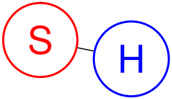
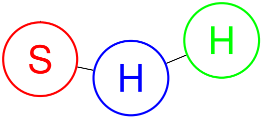
CS 170: Algorithms

CS 170: Algorithms

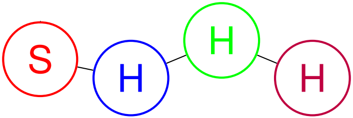
CS 170: Algorithms



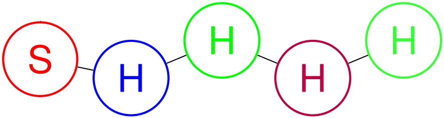
CS 170: Algorithms



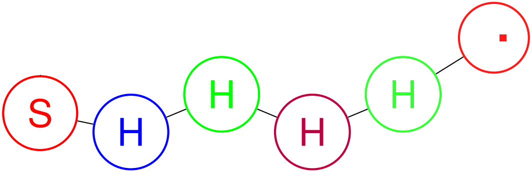
CS 170: Algorithms



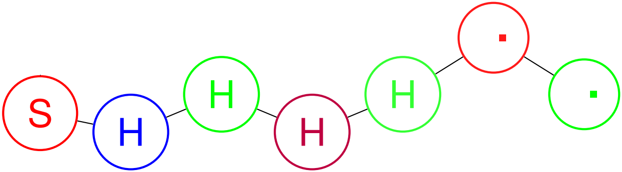
CS 170: Algorithms



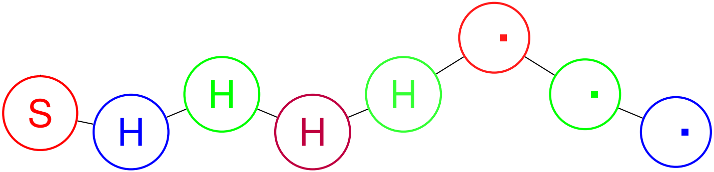
CS 170: Algorithms



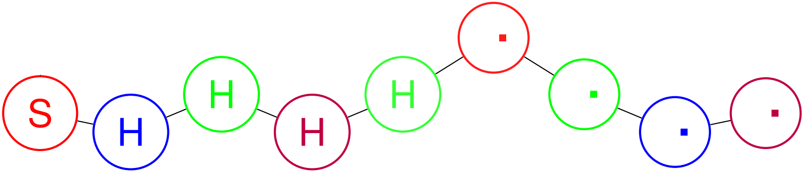
CS 170: Algorithms



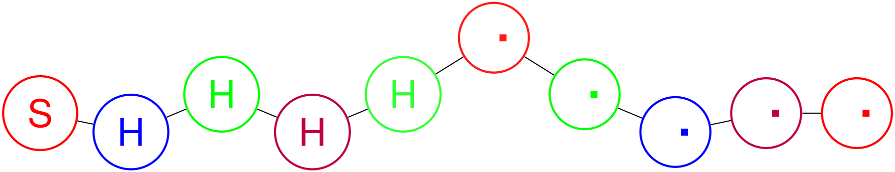
CS 170: Algorithms



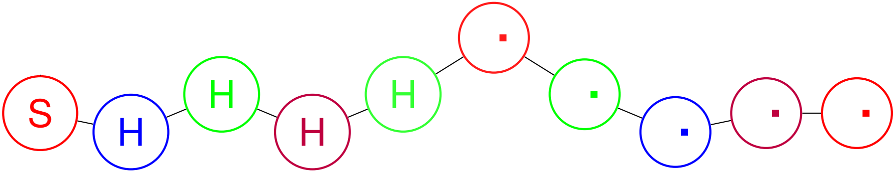
CS 170: Algorithms



CS 170: Algorithms

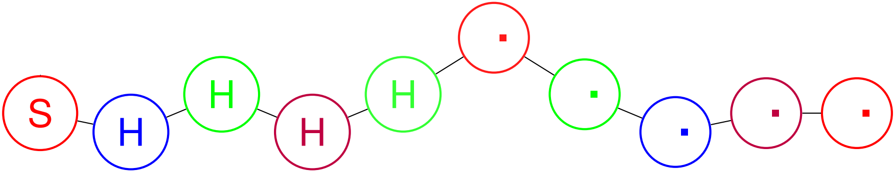


CS 170: Algorithms



No laptops please.

