DIJKSTRA'S ALGORITHM

DIJKSTRA'S ALGORITHM

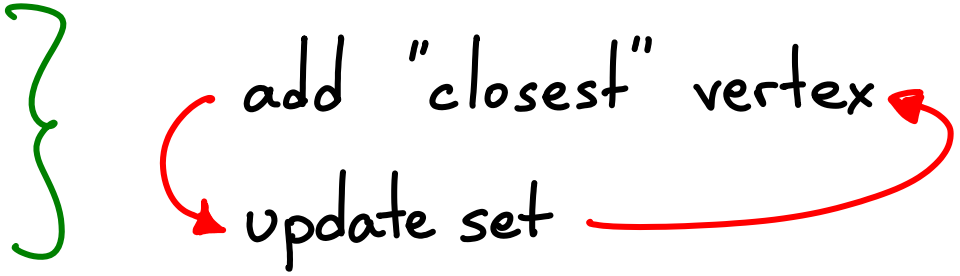
Grow SSSP tree greedily.

Maintain set of vertices to add.

DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

Maintain set of vertices to add.



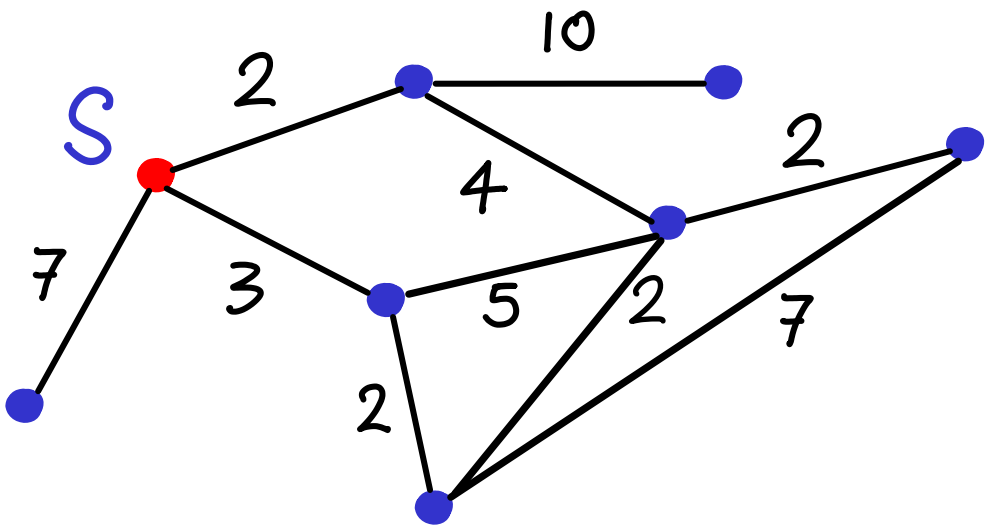
DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

Maintain set of vertices to add.

add "closest" vertex

update set



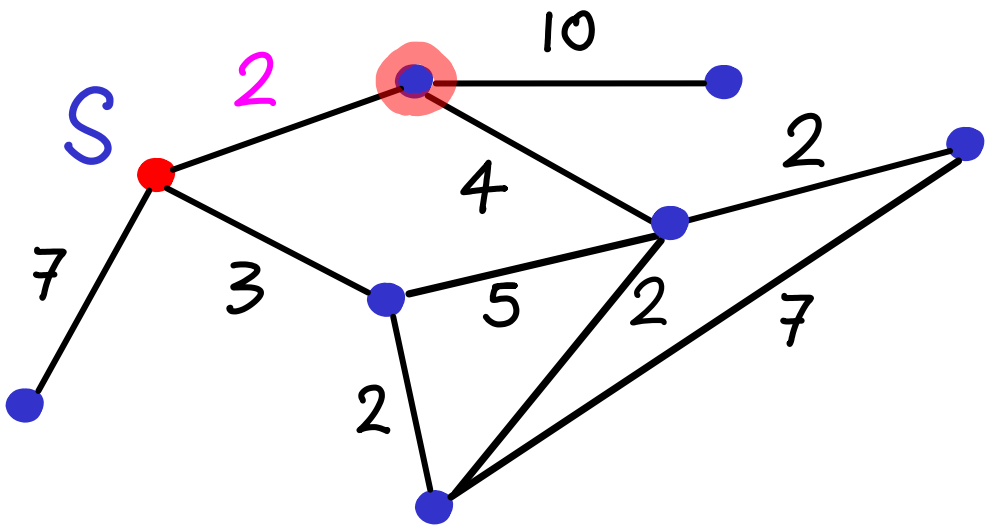
DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

Maintain set of vertices to add.

add "closest" vertex

update set



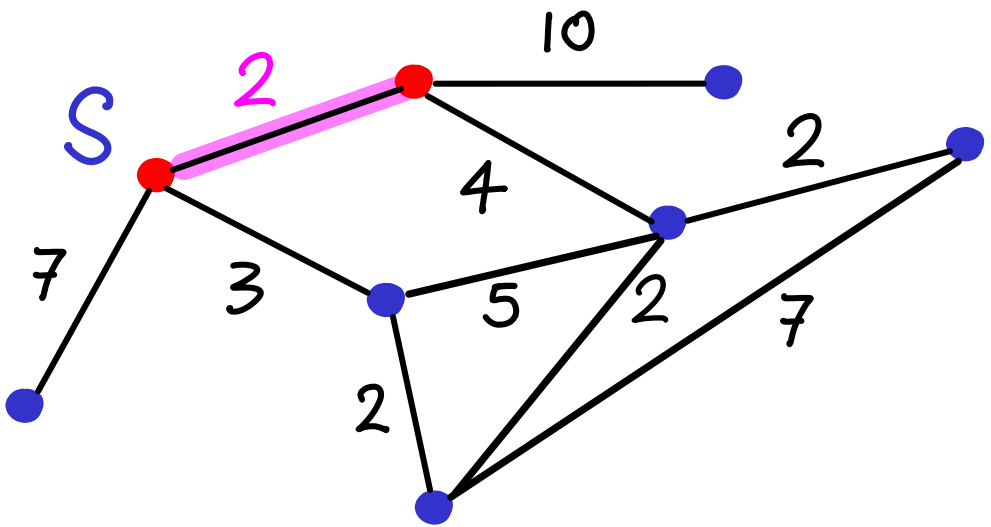
DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

Maintain set of vertices to add.

add "closest" vertex

update set



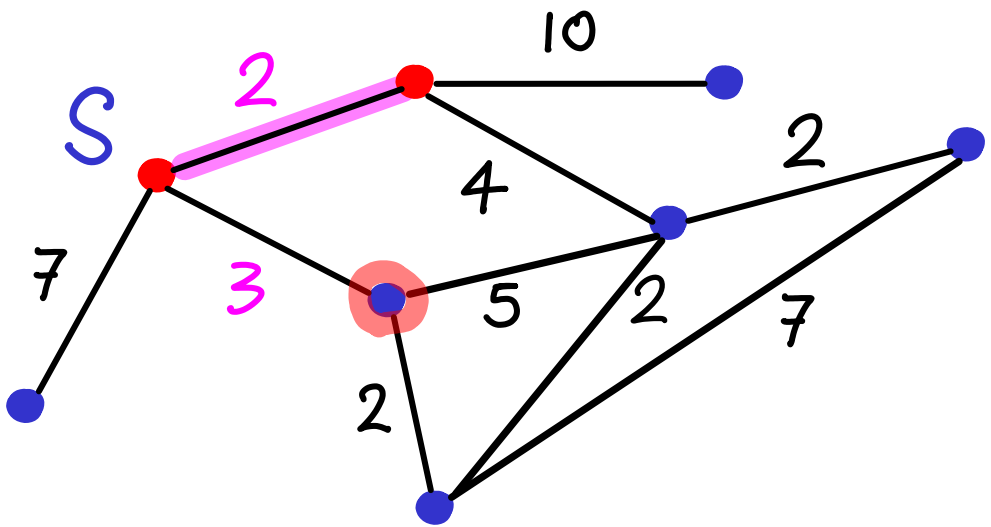
DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

Maintain set of vertices to add.

add "closest" vertex

update set



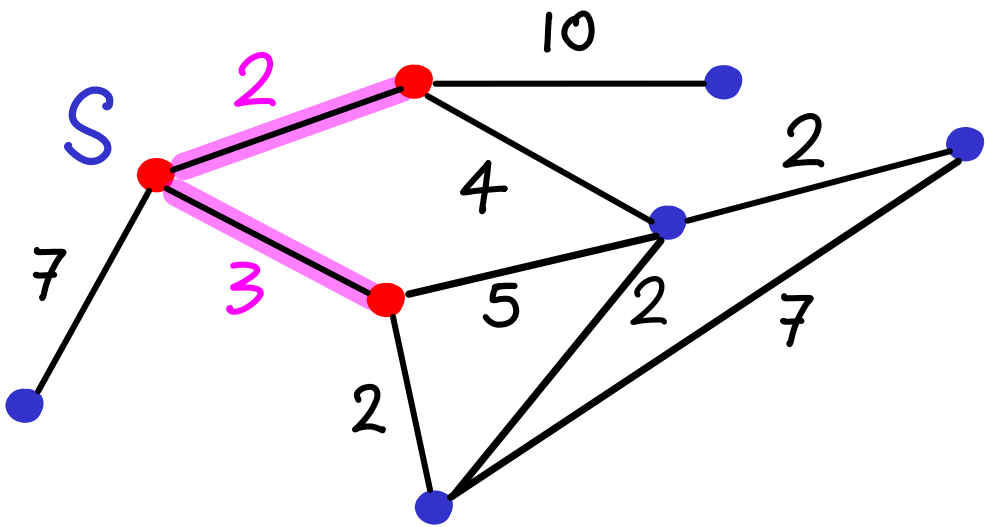
DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

Maintain set of vertices to add.

add "closest" vertex

update set



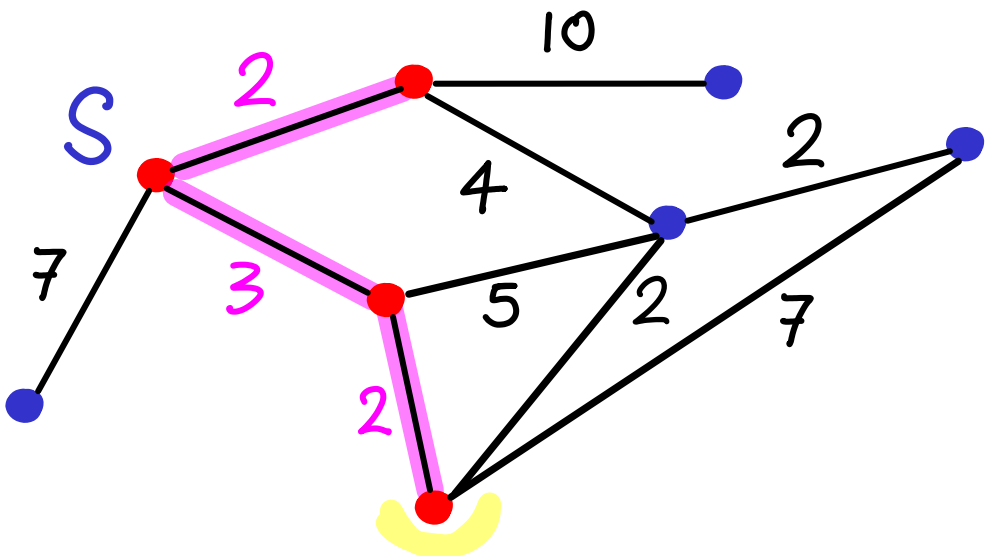
DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

