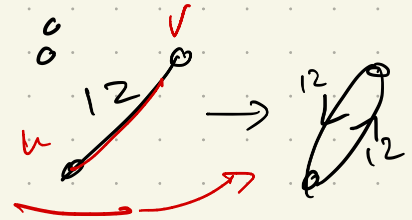
Algorithms : fall 2020

Shortest paths
(part 1)

Next problem: Shortest paths

Goal: Find shortest path from s to v.

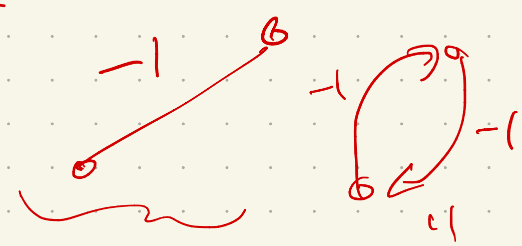
We'll think directed, but really could do undirected w/no negative edges:



Motivation:

- maps
- routing

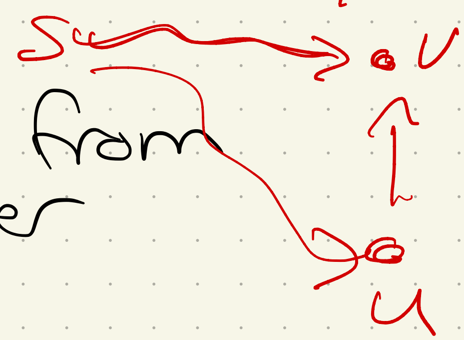
abstract but weighted



introduced neg. cycles

Usually, to solve this, need to solve a more general problem:

Find shortest paths from s to every other vertex.

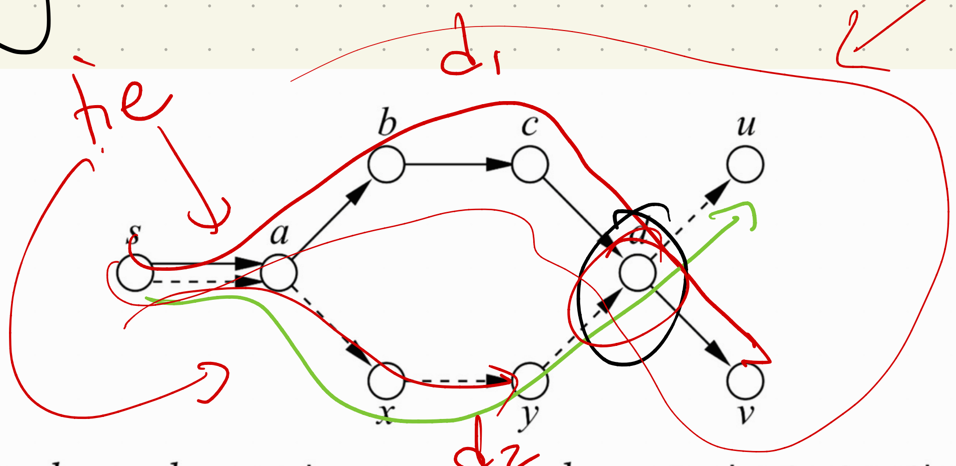


Called the single-source shortest path tree.

Some notes:

- Why a tree?

make it a tree



If $s \to a \to b \to c \to d \to v$ and $s \to a \to x \to y \to d \to u$ are shortest paths, then $s \to a \to b \to c \to d \to u$ is also a shortest path.

