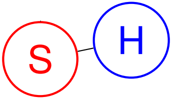
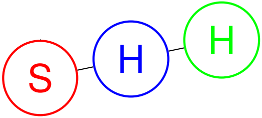
CS 170: Algorithms

CS 170: Algorithms

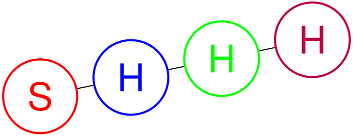
CS 170: Algorithms



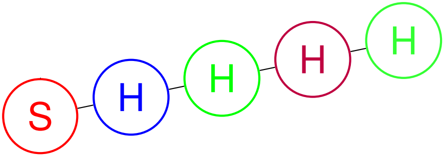
CS 170: Algorithms



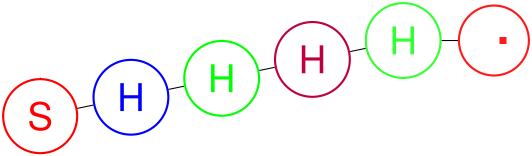
CS 170: Algorithms



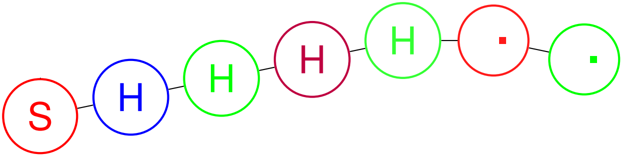
CS 170: Algorithms



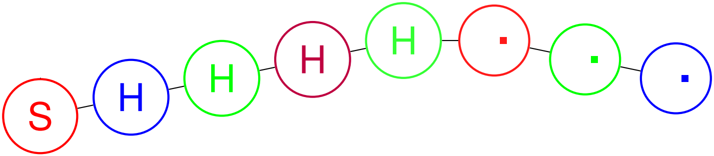
CS 170: Algorithms



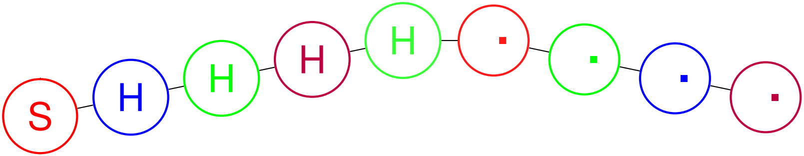
CS 170: Algorithms



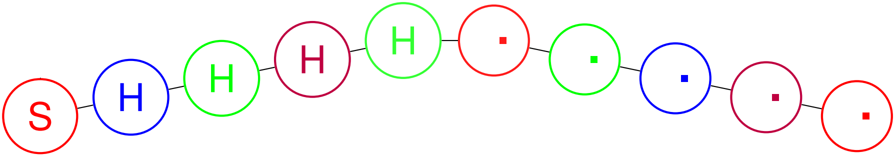
CS 170: Algorithms



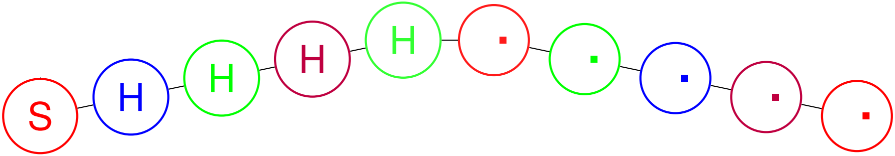
CS 170: Algorithms



CS 170: Algorithms

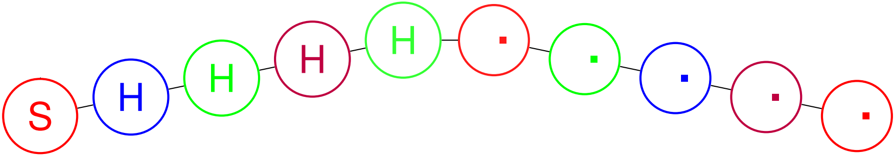


CS 170: Algorithms



No laptops please.

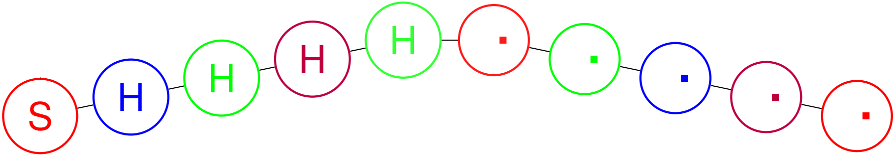
CS 170: Algorithms



No laptops please.

