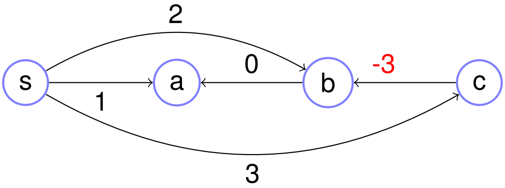


Dijkstra:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

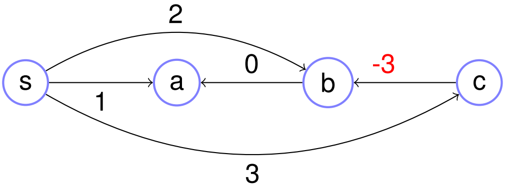


Dijkstra:
Process *s*:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

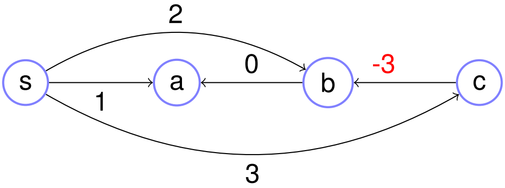


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

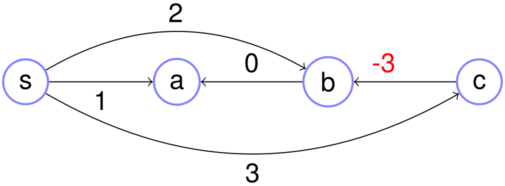


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

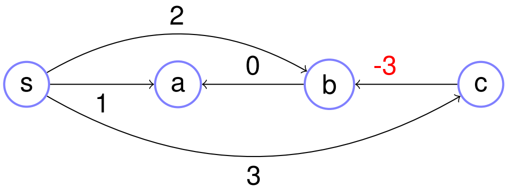


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

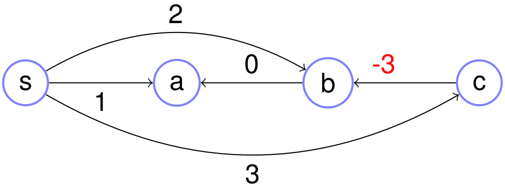


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

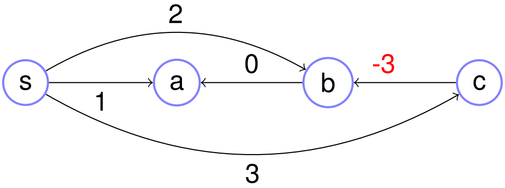


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

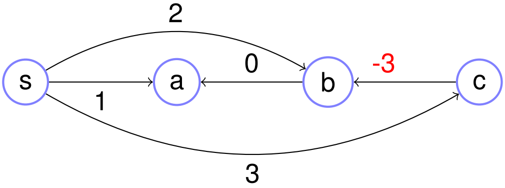


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

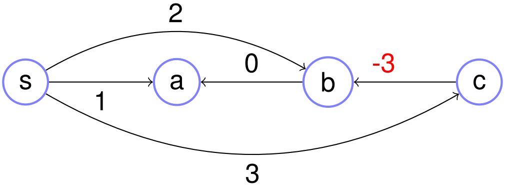


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

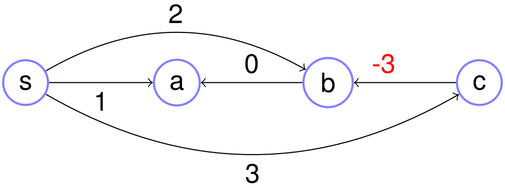


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

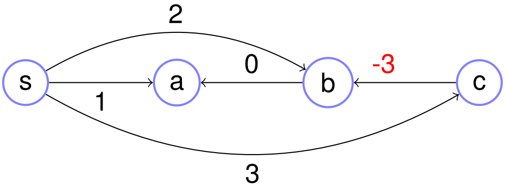


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!
$d(a)$ still 1.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

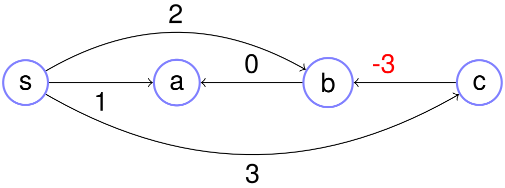


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!
$d(a)$ still 1. Should be 0

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.

Update/Bellman-Ford.

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

Update/Bellman-Ford.

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

In Dijkstra: Process closest unprocessed node,

Update/Bellman-Ford.

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.

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.

Update/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.