Maintain set of vertices to add.

add "closest" vertex

update set



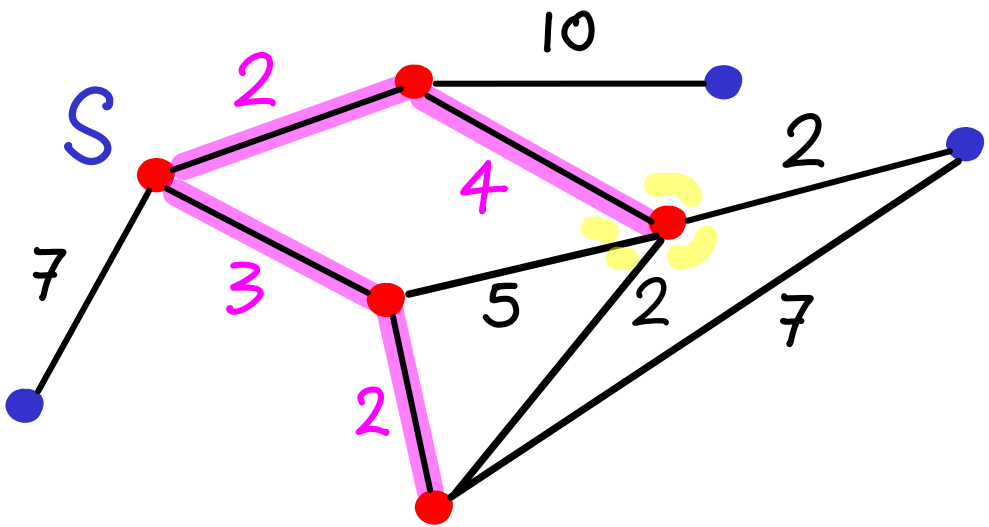
DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

Maintain set of vertices to add.

add "closest" vertex

update set



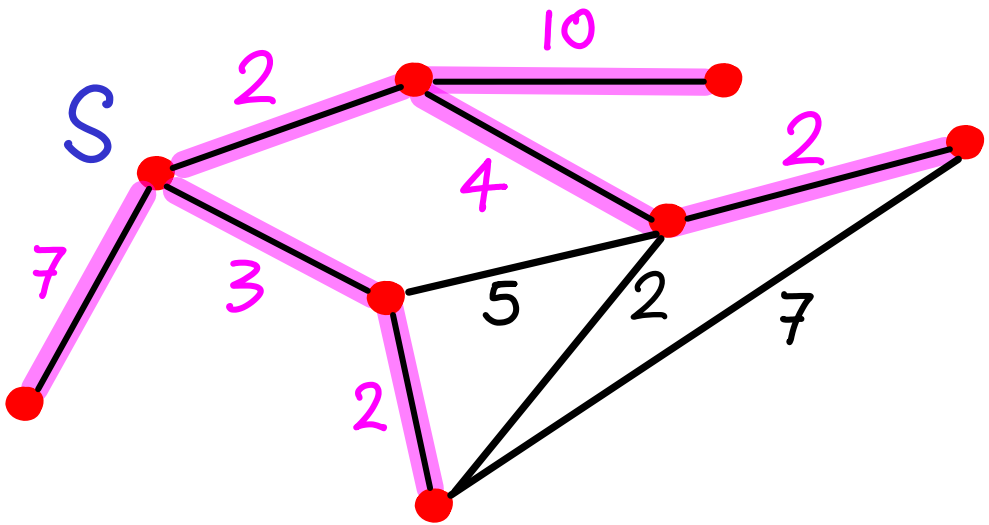
DIJKSTRA'S ALGORITHM

Grow SSSP tree greedily.

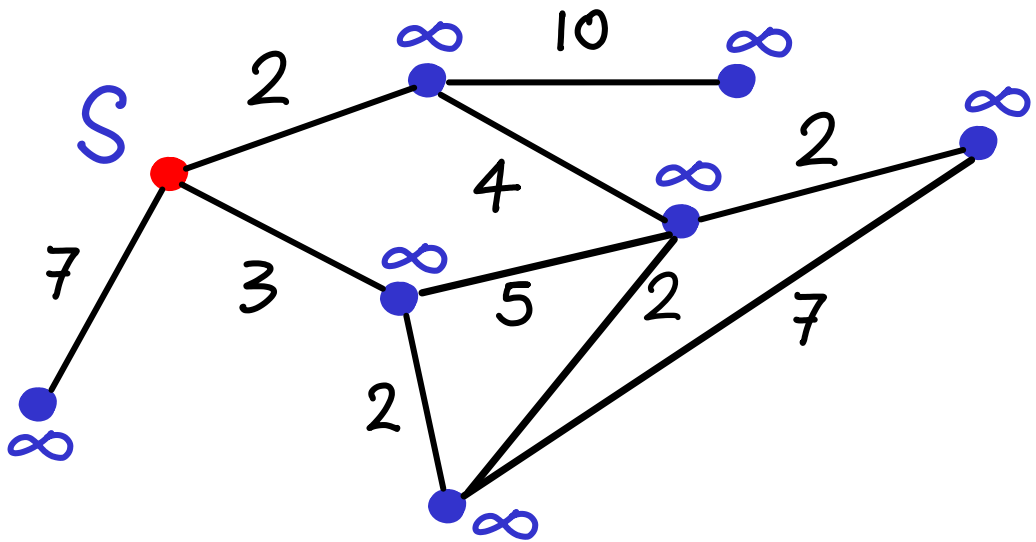
Maintain set of vertices to add.

add "closest" vertex

update set



Initialize : $S=0$, others $=\infty$



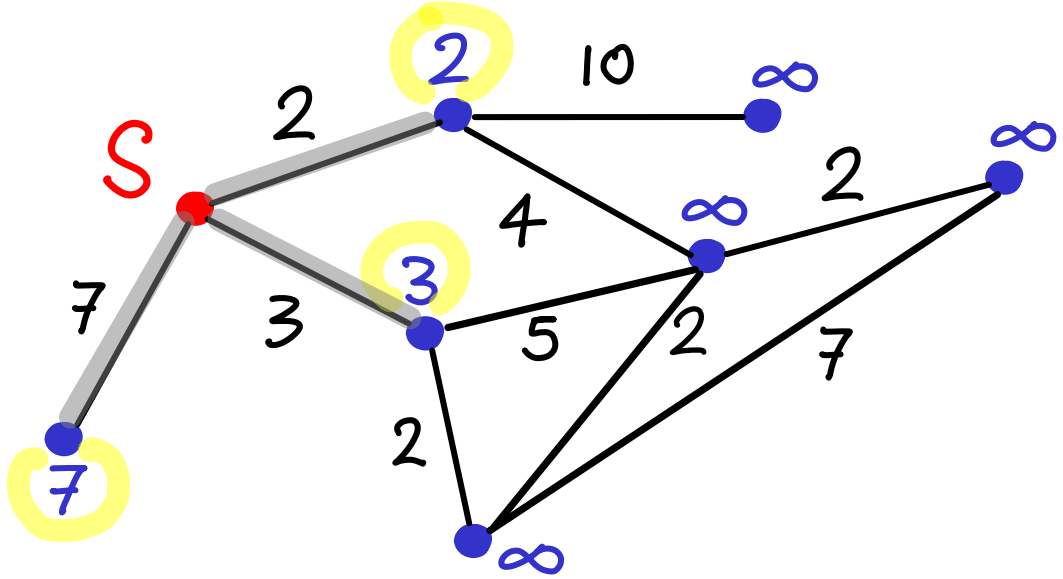
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



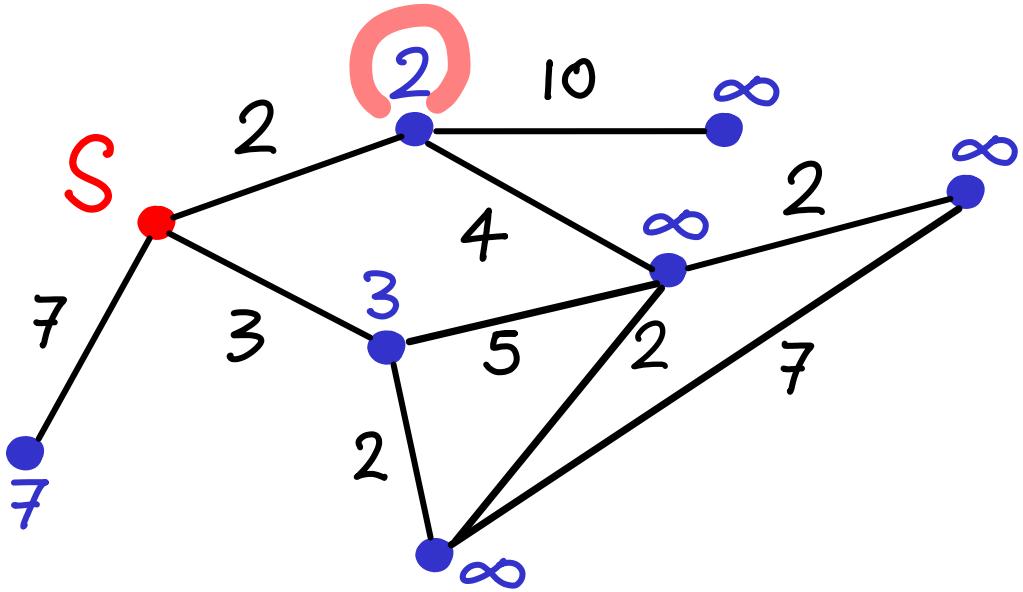
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



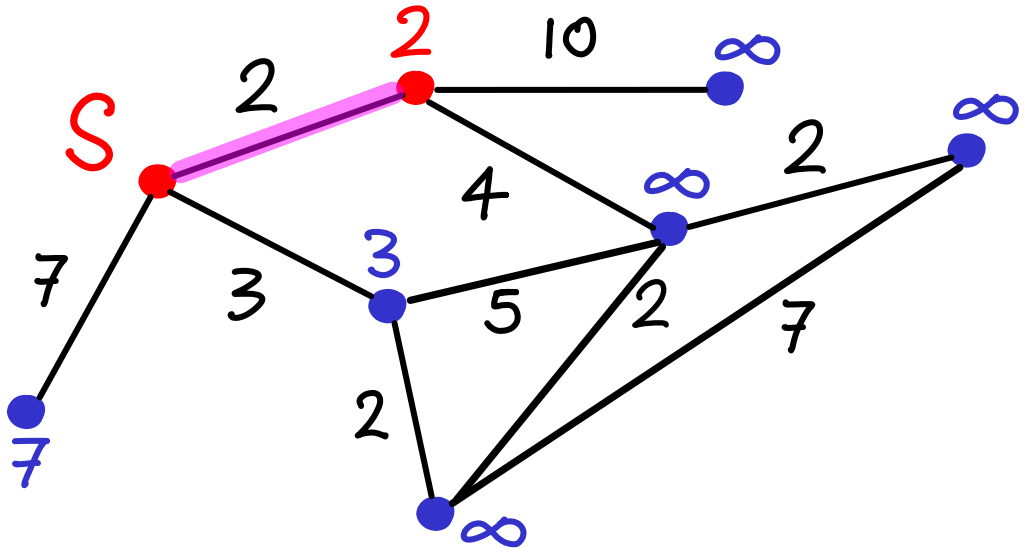
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