- Negative ~~edges~~? cycles

shortest s-t path

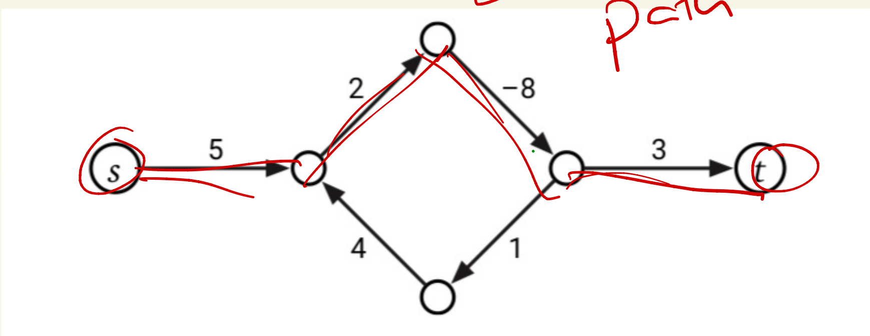
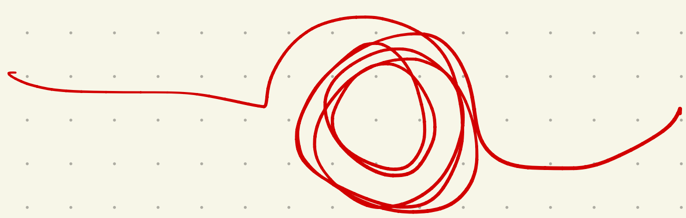
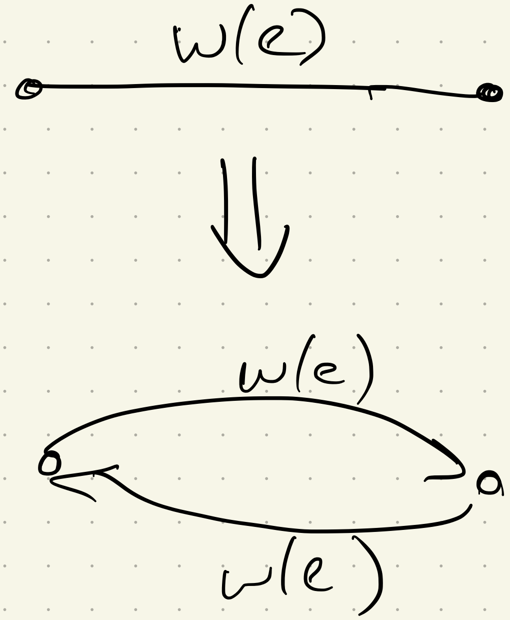


Figure 8.3. There is no shortest walk from s to t.



Also: If undirected, can simulate w/ directed.

ie



Unless!! negative edges

so here: directed.

B/c gets weird:

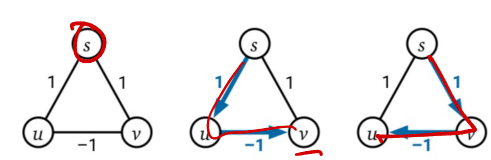
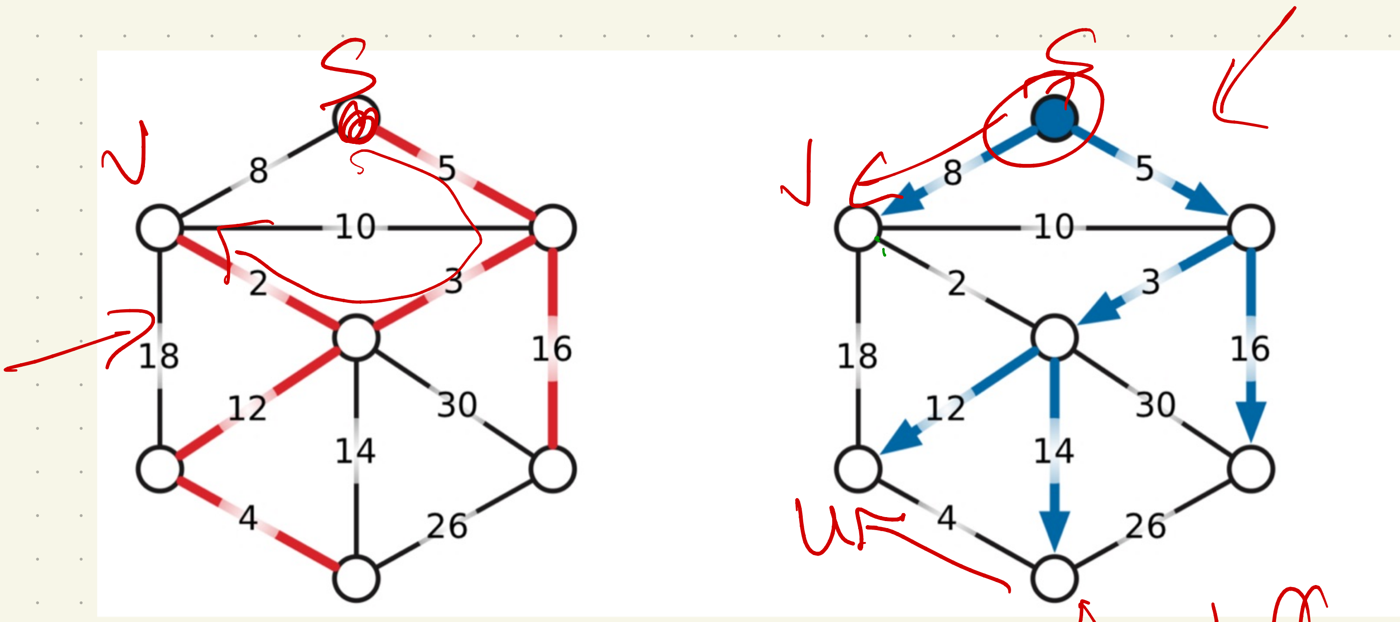