Update/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..

Update/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..

$$s \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \cdots \longrightarrow u_k \longrightarrow t$$

Update/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..

$$s \to u_1 \to u_2 \to \cdots \to u_k \to t$$

If update (s, u_1),

Update/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..

$s \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \rightarrow t$

If update (s, u_1), then (u_1, u_2),

Update/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..

$$s \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \cdots \longrightarrow u_k \longrightarrow t$$

If update (s, u_1), then (u_1, u_2), ...

Update/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..

$$s \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \rightarrow t$$

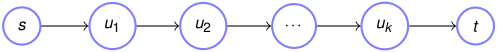
If update (s, u_1), then (u_1, u_2), ... and finally (u_k, t).

Update/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, u_1), then (u_1, u_2), ... and finally (u_k, t). It's

Update/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..

$$s \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \cdots \longrightarrow u_k \longrightarrow t$$

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

Update/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..

$$s \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \cdots \longrightarrow u_k \longrightarrow t$$

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

Update/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..

$$s \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \cdots \longrightarrow u_k \longrightarrow t$$

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

Update/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..

$$s \to u_1 \to u_2 \to \cdots \to u_k \to t$$

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

Update/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..

$$s \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \cdots \longrightarrow u_k \longrightarrow t$$

If update (s, u_1), then (u_1, u_2), ...and finally (u_k, t). It's all good!
How??? Update! Here,

Update/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..

$$s \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \rightarrow t$$

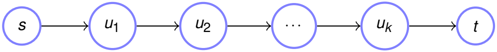
If update (s, u_1), then (u_1, u_2), ... and finally (u_k, t). It's all good!
How??? Update! Here, there,

Update/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, u_1), then (u_1, u_2), ...and finally (u_k, t). It's all good!
How??? Update! Here, there, everywhere...

Update/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..

$$s \to u_1 \to u_2 \to \cdots \to u_k \to t$$

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:

Update/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..

$$s \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \cdots \longrightarrow u_k \longrightarrow t$$

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/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..

$$s \to u_1 \to u_2 \to \cdots \to u_k \to t$$

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...

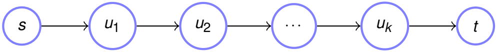
Bellman-Ford:
  do $n - 1$ times,
   update all edges.

Update/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,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 ith loop,

Update/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..

$$s \longrightarrow u_1 \longrightarrow u_2 \longrightarrow \cdots \longrightarrow u_k \longrightarrow t$$

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 ith loop, $d(v)$ is correct for v with i edge shortest paths.

Update/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..

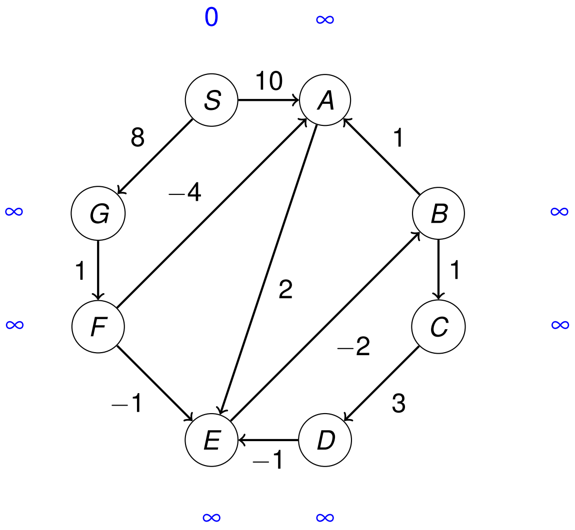
$$s \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \rightarrow t$$

If update (s, u_1), then (u_1, u_2), ...and finally (u_k, t). It's all good!
How??? Update! Here, there, everywhere...

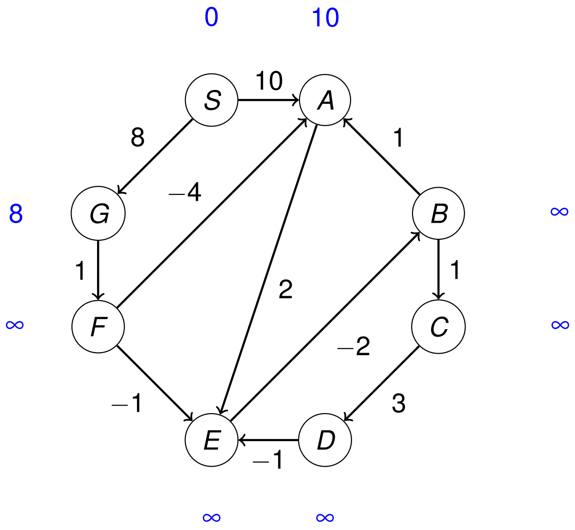
Bellman-Ford:
  do $n-1$ times,
    update all edges.