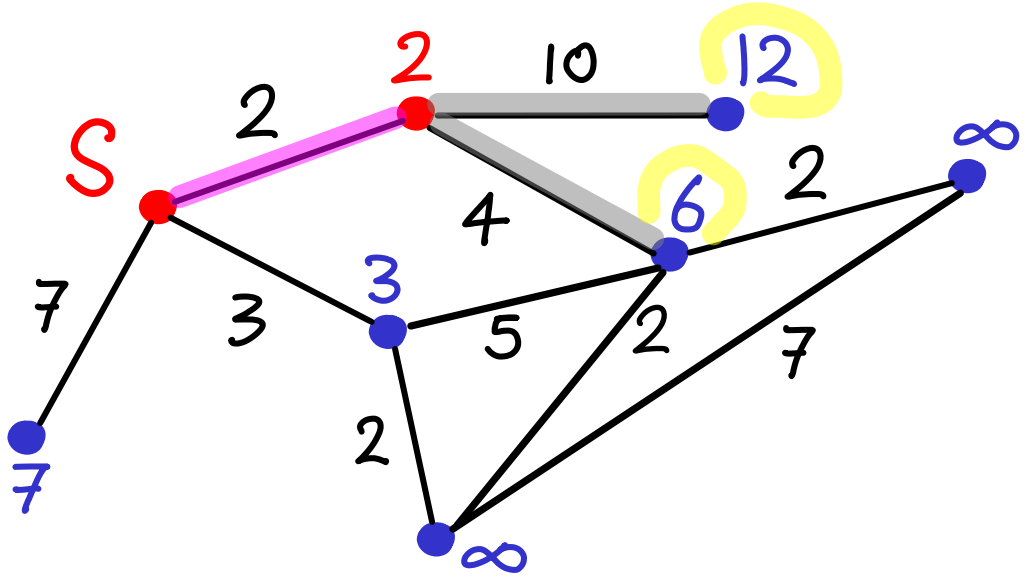
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



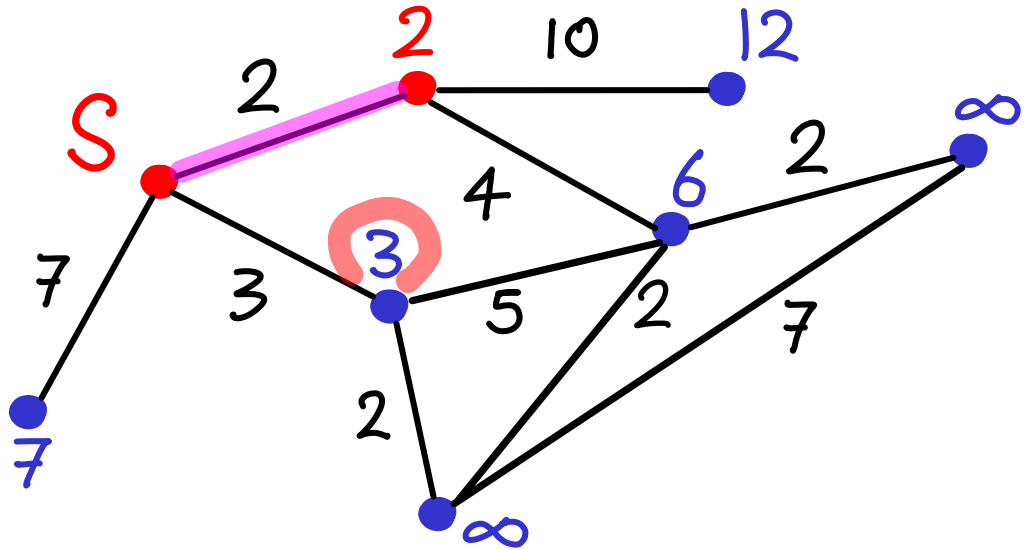
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



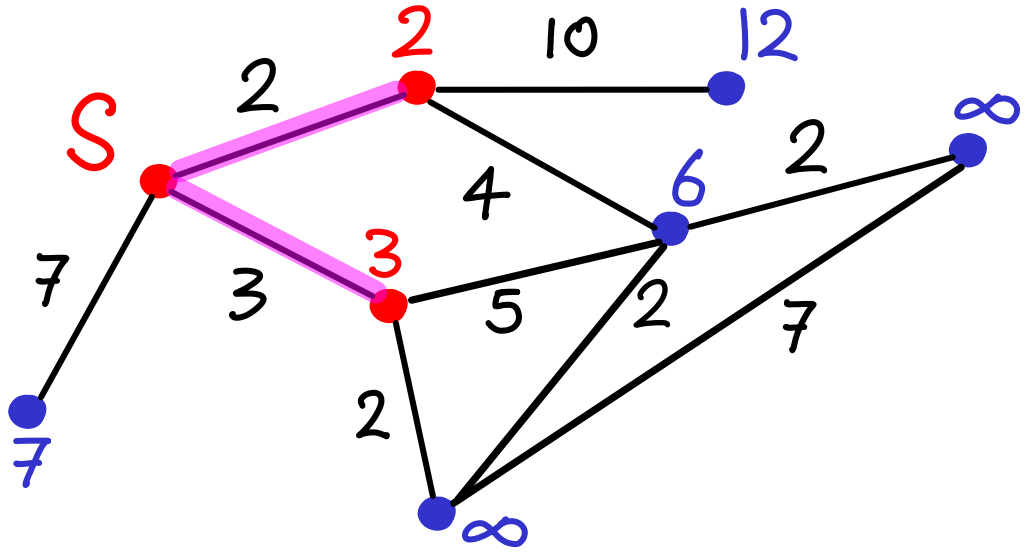
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



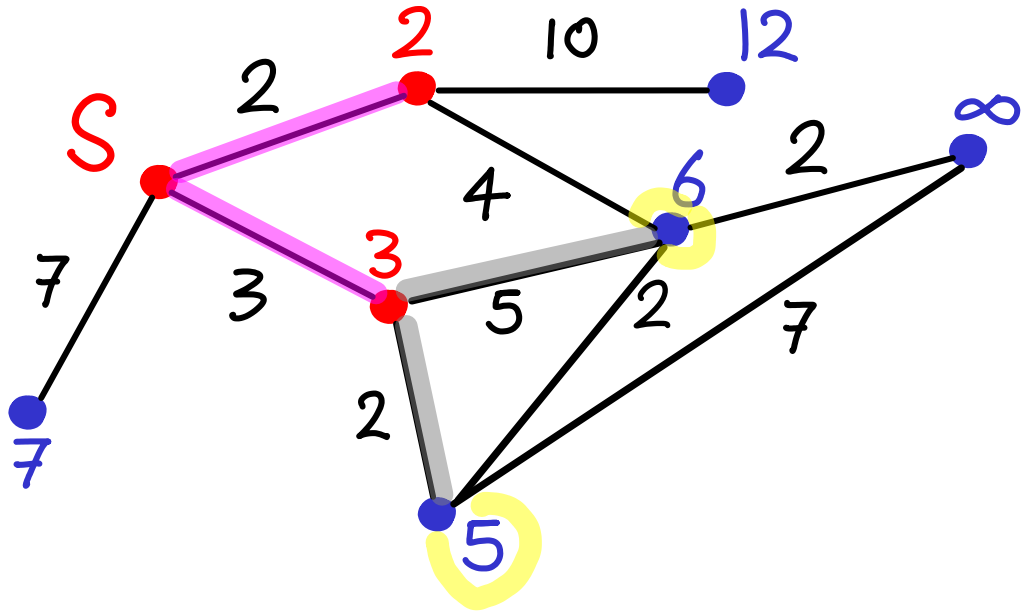
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



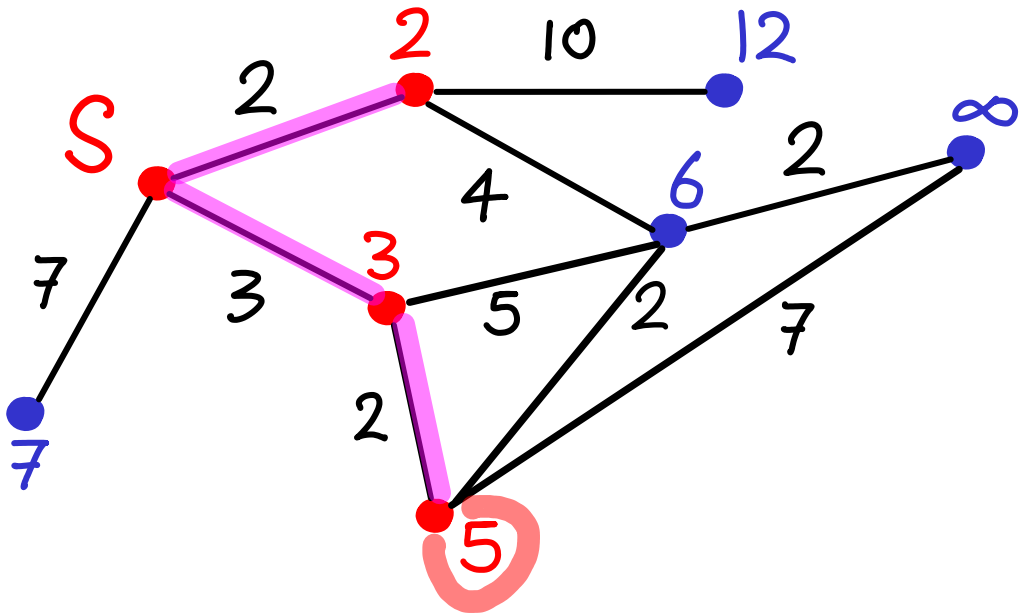
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



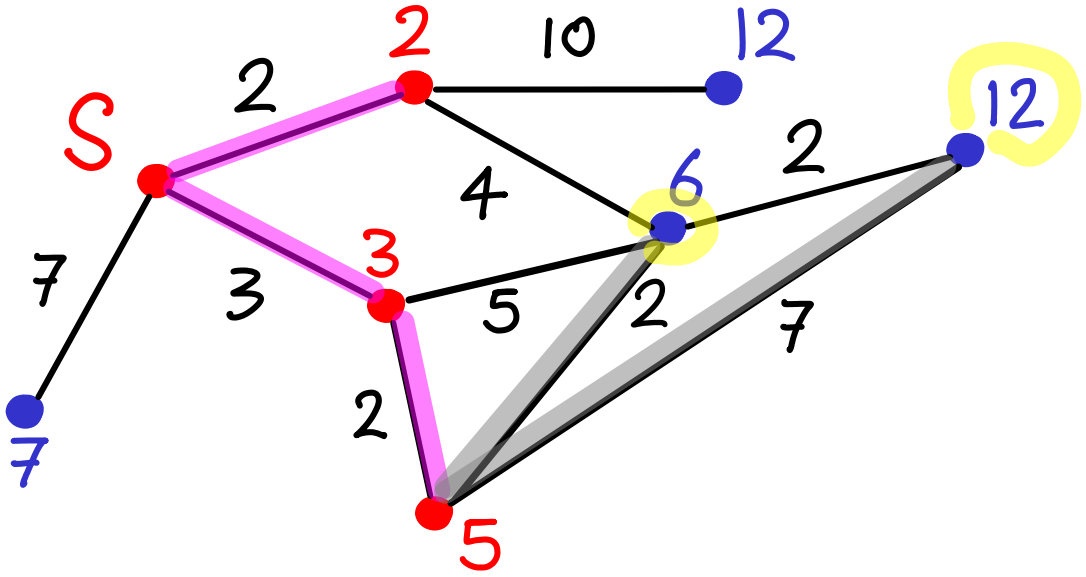
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