Thank you

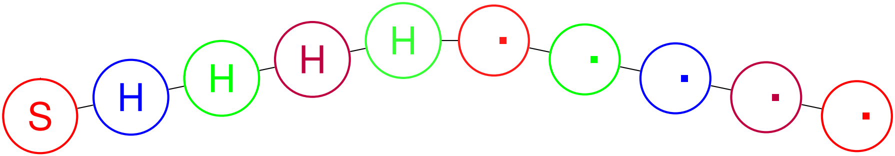
CS 170: Algorithms



No laptops please.

Thank you !

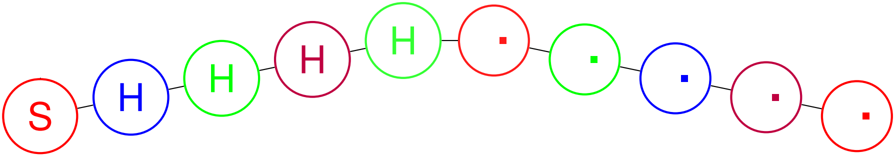
CS 170: Algorithms



No laptops please.

Thank you ! !

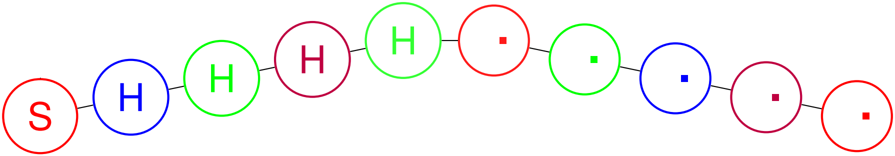
CS 170: Algorithms



No laptops please.

Thank you ! ! !

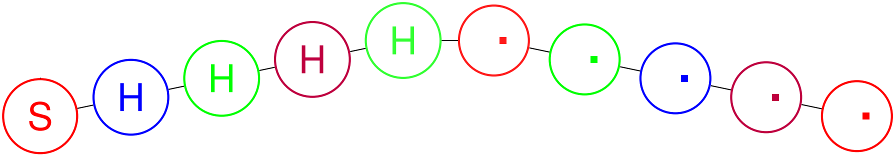
CS 170: Algorithms



No laptops please.

Thank you ! ! ! !

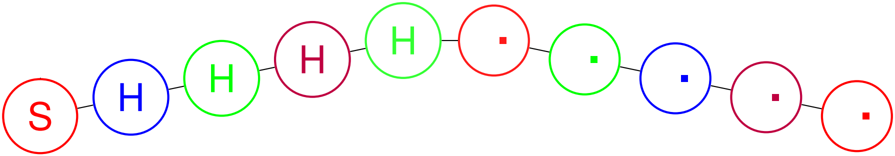
CS 170: Algorithms



No laptops please.

Thank you ! ! ! ! !

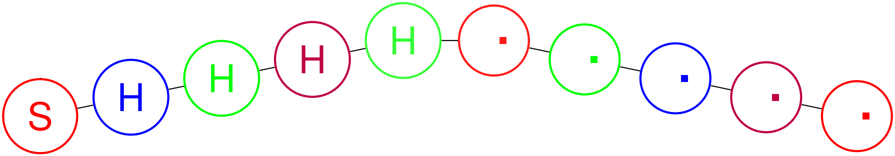
CS 170: Algorithms



No laptops please.

Thank you ! ! ! ! ! !

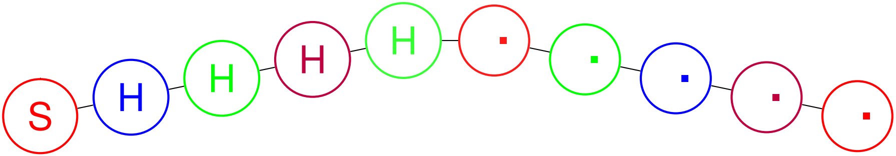
CS 170: Algorithms



No laptops please.

Thank you ! ! ! ! ! ! !

CS 170: Algorithms



No laptops please.

Thank you ! ! ! ! ! ! ! !