Correctness: After ith loop, $d(v)$ is correct for v with i edge shortest paths.
Time: $O(|V||E|)$

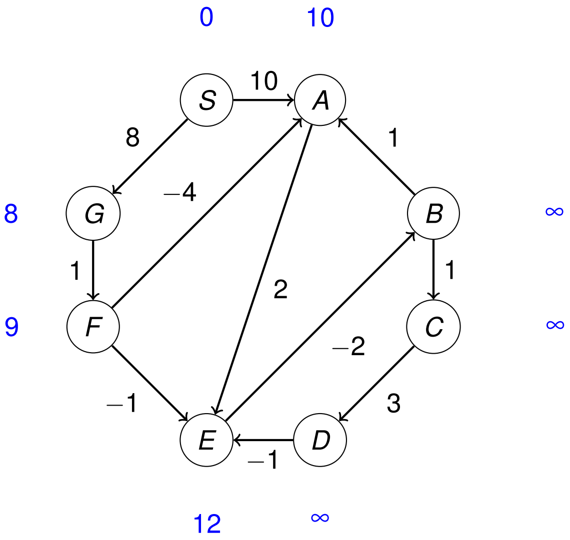
Example.



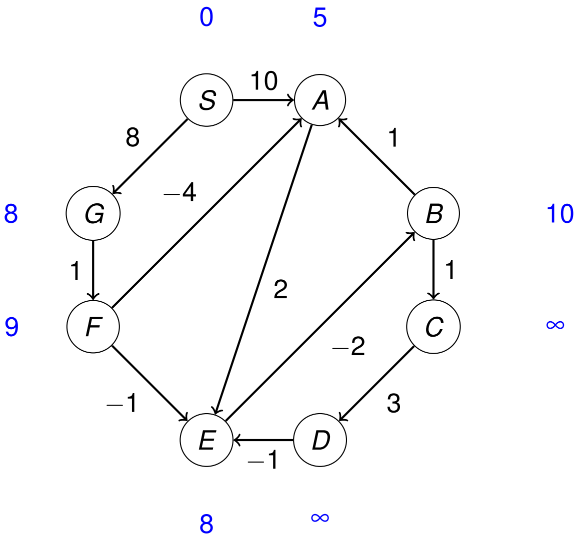
Example.



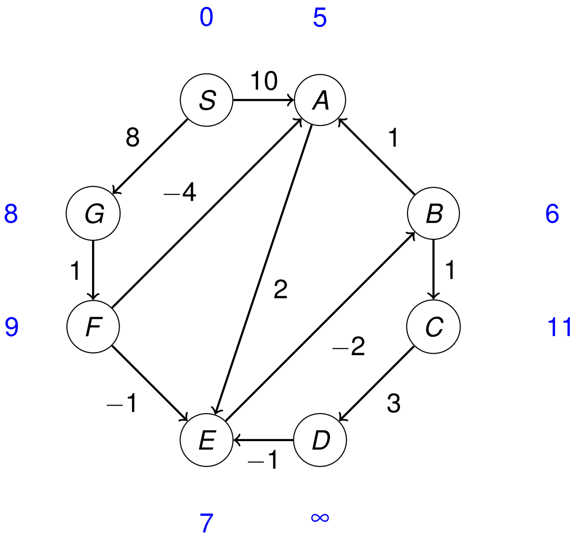
Example.



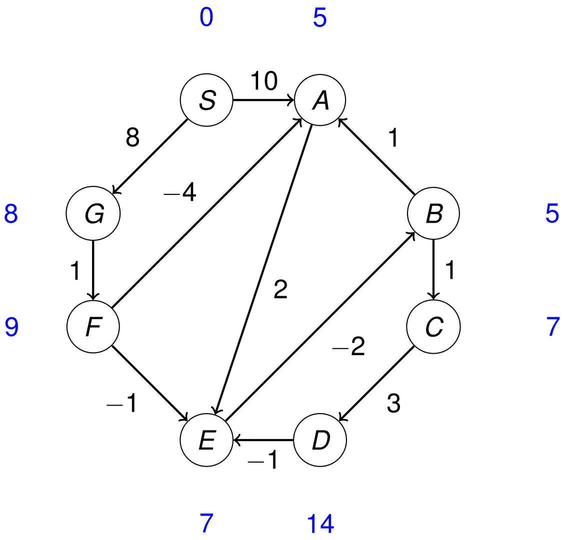
Example.