Figure 8.4. An undirected graph where shortest paths from s are unique but do not define a tree.

How to solve?? flows → in a few chapters

Important to realize:

MST ≠ SSSP



t: different SSSP tree

Why?

MST - global smallest object.

SSSP tree: distance to one vertex

Computing a SSSP:

(Ford 1956 & Dantzig 1957)

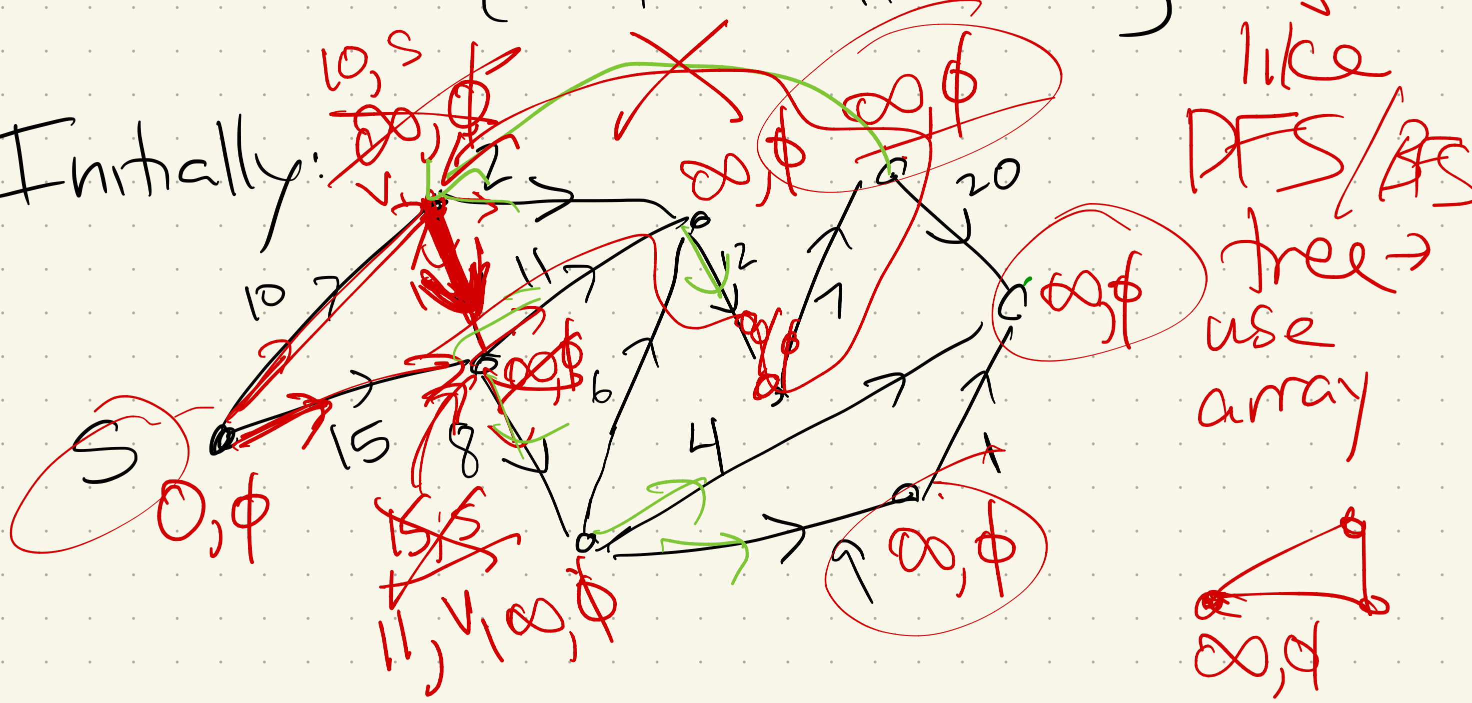
Each vertex will store 2 values.