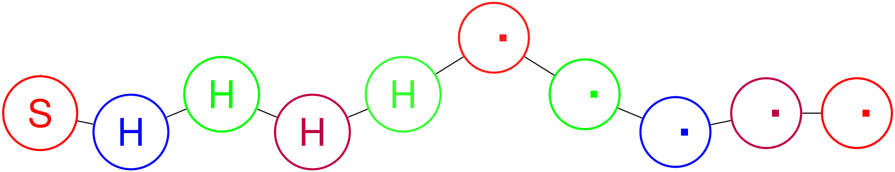
CS 170: Algorithms



No laptops please.

Thank you

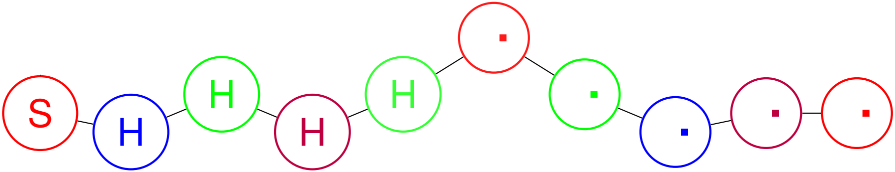
CS 170: Algorithms



No laptops please.

Thank you !

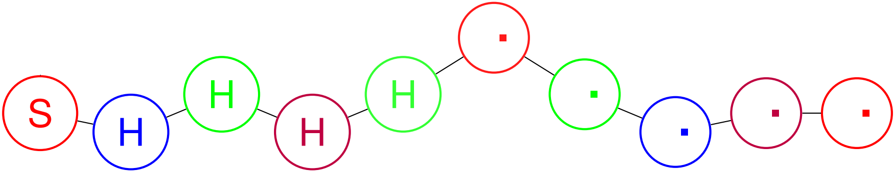
CS 170: Algorithms



No laptops please.

Thank you ! !

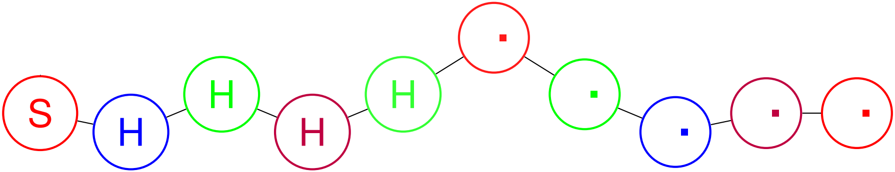
CS 170: Algorithms



No laptops please.

Thank you ! ! !

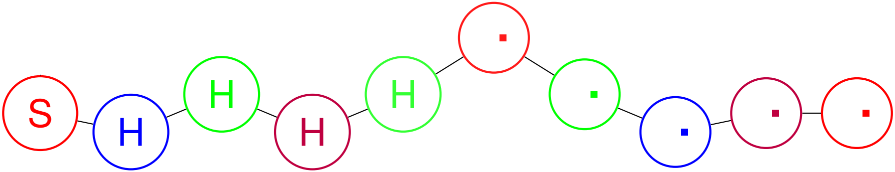
CS 170: Algorithms



No laptops please.

Thank you ! ! ! !

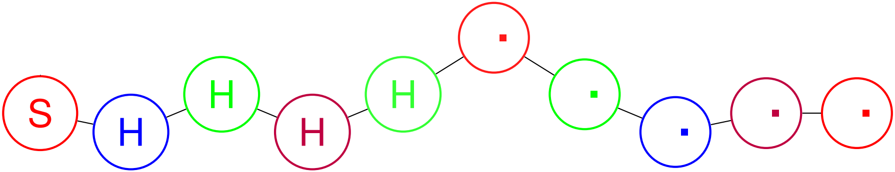
CS 170: Algorithms



No laptops please.

Thank you ! ! ! ! !

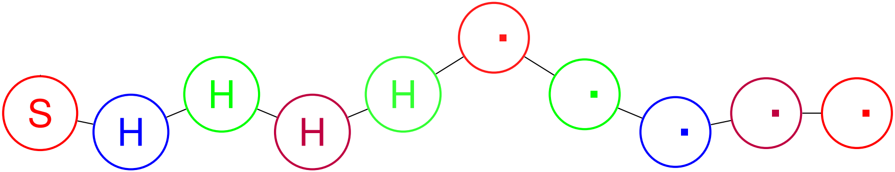
CS 170: Algorithms



No laptops please.

Thank you ! ! ! ! ! !

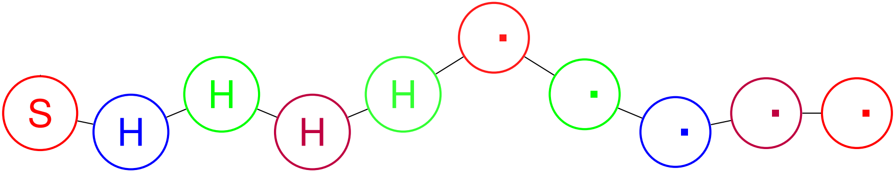
CS 170: Algorithms



No laptops please.