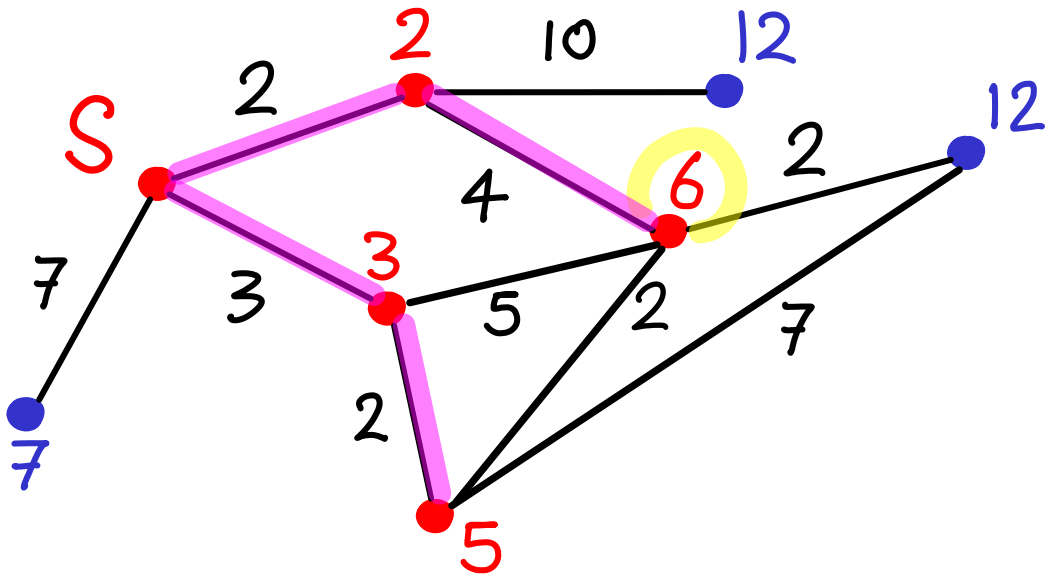
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

= relax incident edges



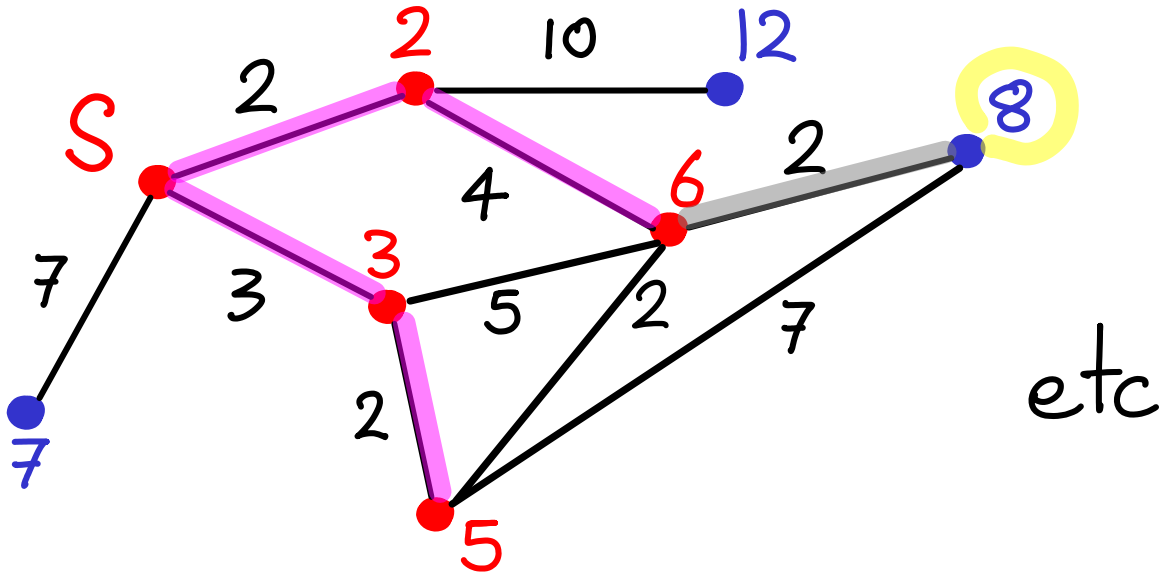
Initialize : $S=0$, others $=\infty$

add "closest" vertex

update set

= extract lowest score

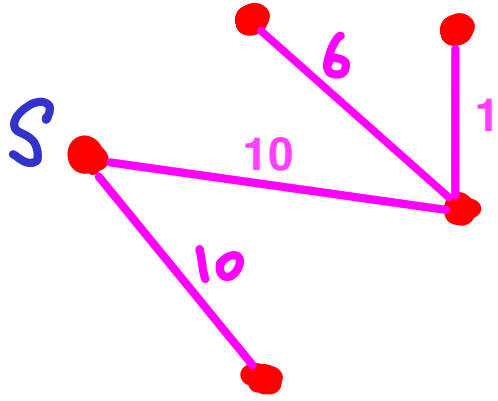
= relax incident edges



etc

Correctness:

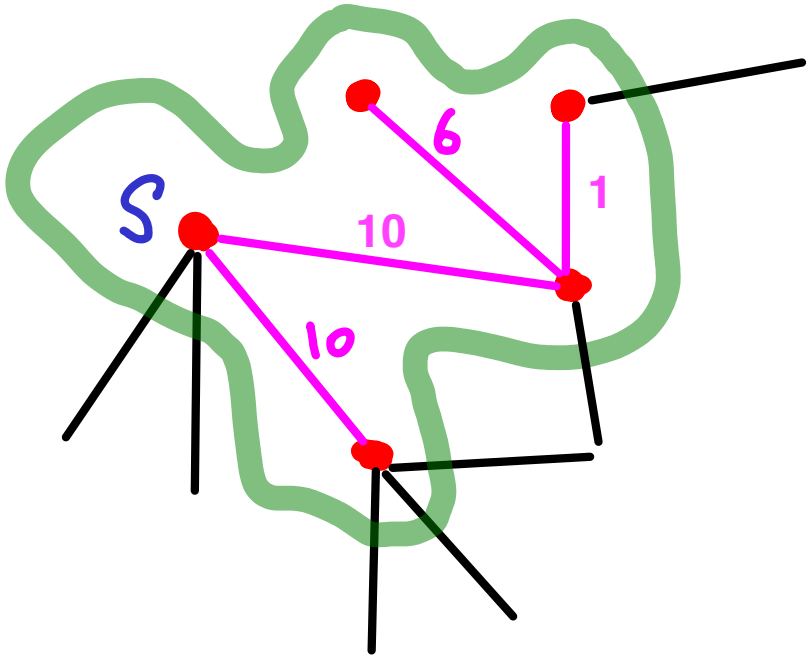
assume we have shortest paths
to a set of red vertices



Correctness:

assume we have shortest paths
to a set of red vertices

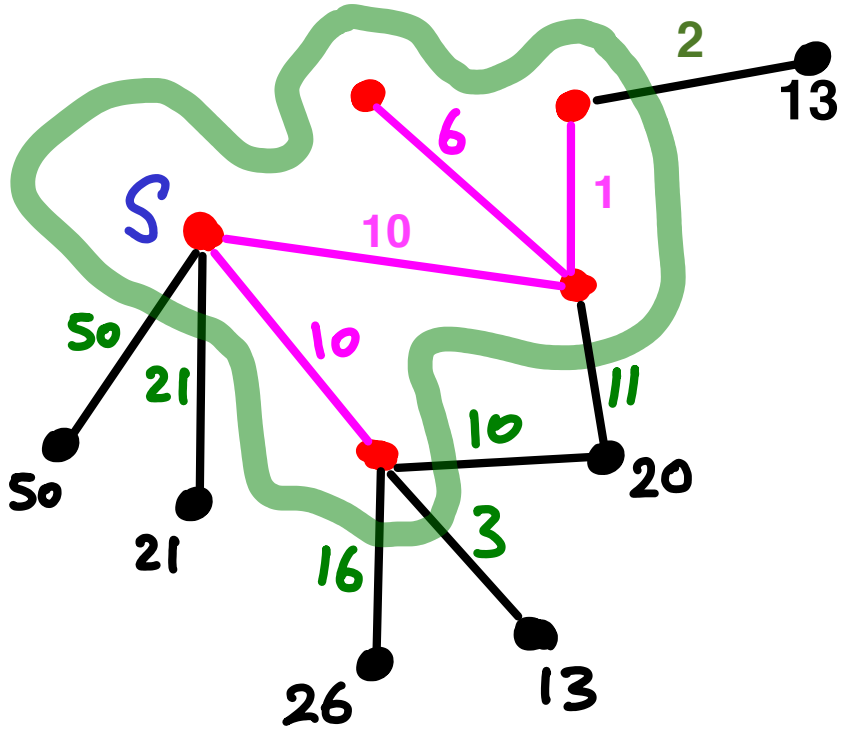
CUT



Correctness:

assume we have shortest paths
to a set of red vertices

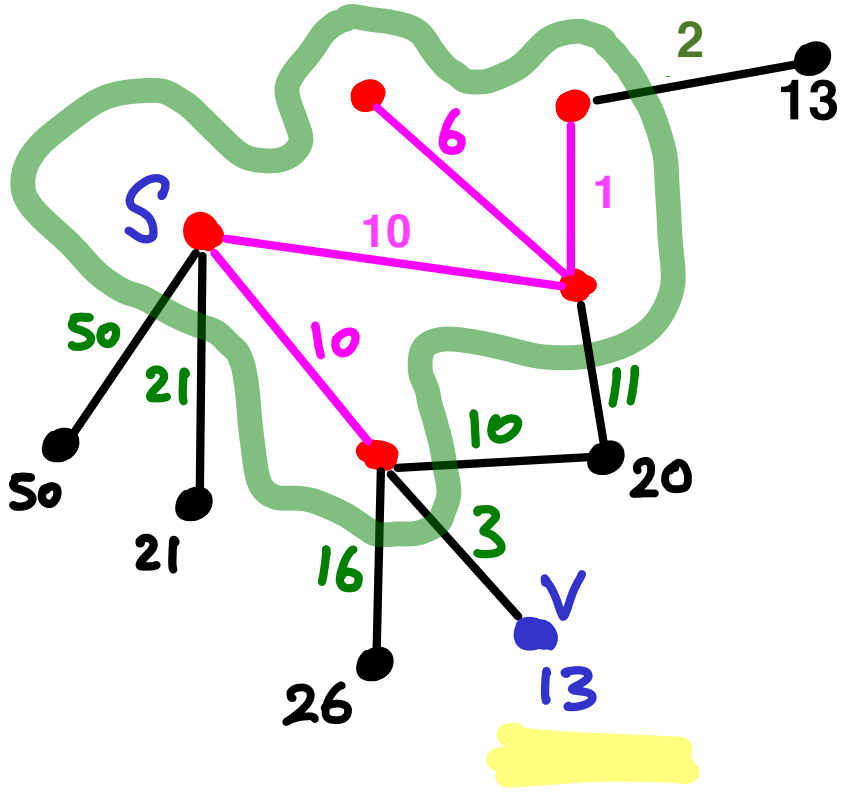
Just across CUT,
vertices have finite scores
= [a path in known set] + black edge



Correctness:

assume we have shortest paths
to a set of red vertices

Just across CUT,
vertices have finite scores
= [a path in known set] + black edge