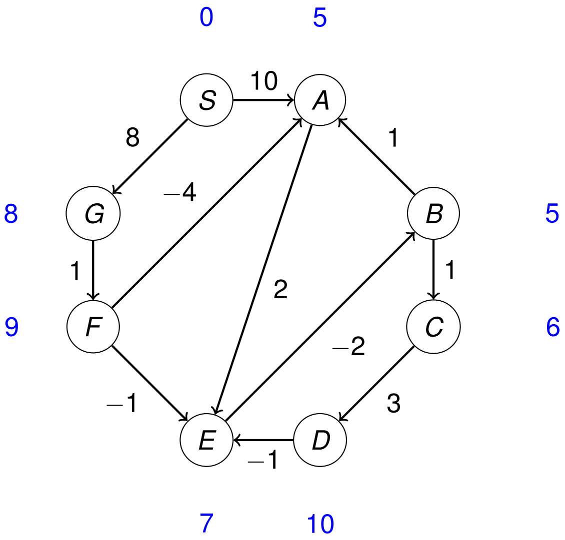
Example.



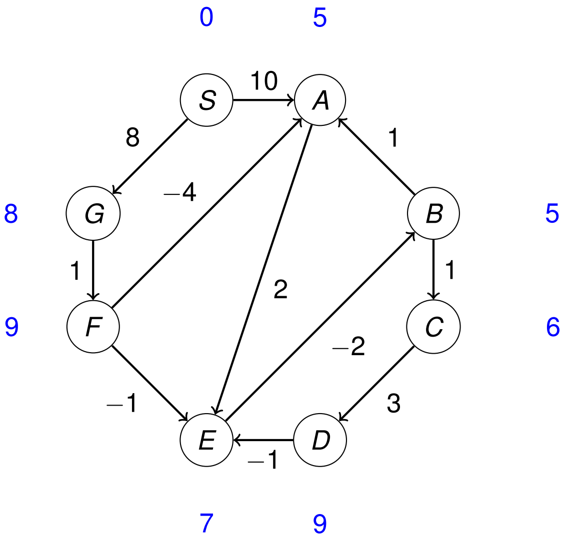
Example.



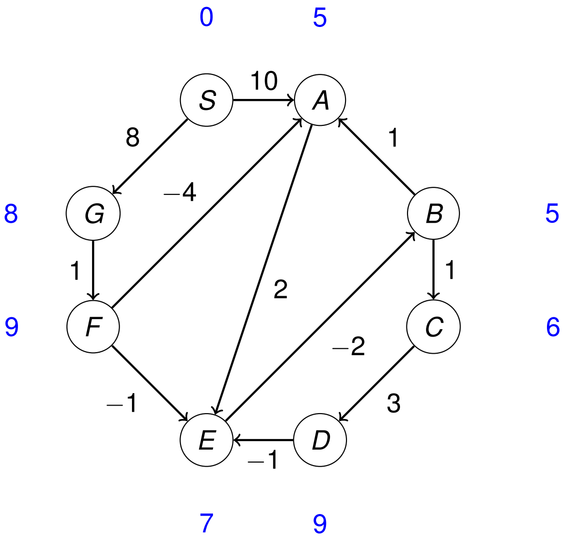
Example.



Example.



Example.



Careful: Negative Cycles.

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

Careful: Negative Cycles.

Bellman-Ford: $d(v) \leq$ length of i 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.

Assumes 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: $d(v) \leq$ length of i 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.

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.

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: $d(v) \leq$ 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?

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.

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 ????.

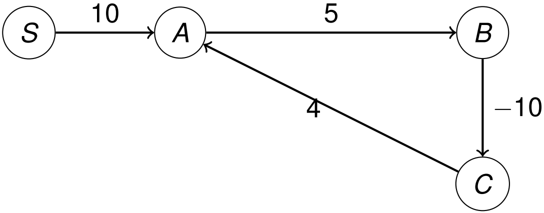
Not necessarily with negative edges.

When won't it?