Thank you ! ! ! ! ! ! !

CS 170: Algorithms



No laptops please.

Thank you ! ! ! ! ! ! ! !

Midterm.

Thursday.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.

- True/false.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.
 True/false.
 Short/and pretty short answers.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.

- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.

- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Why?

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.

- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Why? Coverage.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.

- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Why? Coverage. Fairness.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.
   True/false.
   Short/and pretty short answers.
   Two/Three longer questions.

Why? Coverage. Fairness. Smoothness.

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.
- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Why? Coverage. Fairness. Smoothness.

One cheat sheet:

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.
- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Why? Coverage. Fairness. Smoothness.

One cheat sheet: one side!

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.
- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Why? Coverage. Fairness. Smoothness.

One cheat sheet: one side!
- 8.5×11, No mobius strips,

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.
- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Why? Coverage. Fairness. Smoothness.

One cheat sheet: one side!
- 8.5×11, No mobius strips,

Midterm.

Thursday.

Rooms: will give algorithm shortly.

Midterm style: see midterm 1 last fall.
- True/false.
- Short/and pretty short answers.
- Two/Three longer questions.

Why? Coverage. Fairness. Smoothness.

One cheat sheet: one side!
- 8.5×11, No mobius strips,

Must be questions.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

Negative edges.

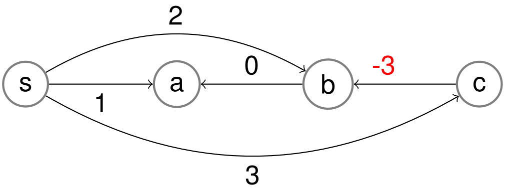
Notice: argument for Dijkstra breaks for negative edges.

For example.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

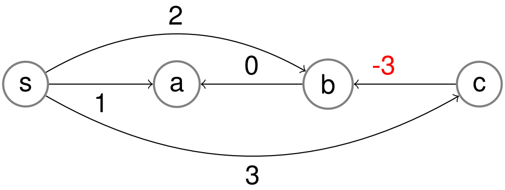


Dijkstra:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

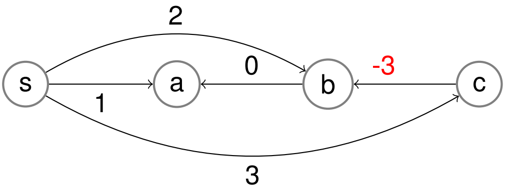


Dijkstra:
Process s:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

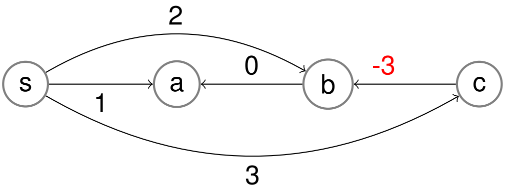


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

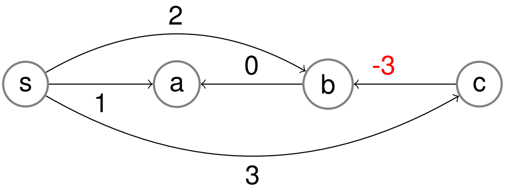


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

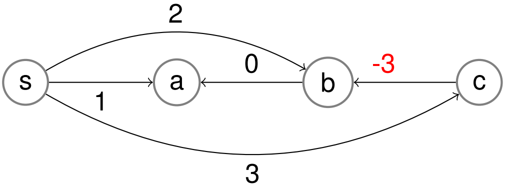


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

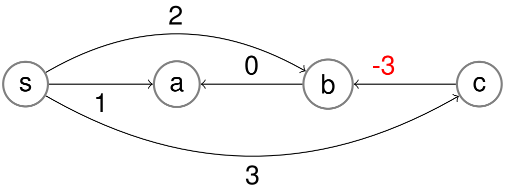


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

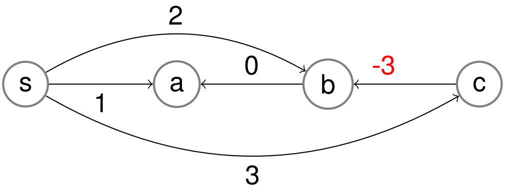


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

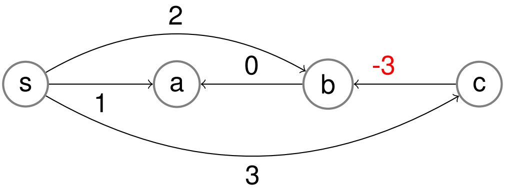


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$:

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

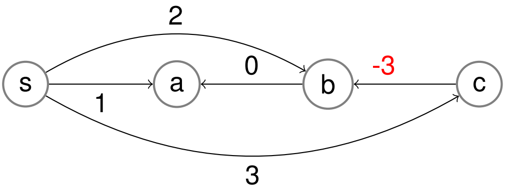


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

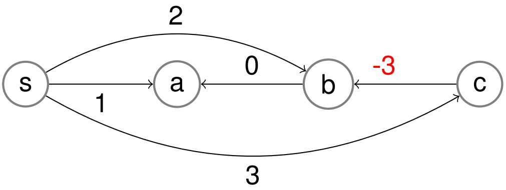


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

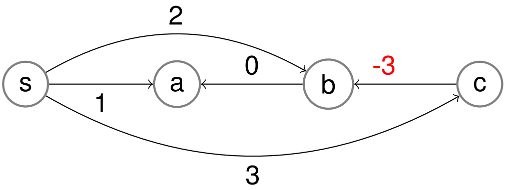


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!
$d(a)$ still 1.

Negative edges.

Notice: argument for Dijkstra breaks for negative edges.

For example.

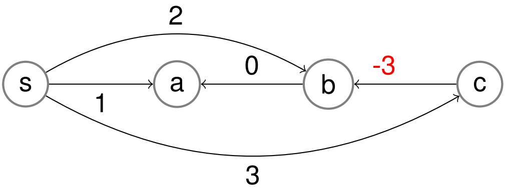


Dijkstra:
Process s: Set $d(a) = 1, d(b) = 2, d(c) = 3$.
Process a, $d(a) = 1$: No outgoing edges.
Process b, $d(b) = 2$: $d(a)$ still set to 1.
Process c, $d(c) = 3$: Set $d(b) = 0$.
But, can't process b, again!!!
$d(a)$ still 1. Should be 0

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..

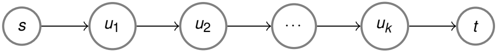
$$s \rightarrow u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \rightarrow t$$

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),

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..

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

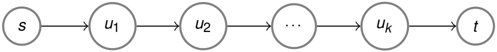
If update (s, u_1), then (u_1, u_2),

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..



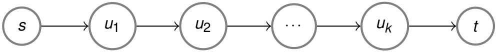
If update (s, u_1), then (u_1, u_2), ...

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..



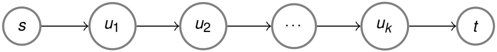
If update (s, u_1), then (u_1, u_2), ... and finally (u_k, t).

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), $\ldots$ 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..

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

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

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..

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

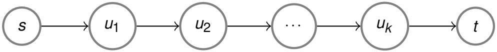
If update (s, u_1), then (u_1, u_2), ... and finally (u_k, t). It's all good!

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..



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..

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

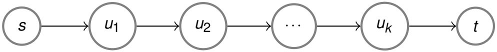
If update (s, u_1), then (u_1, u_2), $\ldots$ and finally (u_k, t). It's all good!
How??? Update!

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..



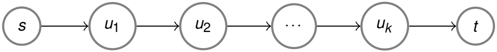
If update (s, u_1), then (u_1, u_2), $\ldots$ and finally (u_k, t). It's all good!
How??? Update! Here,

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..



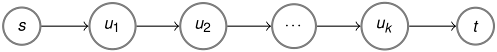
If update (s, u_1), then (u_1, u_2), ... and finally (u_k, t). It's all good!
How??? Update! Here, there,

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..

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

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

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

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/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.

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.

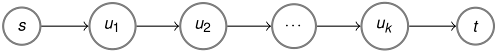
Correctness: After ith loop,

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..



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, if there is a negative cycle!

Negative cycle detection.

After n iterations, some distance changes, if there is a negative cycle.!

Negative cycle detection.

After n iterations, some distance changes, if there is a negative cycle.!

Consider a negative cycle, C:

Negative cycle detection.

After n iterations, some distance changes, if there is a negative cycle.!

Consider a negative cycle, C:

$\sum_{e=(u,v)\in C} l(u,v)$

Negative cycle detection.

After n iterations, some distance changes, if there is a negative cycle.!

Consider a negative cycle, C:

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

Negative cycle detection.

After n iterations, some distance changes, if there is a negative cycle.!

Consider a negative cycle, C:

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

$\rightarrow$ there is e,

Negative cycle detection.