(Think of these as tentative shortest paths)

- dist(v) is length of tentative shortest s⇝v path
  (or ∞ if don't have an option yet)

- pred(v) is the predecessor of v on that tentative path s⇝v
  (or NULL if none)

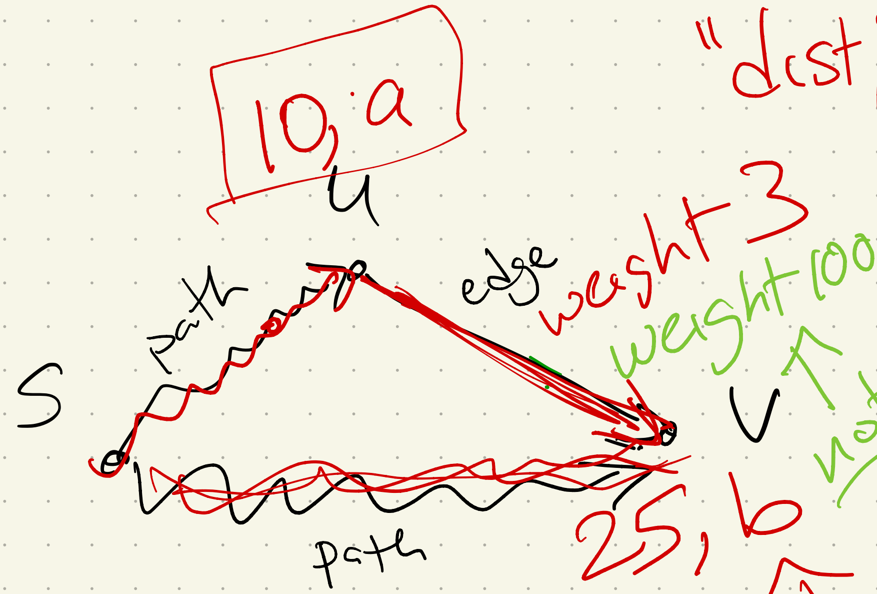
Initially:



We say an edge $\overrightarrow{uv}$ is <u>tense</u>

if $\underline{dist(u)} + \underline{w(u \to v)} < \underline{dist(v)}$

"dist" is that best guess so far



better path to v!

v is bad guess → could improve

If $u \to v$ is tense:

update w/ better path!

$dist(v) = dist(u) + w(u \to v)$

+ update previous node

So, relax:

RELAX$(u \to v)$:
  $dist(v) \leftarrow dist(u) + w(u \to v)$
  $pred(v) \leftarrow u$

Algorithm:

Repeatedly find tense edges + relax them.

When none remain, the pred(v) edges form the SSSP tree.

← needs a proof!!

```
InitSSSP(s):
    dist(s) ← 0
    pred(s) ← Null
    for all vertices v ≠ s
        dist(v) ← ∞
        pred(v) ← Null
```

```
GenericSSSP(s):
    InitSSSP(s)
    put s in the bag
    while the bag is not empty
        take u from the bag
        for all edges u→v
            if u→v is tense
                Relax(u→v)
                put v in the bag
```

To do: which "bag"?