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

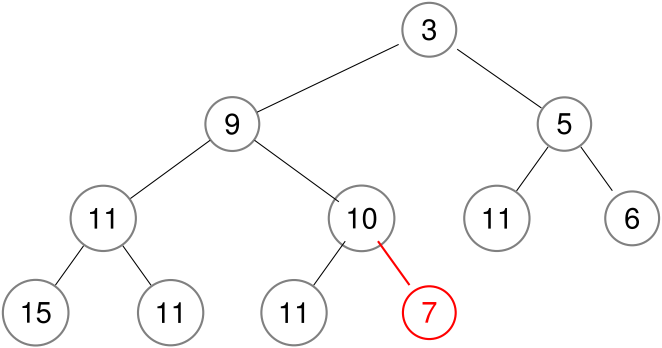
Binary Heap.

Heap[1]: bigger children.

[1] values only

Binary Heap.

Heap[1]: bigger children.
$\Longrightarrow$ smallest at root.

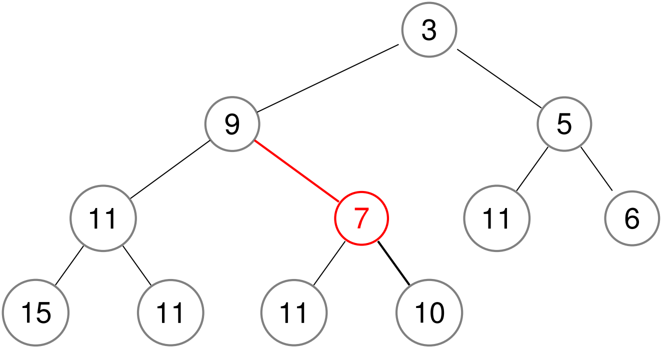


[1] values only

Binary Heap.

Heap[1]: bigger children.

$\implies$ smallest at root.



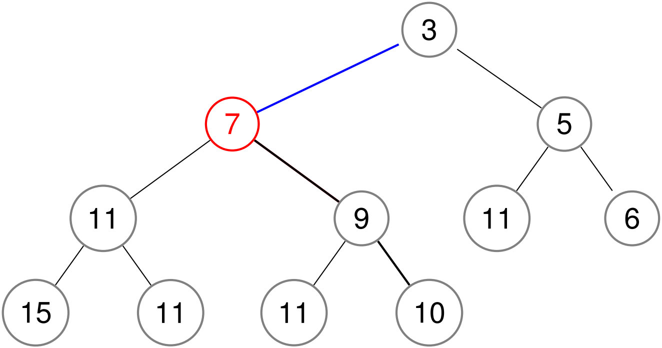
Insert(7):

[1] values only

Binary Heap.

Heap[1]: bigger children.

$\Longrightarrow$ smallest at root.



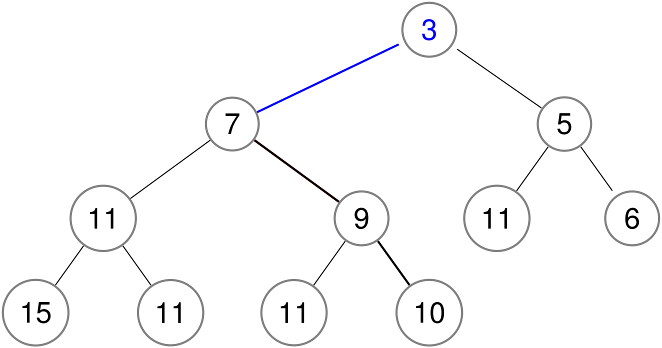
Insert(7): Bubble up: check parent.

[1] values only

Binary Heap.

Heap[1]: bigger children.

$\implies$ smallest at root.



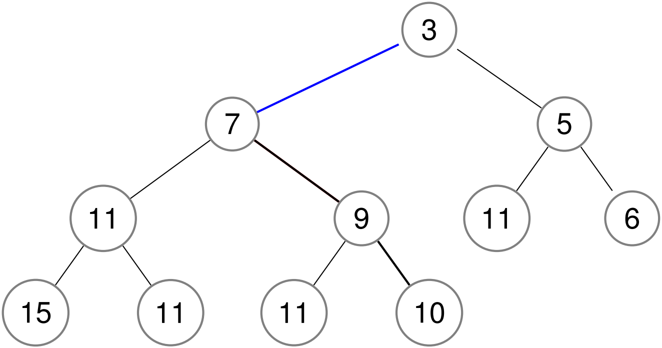
Insert(7): Bubble up: check parent. . **depth** comp.