After n iterations, some distance changes, if there is a negative cycle.!

Consider a negative cycle, C:

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

$\rightarrow$ there is e, where $d(u) + l(u,v) - d(v) < 0$

Negative cycle detection.

After n iterations, some distance changes, if there is a negative cycle.!

Consider a negative cycle, C:

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

$\rightarrow$ there is e, where $d(u) + l(u,v) - d(v) < 0$

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

Negative cycle detection.

After n iterations, some distance changes, if there is a negative cycle.!

Consider a negative cycle, C:

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

$\rightarrow$ there is e, where $d(u) + l(u,v) - d(v) < 0$

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

Update will change a distance!

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?

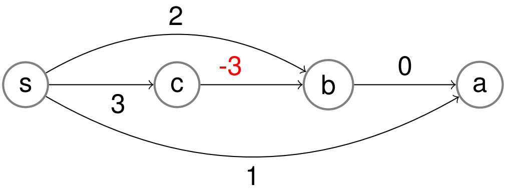
Not possible. No cycles at all!

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

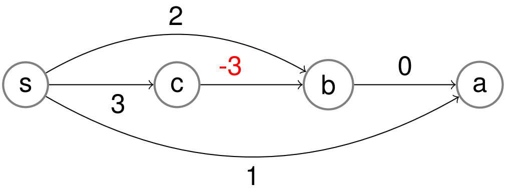
Remember ...Goliad!
Remember ...updating along path makes it all good.

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

DAG

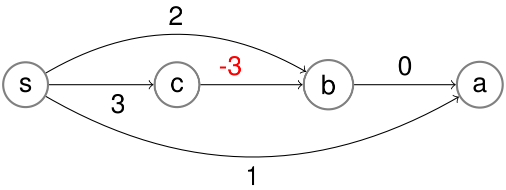


DAG



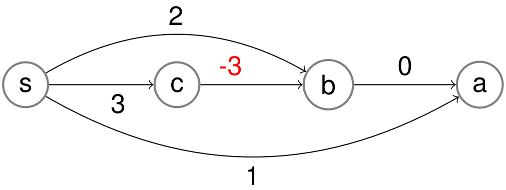
Process s, $d(s) = 0$:

DAG



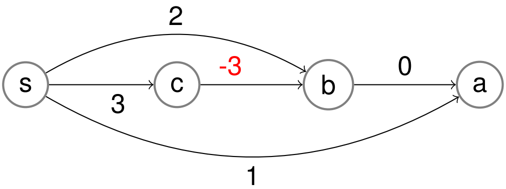
Process s, $d(s) = 0$: Updates $d(c) = 3$,

DAG



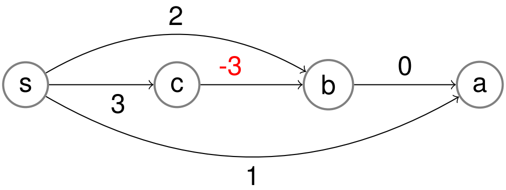
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2$,

DAG



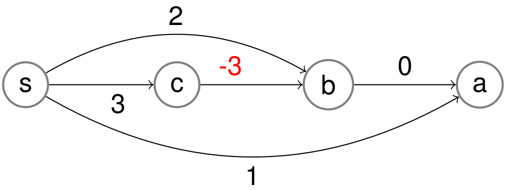
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.

DAG



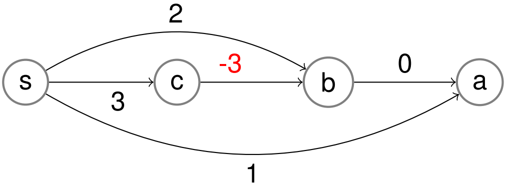
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$:

DAG



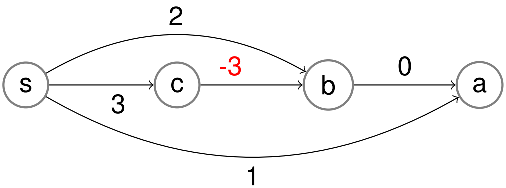
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.

DAG



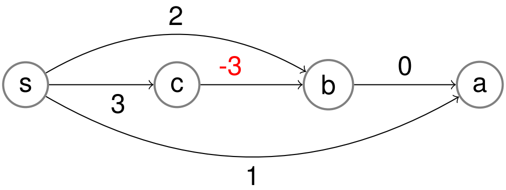
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.
Process b, $d(b) = 0$:

DAG



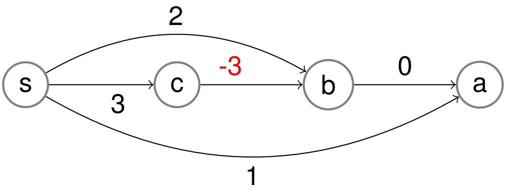
Process s, $d(s) = 0$: Updates $d(c) = 3, d(b) = 2, d(a) = 1$.
Process c, $d(c) = 3$: Update $d(b) = 0$.
Process b, $d(b) = 0$: $d(a) = 0$.