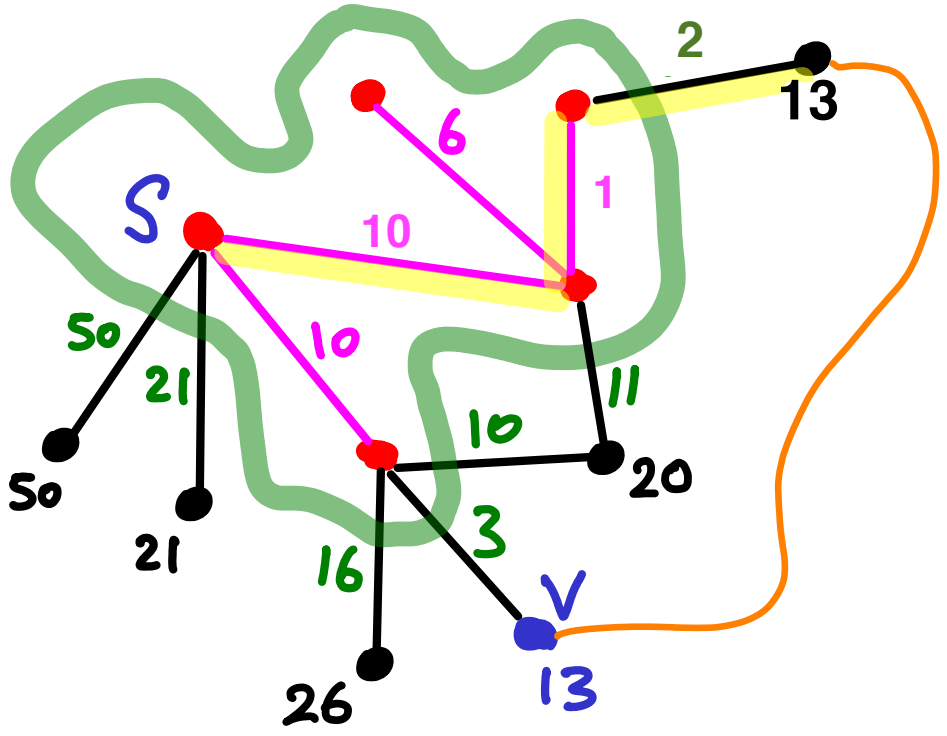


V = "closest" to S

Correctness:

assume we have shortest paths
to a set of red vertices

Just across CUT,
vertices have finite scores
= [a path in known set] + black edge



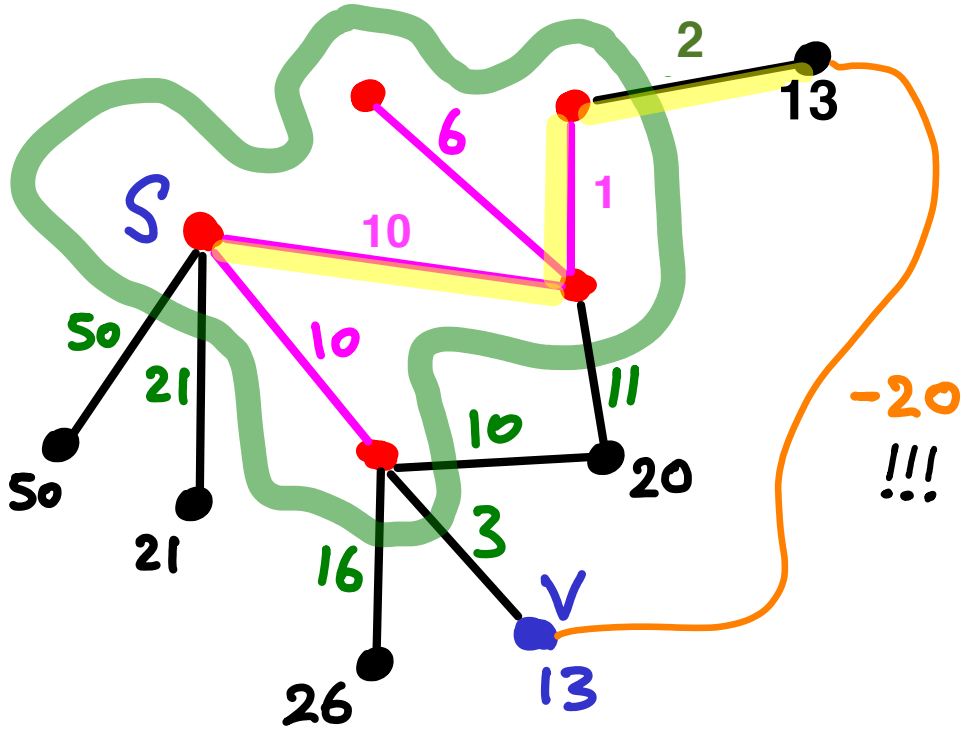
v = "closest" to S

Any other path s ~~~ v
will cost at least as much

Correctness:

assume we have shortest paths
to a set of red vertices

Just across CUT,
vertices have finite scores
= [a path in known set] + black edge



v = "closest" to S

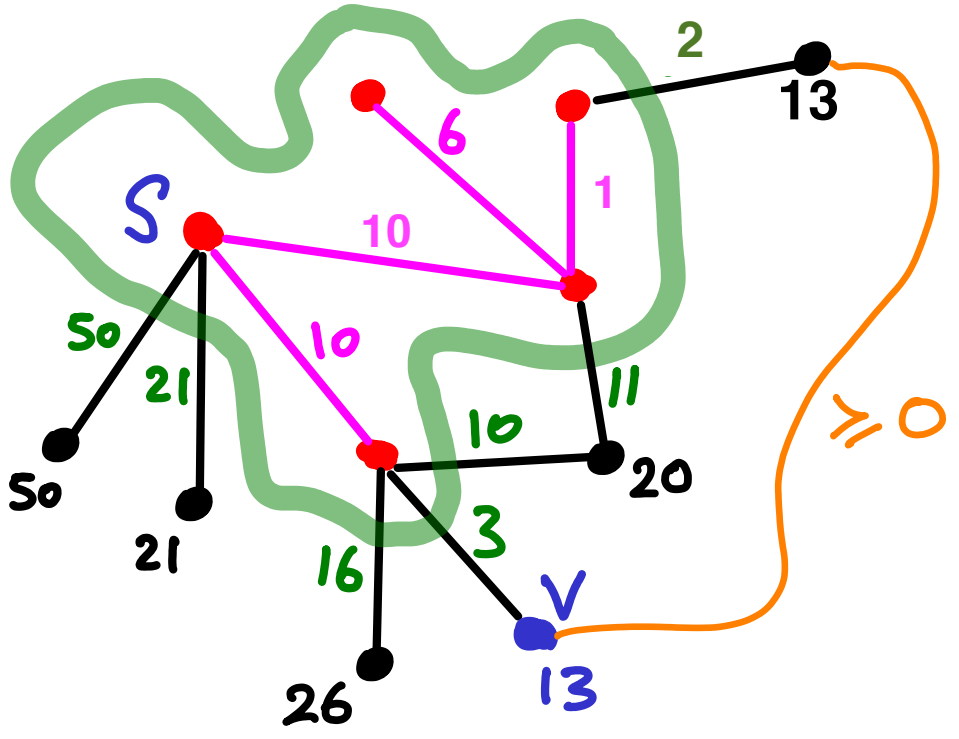
Any other path s ~~~ v
will cost at least as much

} assuming weights ≥ 0

Correctness:

assume we have shortest paths
to a set of red vertices

Just across CUT,
vertices have finite scores
= [a path in known set] + black edge



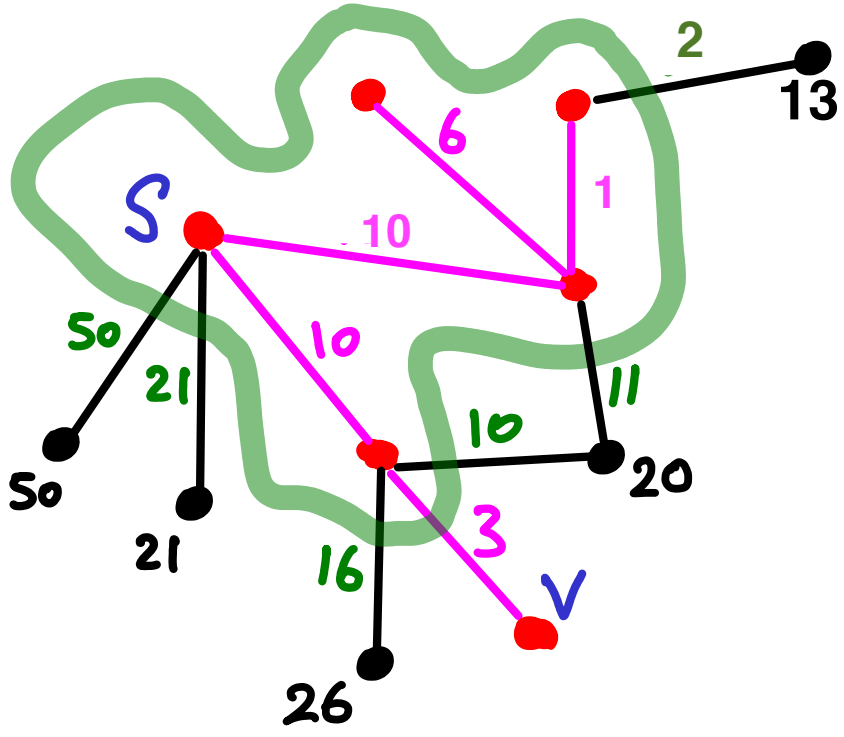
v = "closest" to S → safe to add

Any other path s ~~~ v
will cost at least as much

assuming weights ≥0

Correctness:

assume we have shortest paths
to a set of red vertices



Just across CUT,
vertices have finite scores
= [a path in known set] + black edge

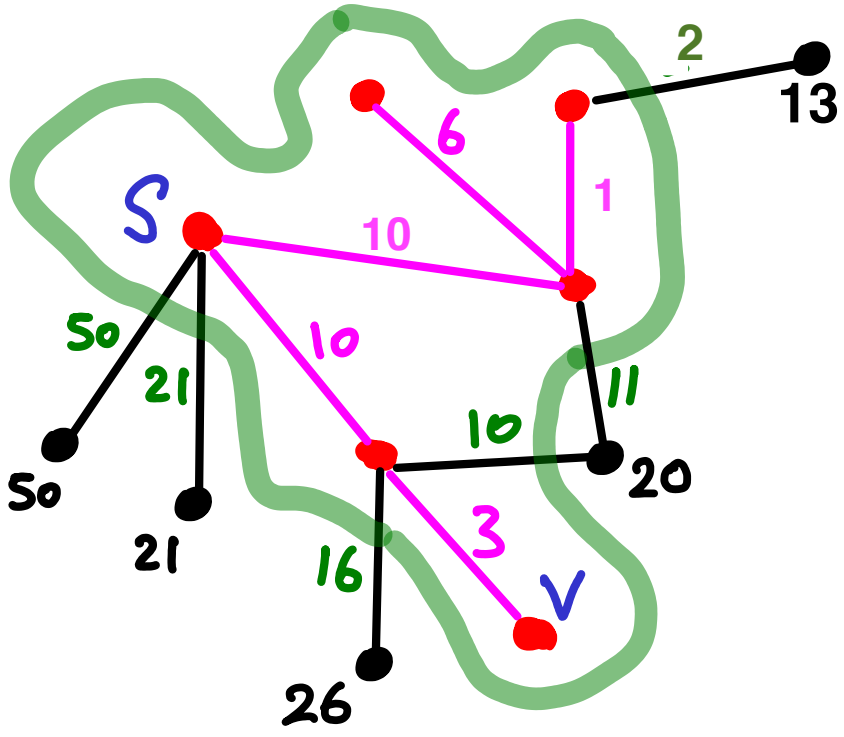
v = "closest" to S → **safe to add**

Any other path s ~~~ v
will cost at least as much

} assuming weights ≥ 0

Correctness:

assume we have shortest paths
to a set of red vertices



Just across CUT,
vertices have finite scores
= [a path in known set] + black edge

v = "closest" to S → **safe to add**

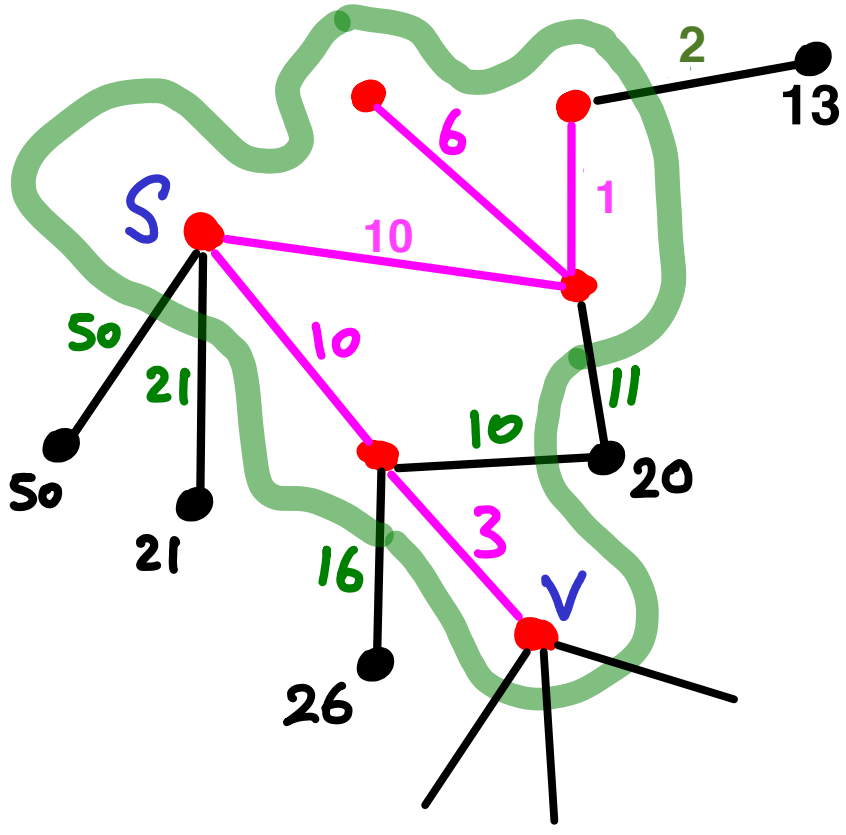
Any other path s ~~~ v
will cost at least as much

} assuming weights ≥ 0

Correctness:

assume we have shortest paths
to a set of red vertices

Just across CUT,
vertices have finite scores
= [a path in known set] + black edge



v = "closest" to S → safe to add

Any other path s ~~~ v
will cost at least as much

assuming weights ≥ 0

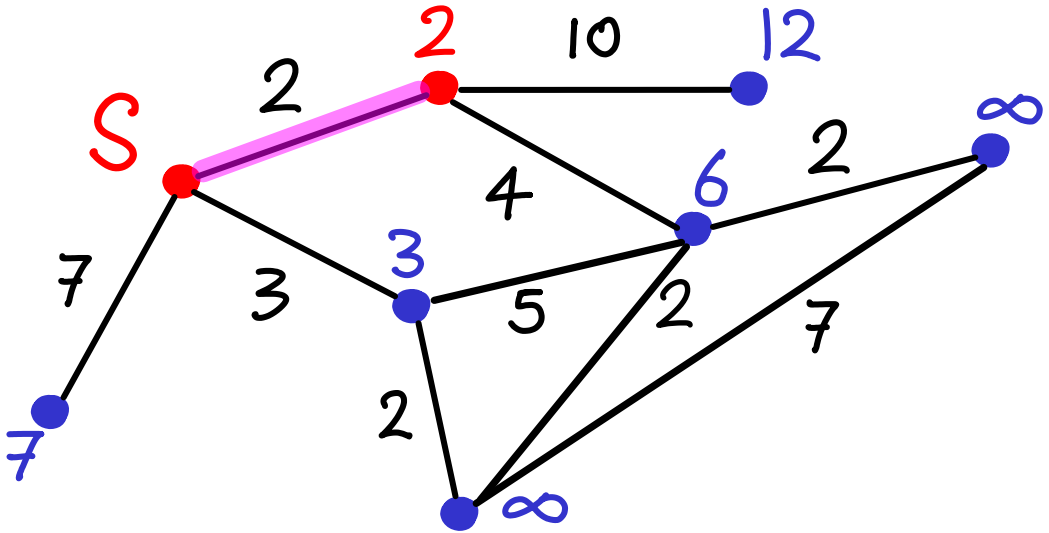
add "closest" vertex

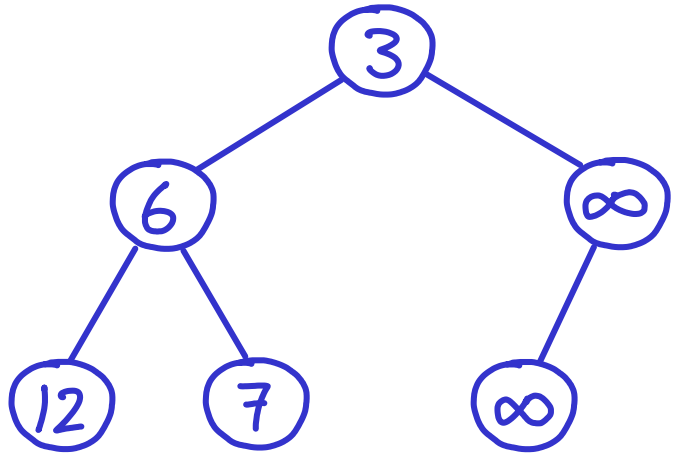
update set

= extract lowest score

= relax incident edges

Use a priority queue: heap



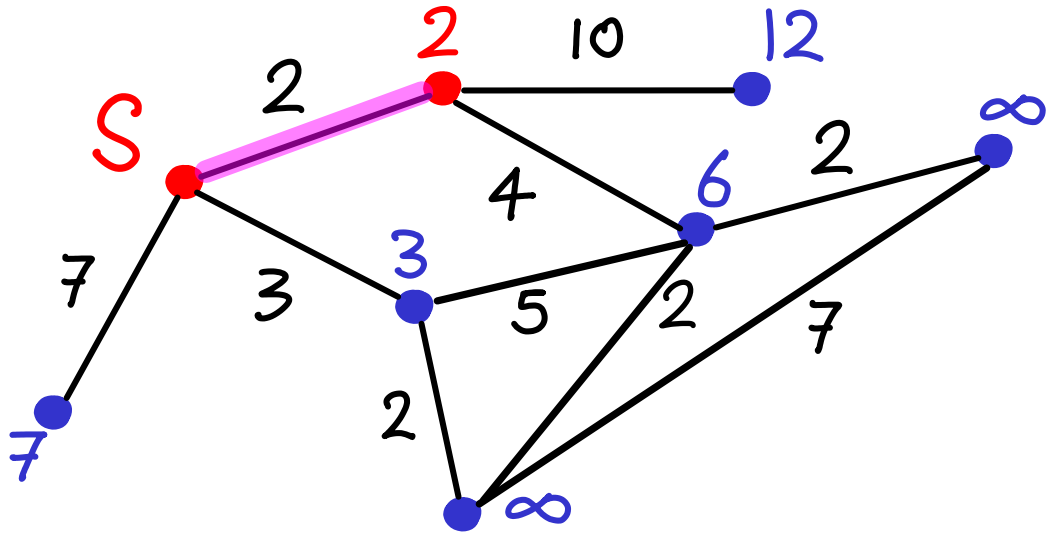


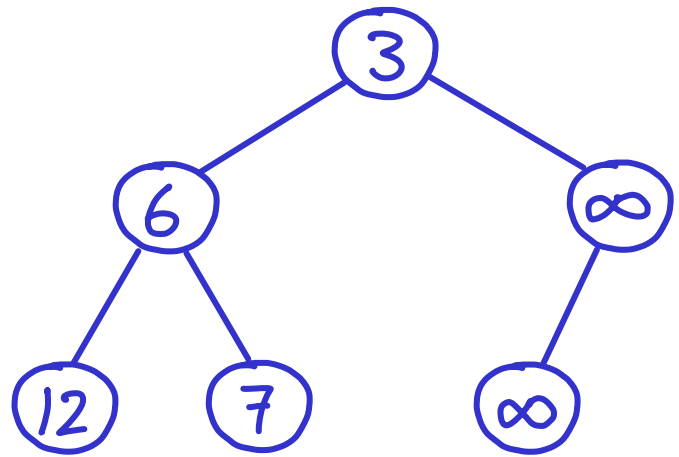
while priority queue not empty

| x: extract min (add to SSSP)

| for each neighbor y of x

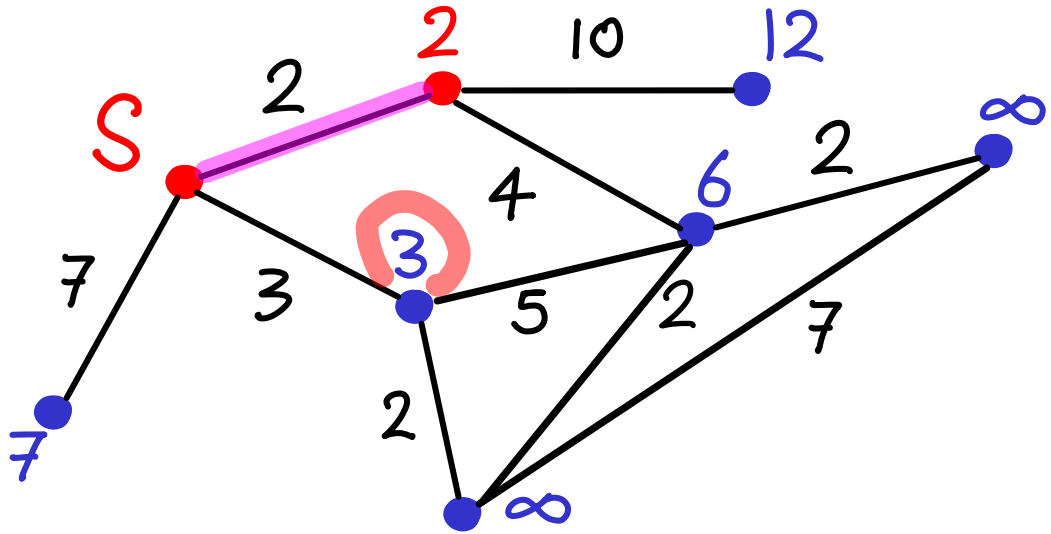
| RELAX(x,y)

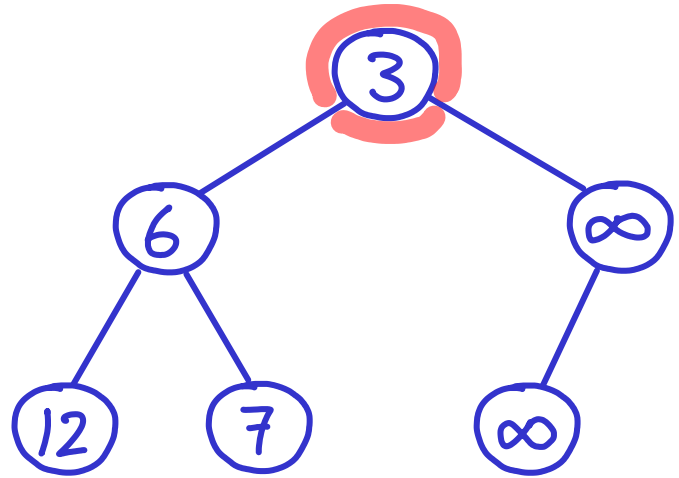




while priority queue not empty

- x: extract min (add to SSSP) → time?
  - for each neighbor y of x
    - RELAX(x,y)