[1] values only

Binary Heap.

Heap[1]: bigger children.

$\implies$ smallest at root.



Insert(7): Bubble up: check parent. . **depth** comp.

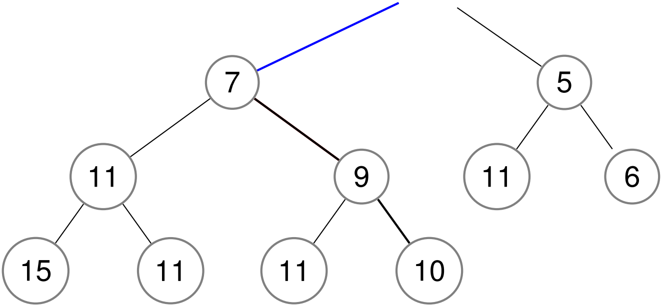
DeleteMin:

[1] values only

Binary Heap.

Heap[1]: bigger children.

$\Longrightarrow$ smallest at root.



Insert(7): Bubble up: check parent. . **depth** comp.

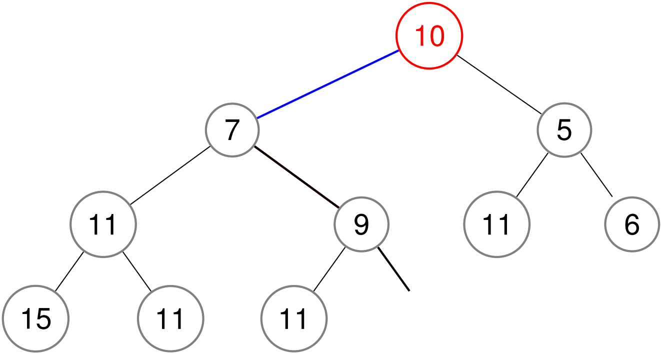
DeleteMin:

[1] values only

Binary Heap.

Heap[1]: bigger children.

$\Longrightarrow$ smallest at root.



Insert(7): Bubble up: check parent. . **depth** comp.

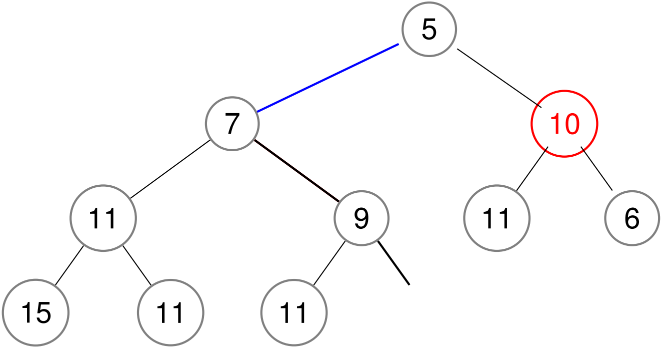
DeleteMin: Replace.

[1]values only

Binary Heap.

Heap[1]: bigger children.

$\Longrightarrow$ smallest at root.



Insert(7): Bubble up: check parent. . **depth** comp.

DeleteMin: Replace. Bubble down: check **both** children..

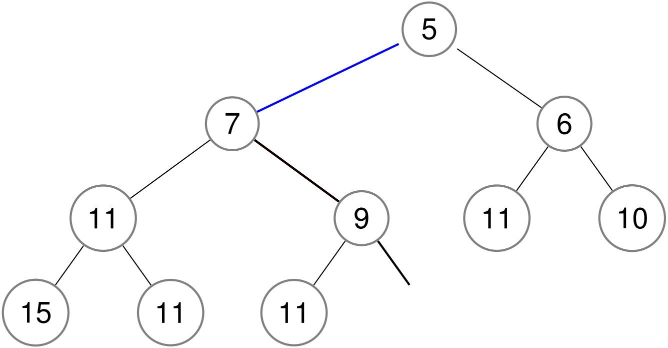
---

[1] values only

Binary Heap.

Heap[1]: bigger children.

$\Longrightarrow$ smallest at root.