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.

Trees.

Def: A tree is a connected graph with no cycles.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n - 1$ edges.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions,

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!)

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not,

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.
Doh!

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths: cycle!

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths: cycle! Not Tree!

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths: cycle! Not Tree!

If yes, connected and no cycle.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.
Adding some edge reduces components by one.
After $n-1$ additions one component!
(If more additions, cycle!) $\square$

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ connected graph with a cycle.
Remove edge on cycle, still connected.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths: cycle! Not Tree!

If yes, connected and no cycle. Tree!

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes?

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes? No?

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes? No?

Yes.

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes? No?

Yes. If edge weights positive.

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes? No?

Yes. If edge weights positive.

If negative edges, then restrict to tree.

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

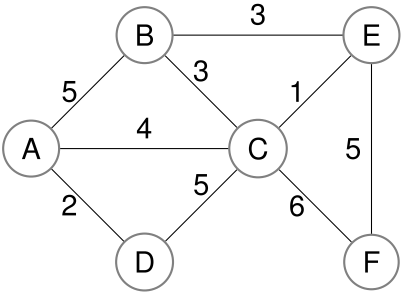
Yes? No?

Yes. If edge weights positive.

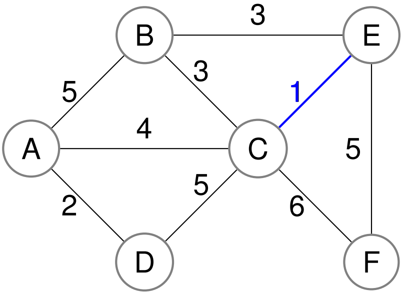
If negative edges, then restrict to tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the lowest weight spanning tree.