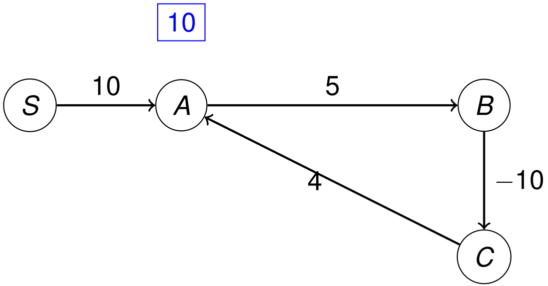
If there is a negative cycle.

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

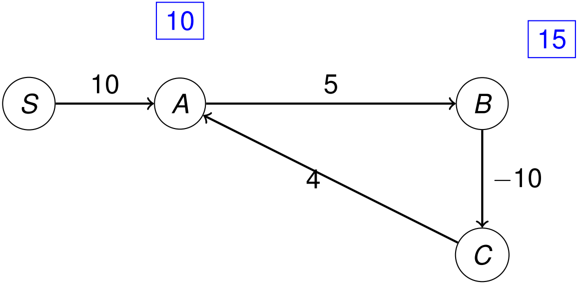
Example: negative cycle



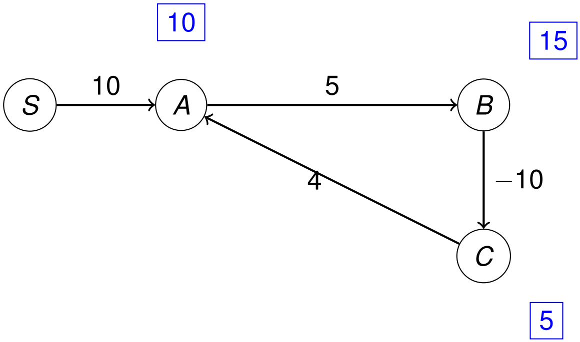
Example: negative cycle



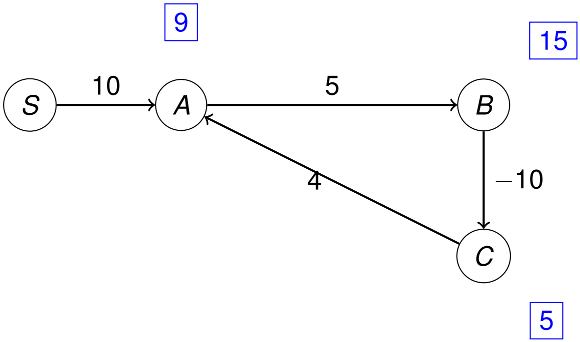
Example: negative cycle



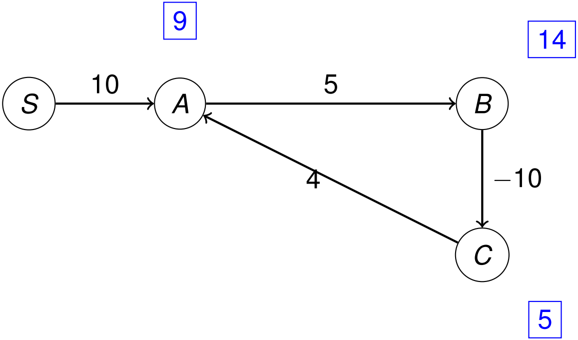
Example: negative cycle



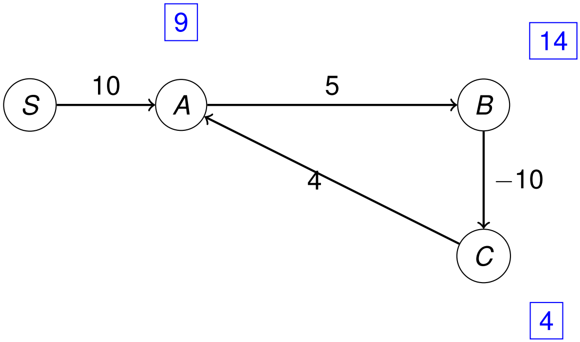
Example: negative cycle



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 $d(v) > d(u) + l(u, v)$.

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.

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

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.

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.

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. $\square$

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. $\square$

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

DAG

Dijkstra:

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.

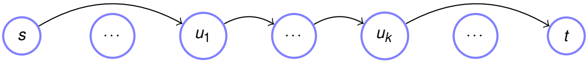
Shortest path for DAG:
linearize/topological sort
process nodes (and update neighbors in order.)

DAG

Every path looks like this in a topological order.

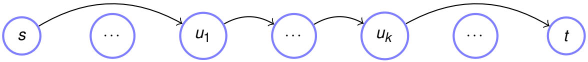
DAG

Every path looks like this in a topological order.