Insert(7): Bubble up: check parent. . **depth** comp.

DeleteMin: Replace. Bubble down: check **both** children..

2$\times$ **depth** – comparisons.

[1] values only

d-ary heap

Degree – d, Depth – $\log_d n$.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n/\log d$.
DeleteMin – $d \log n/\log d$. (Check all children.)

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice:

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n/\log d$.
DeleteMin – $d \log n/\log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n/\log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n/\log d)$ each.

$O(|V| d \log n/\log d + |E| \log n/\log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n/\log d$.
DeleteMin – $d \log n/\log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n/\log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n/\log d)$ each.

$O(|V| d \log n/\log d + |E| \log n/\log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n/\log d)$

For dense graphs it approaches linear.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n/\log d$.
DeleteMin – $d \log n/\log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n/\log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n/\log d)$ each.

$O(|V| d \log n/\log d + |E| \log n/\log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n/\log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.

$O(|V| \log |V| + |E|)$.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.

$O(|V| \log |V| + |E|)$.
Linear for moderately dense graphs!

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."

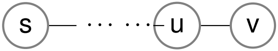
Alt Proof.

Dijkstra:

"Know distance to processed nodes, R."

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

Closest node v has path...

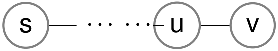


Alt Proof.

Dijkstra:

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

Closest node *v* has path...



u in *R*.

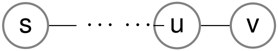
Alt Proof.

Dijkstra:

"Know distance to processed nodes, R."

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

Closest node v has path...

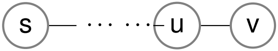


u in R. Since v is closest node not in R.

Alt Proof.

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

Closest node v has path...

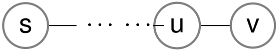


u in R. Since v is closest node not in R.
$d(u)$ correct by induction.

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...

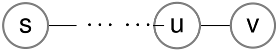
s — $\cdots$ $\cdots$ — u — v

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.

Alt Proof.

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

Closest node v has path...

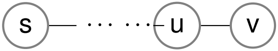


u in R. Since v is closest node not in R.
$d(u)$ correct by induction.
  $d(u)$ corresponds to the length of a shortest path.
$d(v) \leq d(u) + l(u, v)$. Since u was processed by Algorithm.
$d(v)$ corresponds to length of path.
  Set by some u, which corresponds to path by induction plus an edge (u', v').

Alt Proof.

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

Closest node v has path...



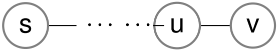
u in R. Since v is closest node not in R.
$d(u)$ correct by induction.
 $d(u)$ corresponds to the length of a shortest path.
$d(v) \leq d(u) + l(u, v)$. Since u was processed by Algorithm.
$d(v)$ corresponds to length of path.
 Set by some u, which corresponds to path by induction plus an edge (u', v').

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...



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.

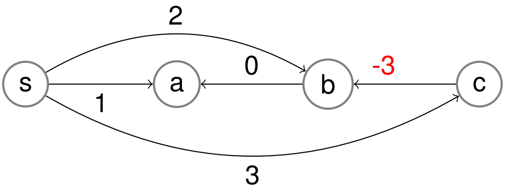
Notice: argument for Dijkstra breaks for negative edges.