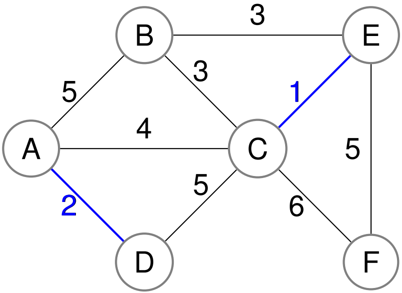
Example and Algorithm



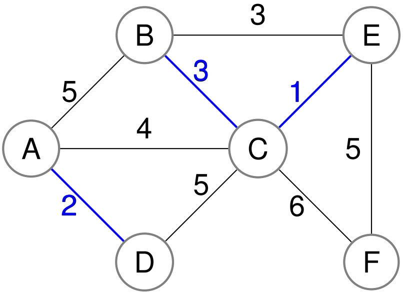
Example and Algorithm



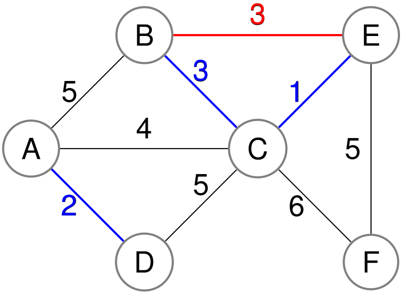
Example and Algorithm



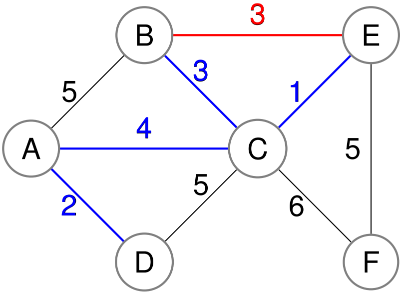
Example and Algorithm



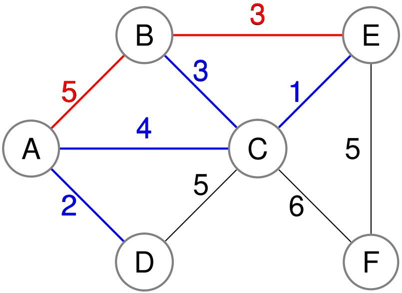
Example and Algorithm



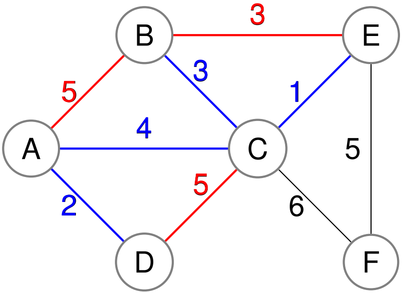
Example and Algorithm



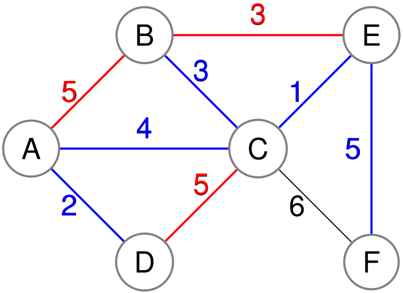
Example and Algorithm



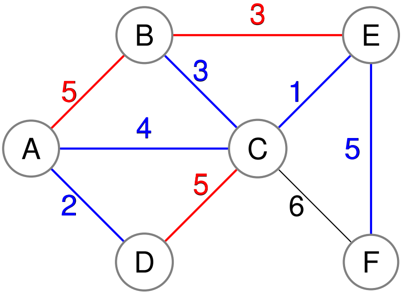
Example and Algorithm



Example and Algorithm



Example and Algorithm



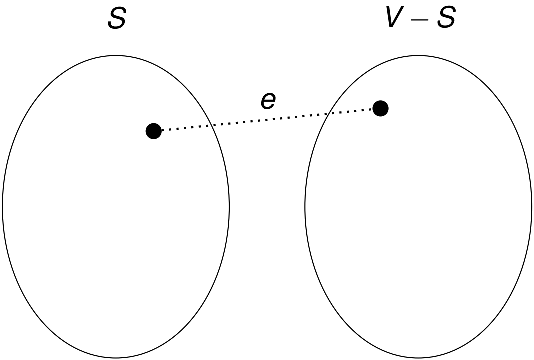
MST: total cost is 2+4+3 +1 + 5 = 15.

Cut property.

Smallest edge across any cut is in some MST.

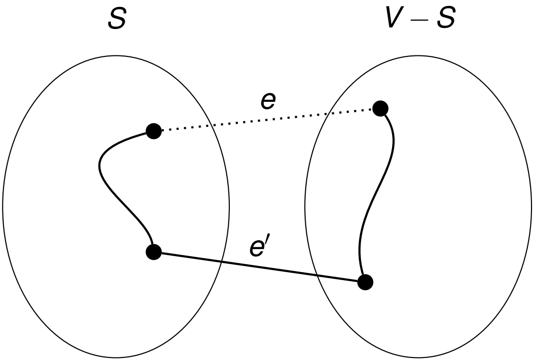
Cut property.

Smallest edge across any cut is in some MST.



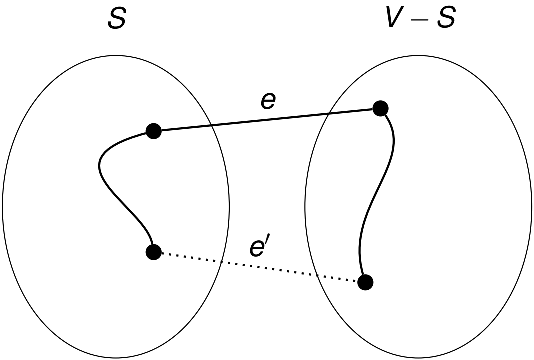
Cut property.

Smallest edge across any cut is in some MST.



Cut property.

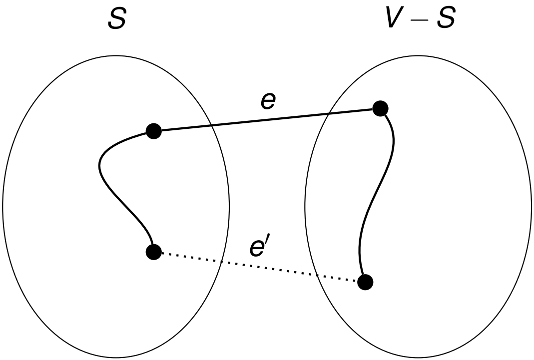
Smallest edge across any cut is in some MST.



Replace e' with e.

Cut property.

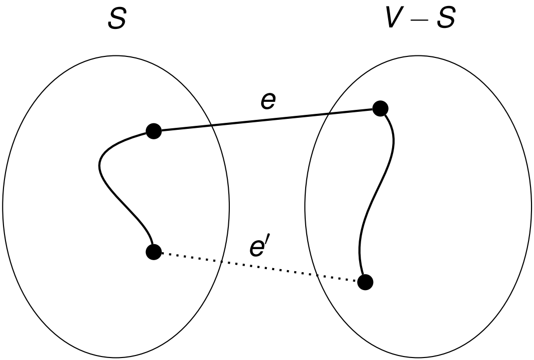
Smallest edge across any cut is in some MST.