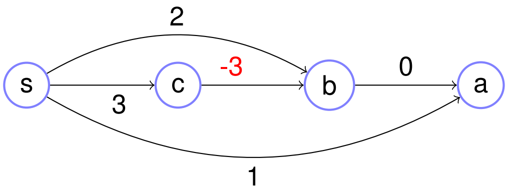
DAG

Every path looks like this in a topological order.

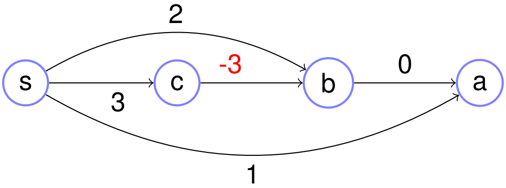


The vertices (and edges) along path are processed in order.

DAG

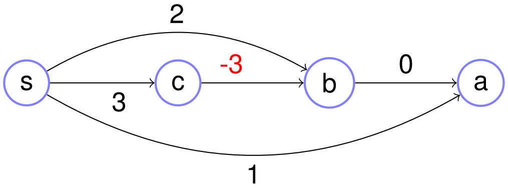


DAG



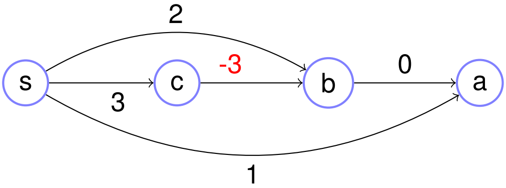
Process s, $d(s) = 0$:

DAG



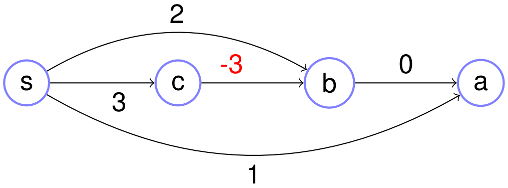
Process s, $d(s) = 0$: Updates $d(c) = 3$,

DAG



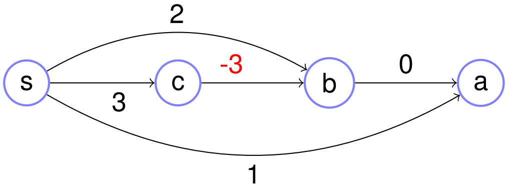
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2,$

DAG



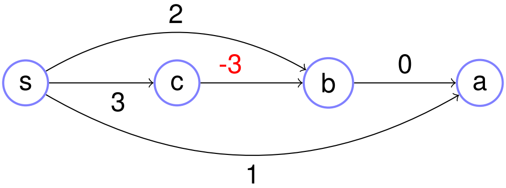
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.

DAG



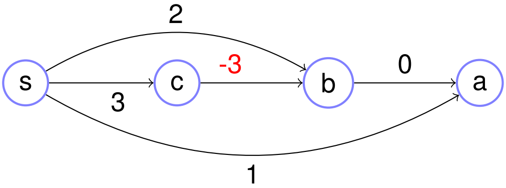
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$:

DAG



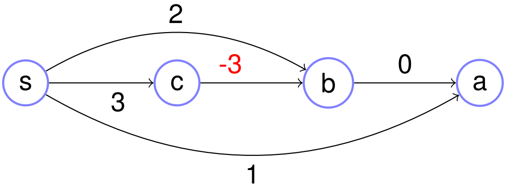
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.

DAG



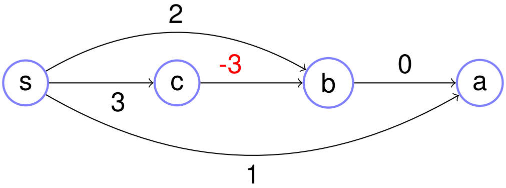
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.
Process b, $d(b) = 0$:

DAG



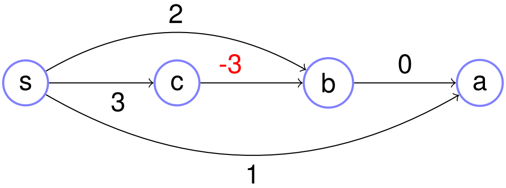
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.
Process b, $d(b) = 0$: $d(a) = 0$.

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, $d(b) = 0$: $d(a) = 0$.
Process a, $d(a) = 0$.

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, $d(b) = 0$: $d(a) = 0$.
Process a, $d(a) = 0$.
Done.

Lecture in a minute

Lecture in a minute

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.

Lecture in a minute

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.

Lecture in a minute

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!