For example.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

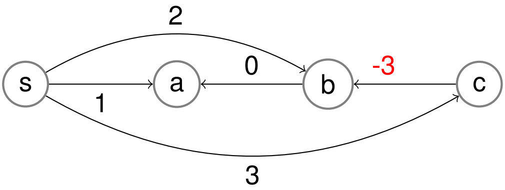


Dijkstra:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

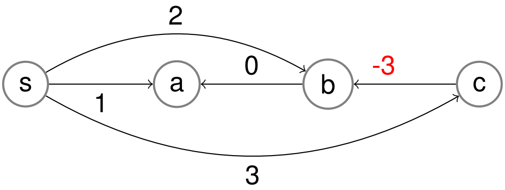


Dijkstra:
Process s:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

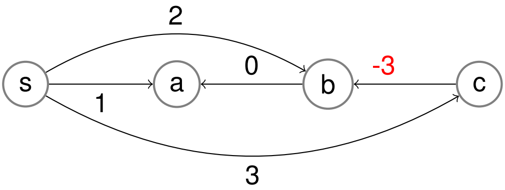


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

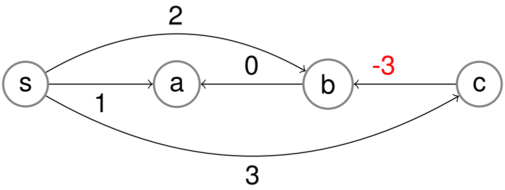


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

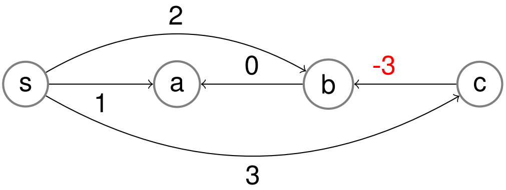


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

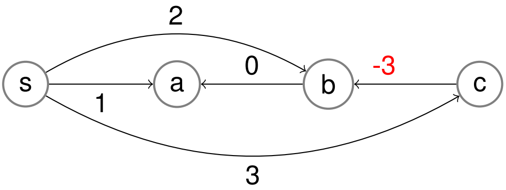


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

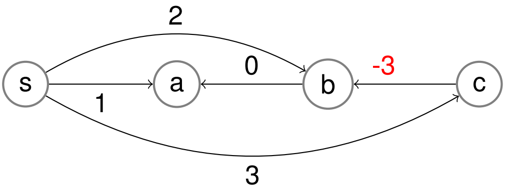


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

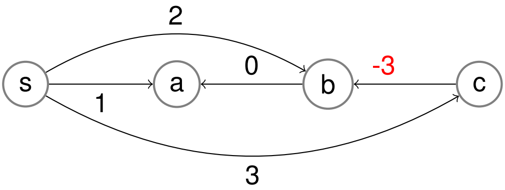


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

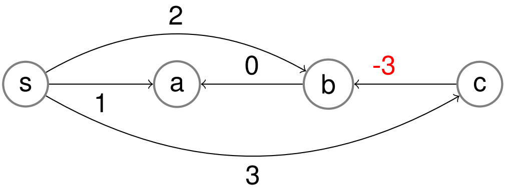


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

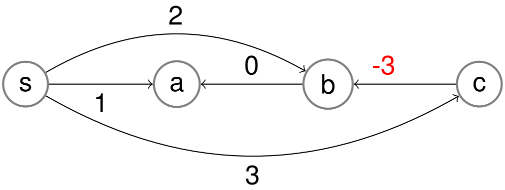


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

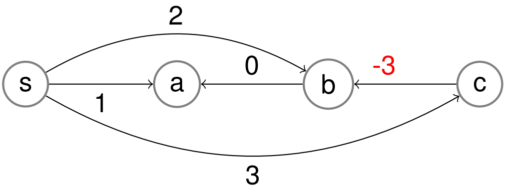


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!
$d(a)$ still 1.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

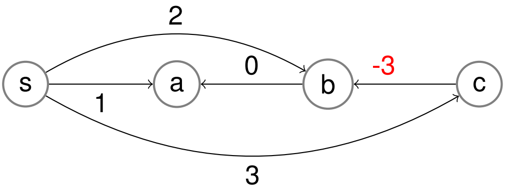


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!
$d(a)$ still 1. Should be 0

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..

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

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..

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

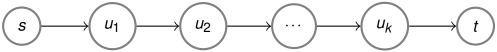
If update (s, u_1),

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..



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..

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

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..

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

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

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..

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

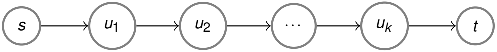
If update (s, u_1), then (u_1, u_2), ... and finally (u_k, t). It's

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

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!

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/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..

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

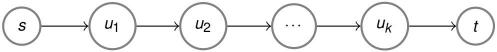
If update (s, u_1), then (u_1, u_2), ... and finally (u_k, t). It's all good!
How??? Update!

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..



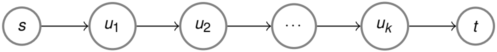
If update (s, u_1), then (u_1, u_2), $\ldots$ and finally (u_k, t). It's all good!
How??? Update! Here,

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,

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...

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! 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..

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

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

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

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

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

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

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

If update (s,u_1), then (u_1,u_2), ... and finally (u_k,t). It's all good!
How??? Update! 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: $O(|V||E|)$

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!