Replace e' with e.
Every pair remains connected.

Cut property.

Smallest edge across any cut is in some MST.



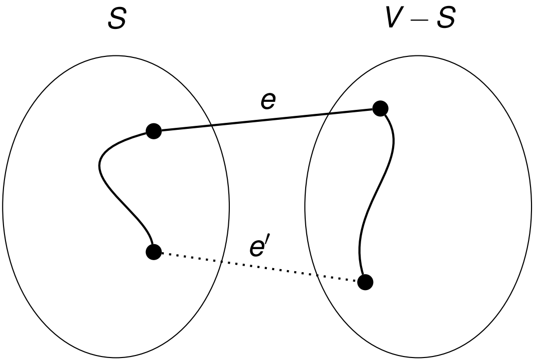
Replace e' with e.

Every pair remains connected.

If used e' can use path through e.

Cut property.

Smallest edge across any cut is in some MST.



Replace e' with e.

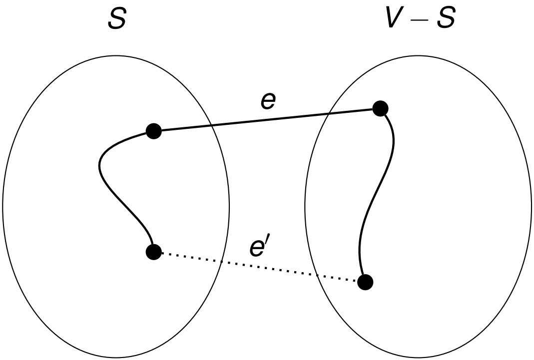
Every pair remains connected.

If used e' can use path through e.

and $n-1$ edges.

Cut property.

Smallest edge across any cut is in some MST.



Replace e' with e.

Every pair remains connected.

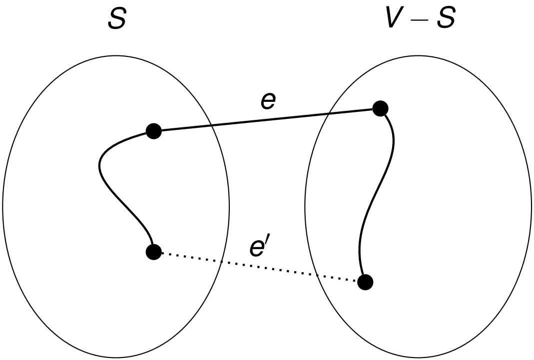
  If used e' can use path through e.

and $n-1$ edges.

So still a tree and is no more costly ($w(e) \leq w(e')$.)

Cut property.

Smallest edge across any cut is in some MST.



Replace e' with e.

Every pair remains connected.

If used e' can use path through e.

and $n-1$ edges.

So still a tree and is no more costly ($w(e) \leq w(e')$.)

□

Kruskal

Sort edges.
For each edge.
    If no cycle, add edge.

Kruskal

Sort edges.
For each edge.
    If no cycle, add edge.

How to check for cycle for edge (u, v)?

Kruskal

Sort edges.
For each edge.
    If no cycle, add edge.

How to check for cycle for edge (u, v)?

Check for path between u and v in set of added edges.

Kruskal

Sort edges.
For each edge.
    If no cycle, add edge.

How to check for cycle for edge (u, v)?

Check for path between u and v in set of added edges.

Total Running time?

Kruskal

Sort edges.
For each edge.
  If no cycle, add edge.

How to check for cycle for edge (u, v)?

Check for path between u and v in set of added edges.

Total Running time?

$O(n)$ time

Kruskal

Sort edges.
For each edge.
    If no cycle, add edge.

How to check for cycle for edge (u, v)?

Check for path between u and v in set of added edges.

Total Running time?

$O(n)$ time $\rightarrow$ $O(nm)$ for Kruskals.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Kruskal

Sort edges.
For each edge (u, v):
  If no cycle, add edge.

Main issue: Check for cycle.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle?

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

"If no cycle"

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

"If no cycle" $\equiv$ "**find**(u) $\neq$ **find**(v)"

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

"If no cycle" $\equiv$ "**find**(u) $\neq$ **find**(v)"

"Add edge"

Kruskal

Sort edges.
For each edge (u, v):
    If no cycle, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

"If no cycle" $\equiv$ "**find**(u) $\neq$ **find**(v)"

"Add edge" $\equiv$ "**union(u,v)**"

See you

See you ..on Monday.