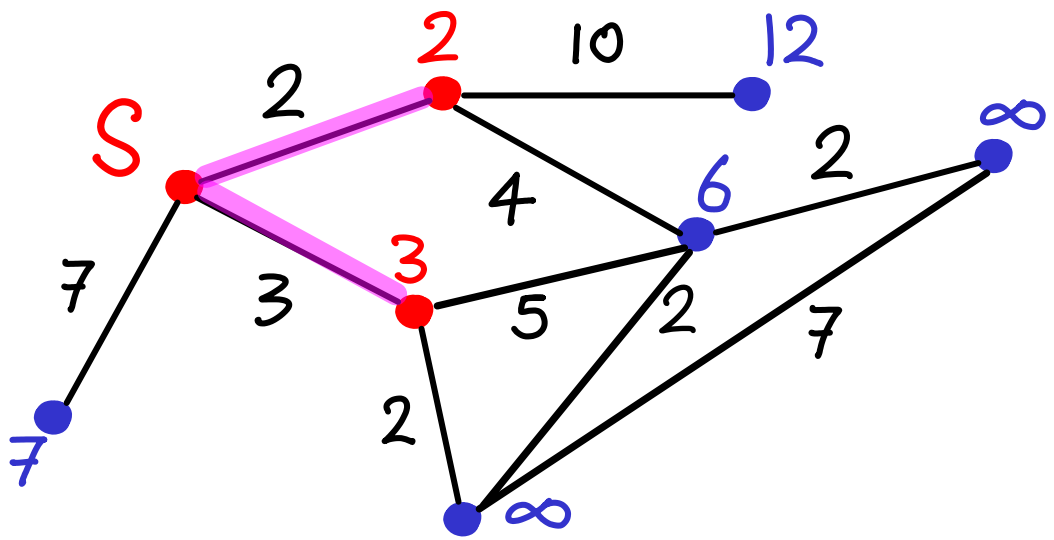


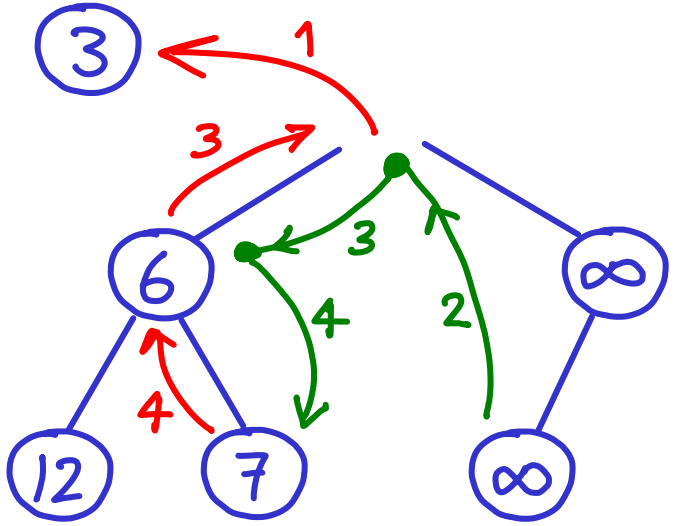
while priority queue not empty

| x: extract min (add to SSSP) → O(logV)

| for each neighbor y of x

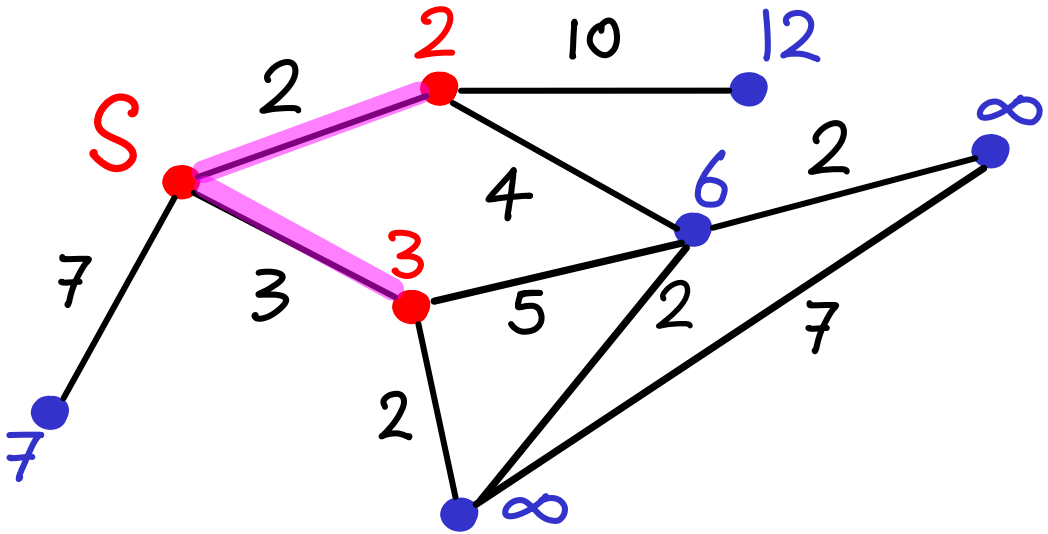
| RELAX(x,y)

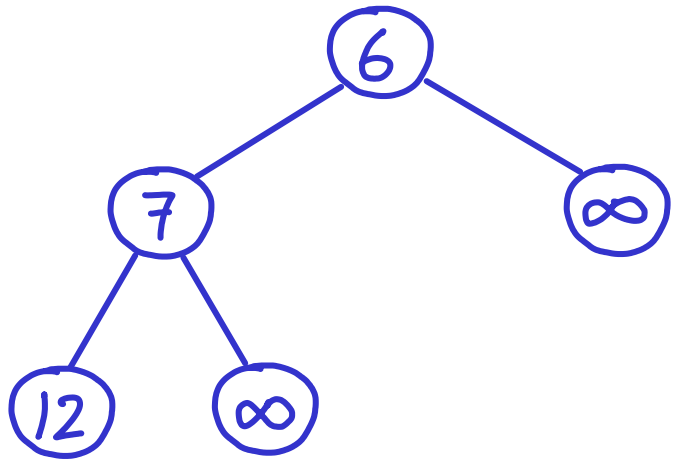




while priority queue not empty

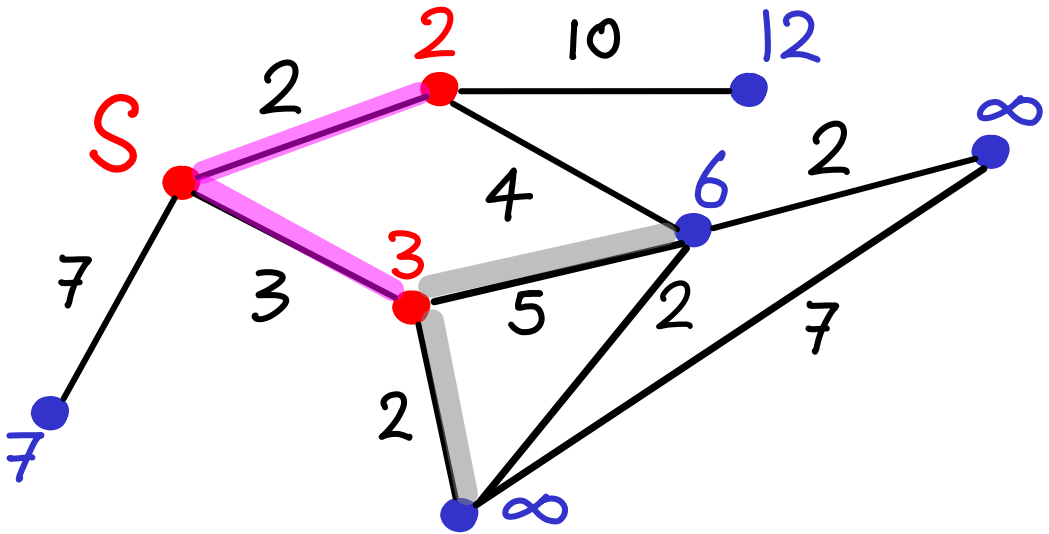
| x: extract min (add to SSSP) → O(logV)
| for each neighbor Y of x → how many?
| RELAX(x,Y)

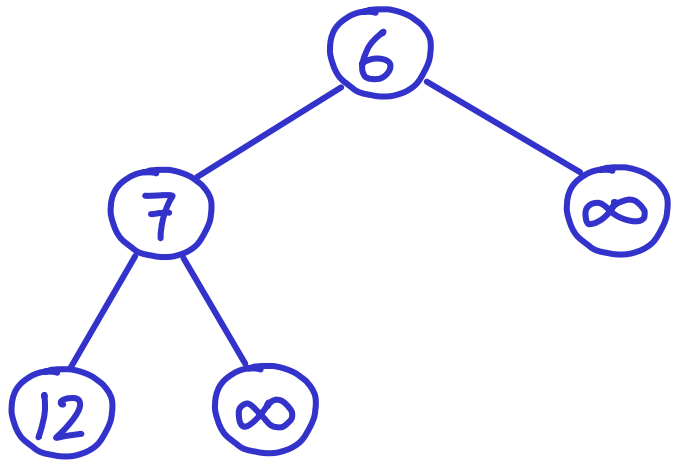




while priority queue not empty

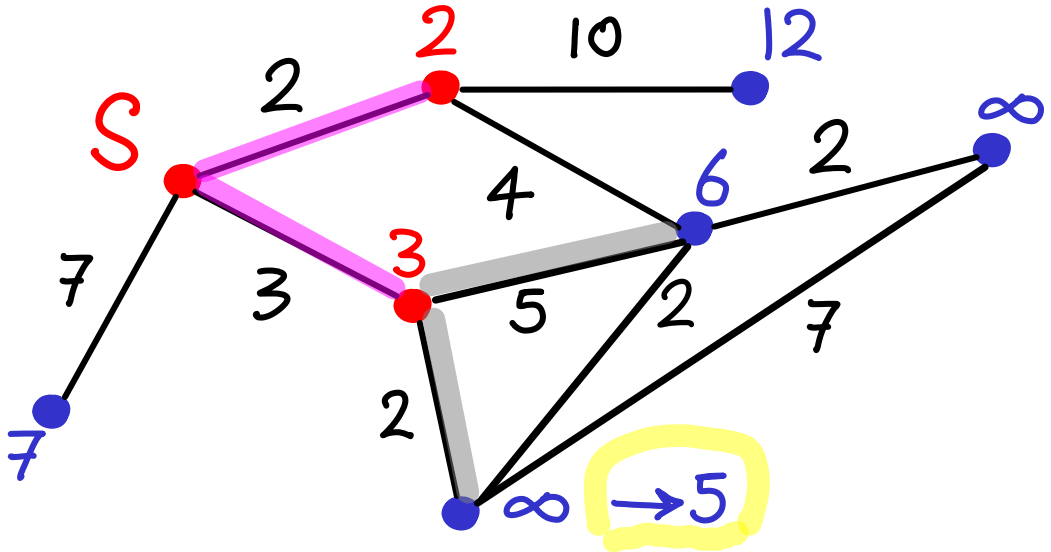
| x: extract min (add to SSSP) → O(logV)
| for each neighbor Y of x → O(deg(x))
| RELAX(x,Y) → cost?