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

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?

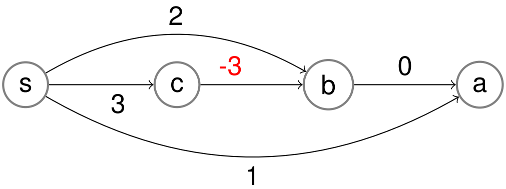
Not possible. No cycles at all!

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

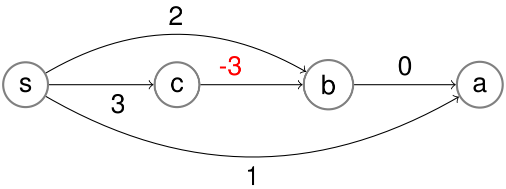
Remember ...Goliad!
Remember ...updating along path makes it all good.

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

DAG

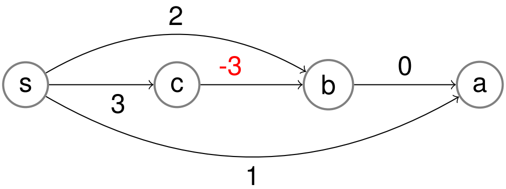


DAG



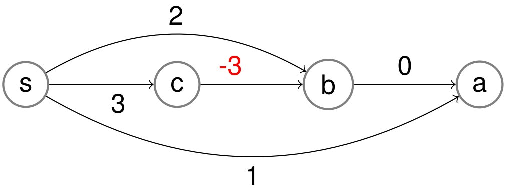
Process s, $d(s) = 0$:

DAG



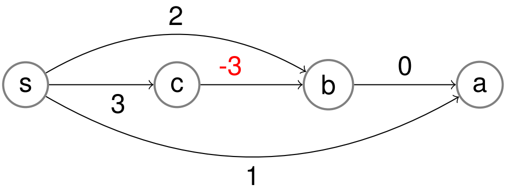
Process s, $d(s) = 0$: Updates $d(c) = 3$,

DAG



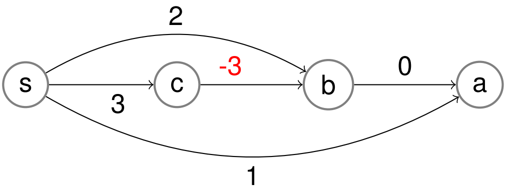
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2,$

DAG



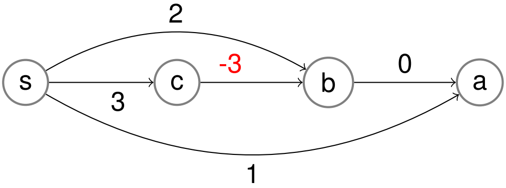
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.

DAG



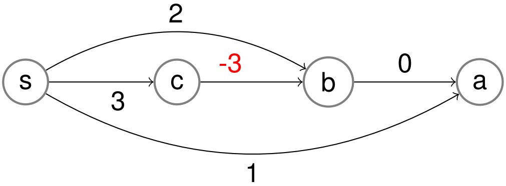
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$:

DAG



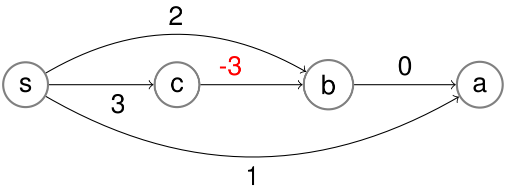
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.

DAG



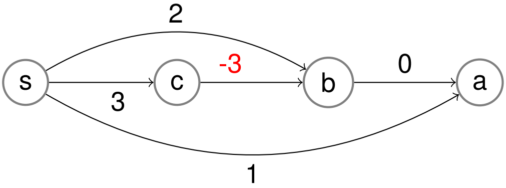
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.
Process b, $d(b) = 0$:

DAG



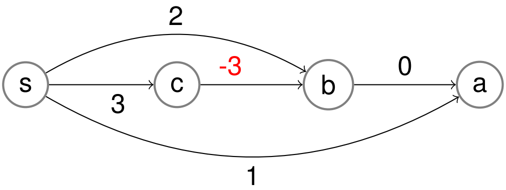
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.
Process b, $d(b) = 0$: $d(a) = 0$.

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.

See you

See you ..on Monday.