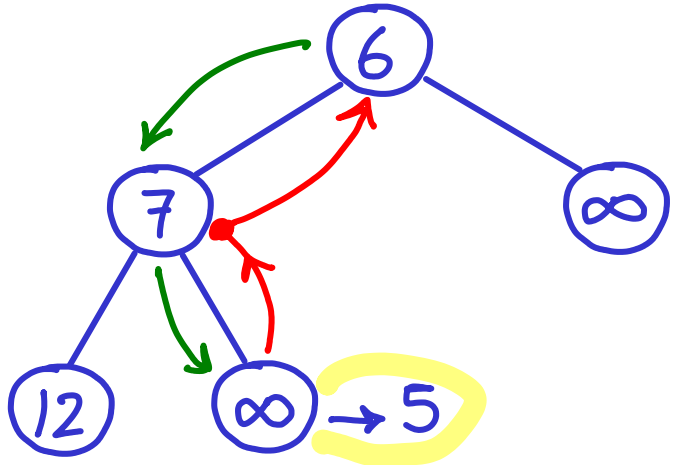




while priority queue not empty

- x: extract min (add to SSSP) → $O(\log V)$
- for each neighbor y of x → $O(\deg(x))$
  - RELAX(x,y) → decrease key → $O(\log V)$

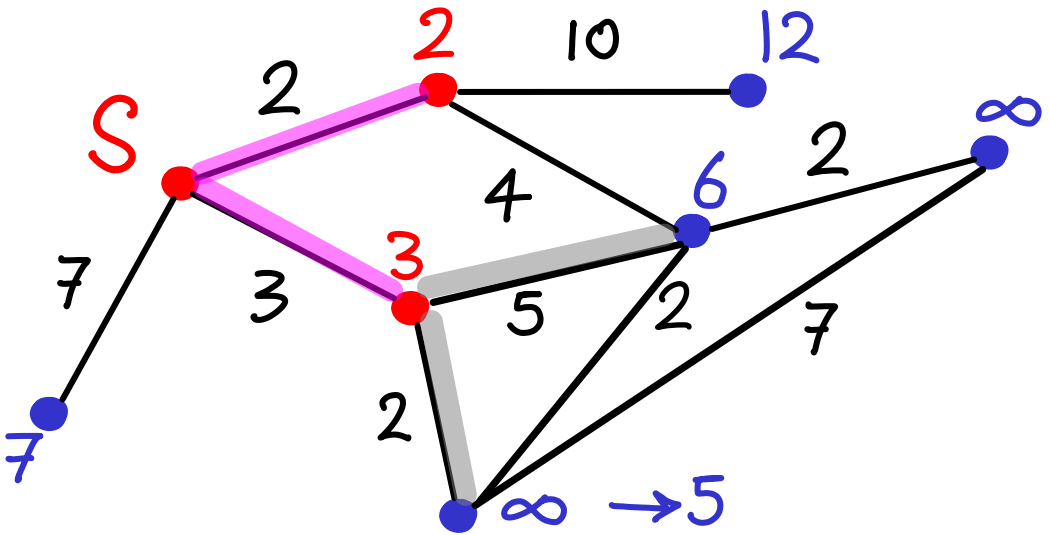


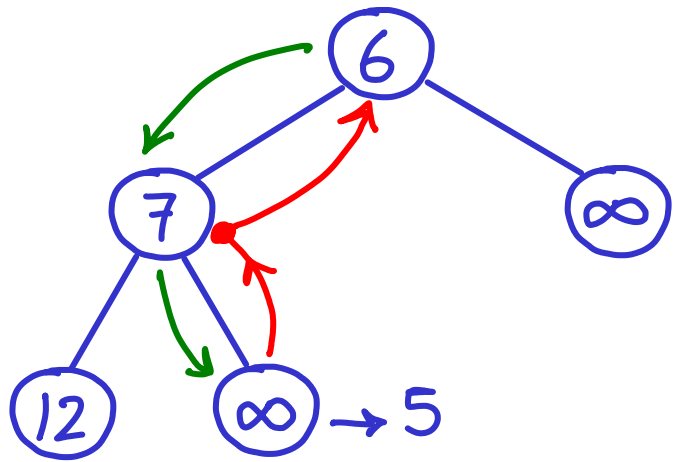


while priority queue not empty

TOTAL

x: extract min (add to SSSP)	→ O(logV)	O(VlogV)
for each neighbor y of x	→ O(deg(x))	?
RELAX(x,y) → decrease key	→ O(logV)	?





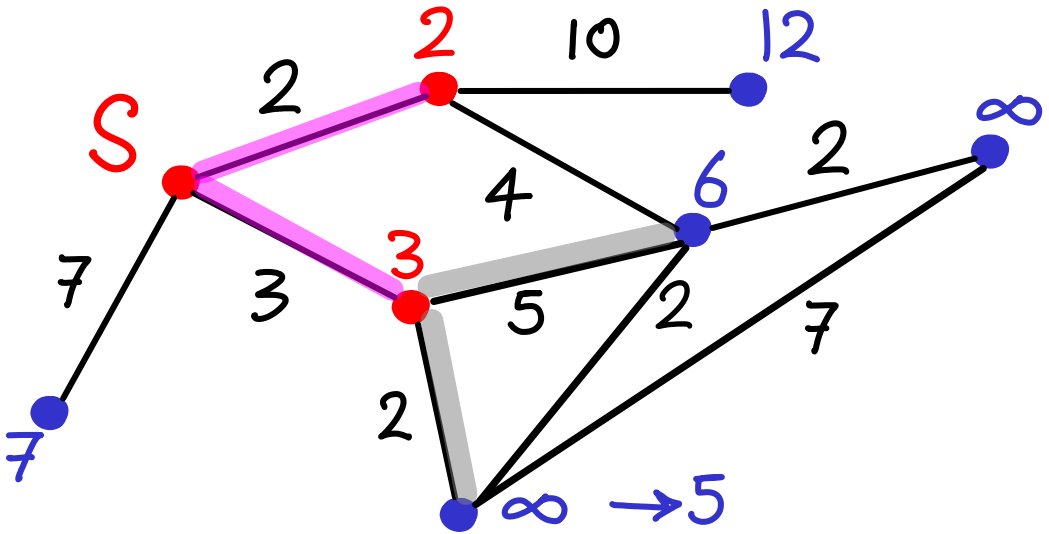
while priority queue not empty

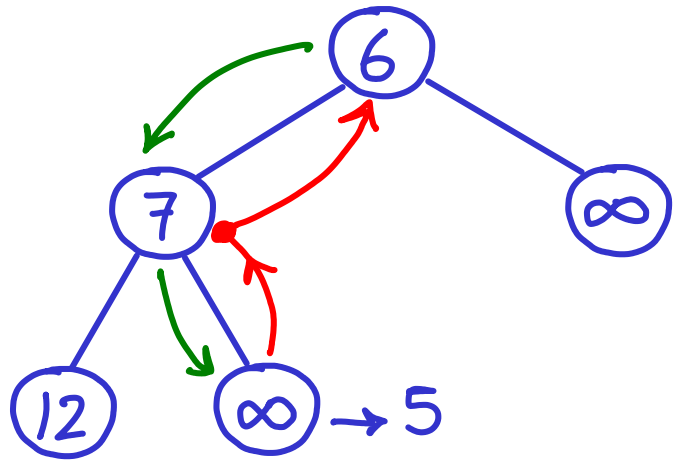
TOTAL

- x: extract min (add to SSSP) → O(logV) —— O(VlogV)
- for each neighbor y of x → O(deg(x)) }
  - RELAX(x,y) → decrease key → O(logV) }

O(ElogV)

if adj. list





add "closest" vertex

update set

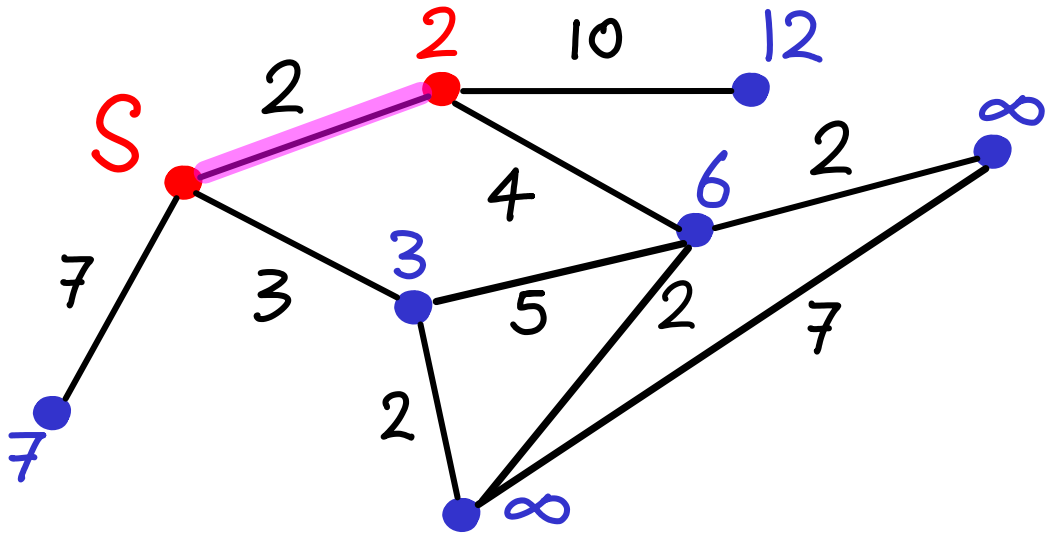
= extract lowest score

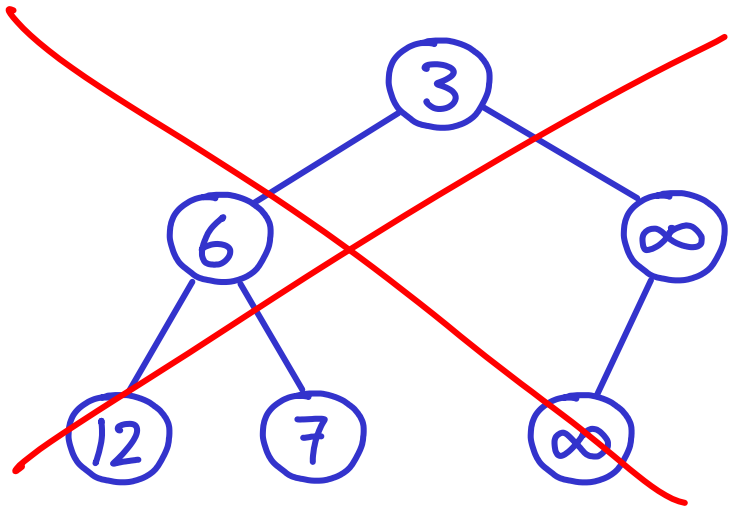
= relax incident edges

= O(V)

= O(V)

If adjacency matrix, don't use queue → $O(V^2)$





DIJKSTRA'S ALGORITHM

Summary: $O(E \log V)$ or $O(V^2)$

$\downarrow$

$V \log V + E \cdot (\text{decrease key})$

DIJKSTRA'S ALGORITHM

Summary: $O(E \log V)$ or $O(V^2)$

$\downarrow$

$V \log V + E \cdot (\text{decrease key})$

FYI: Fibonacci heap does decrease-key in $O(1)$ amortized

↳ $O(E + V \log V)$