← data structure

runtime: how many times can an edge be tense?

"Easy"(??) warm-up:

distance = # of edges

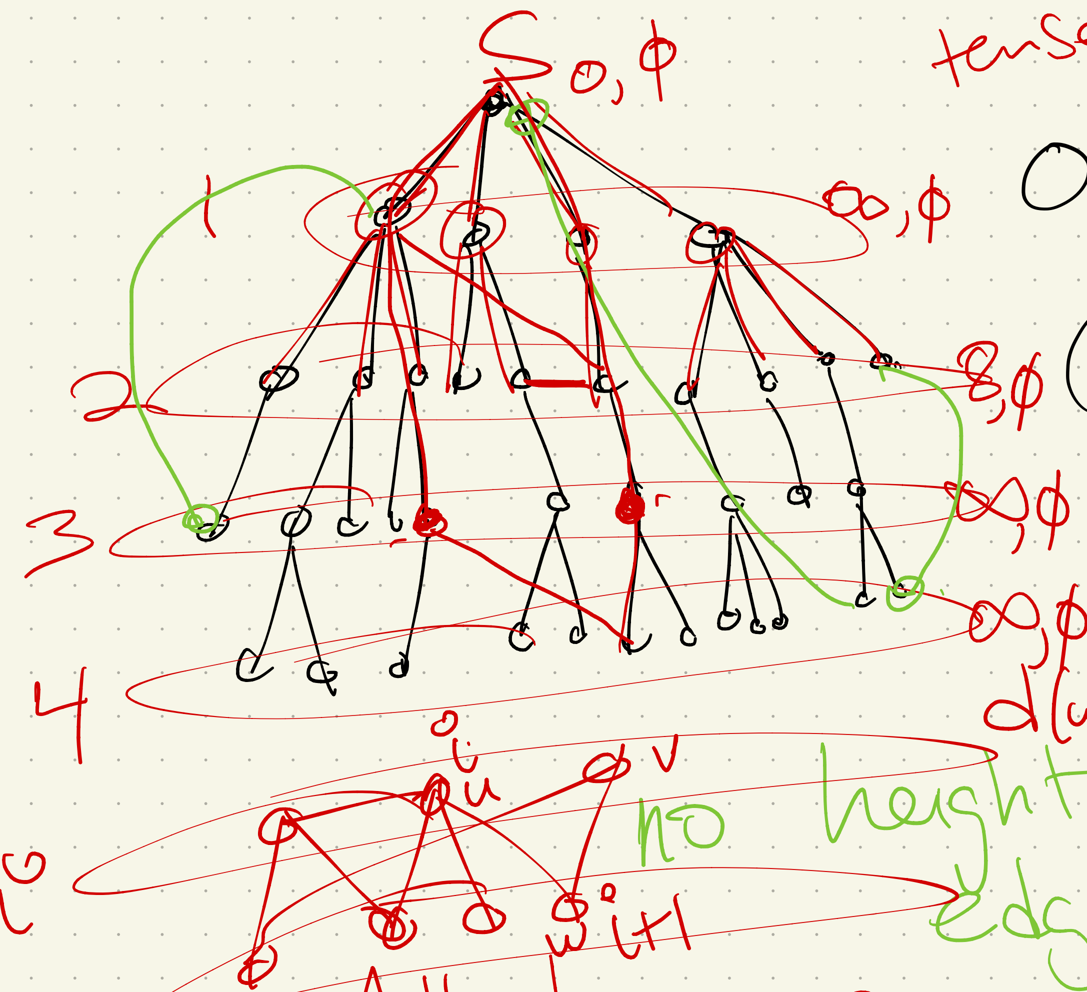
What if unweighted?

→ use a queue

How does "tense" work?

(Hint: think BFS!)

each edge in tense once!



Other edges?

(ie can anything be tense?)

d(u)+1 beats d(w) through other path

no height 2 edges

All edges in same level, or 1 up

What the heck is his token??

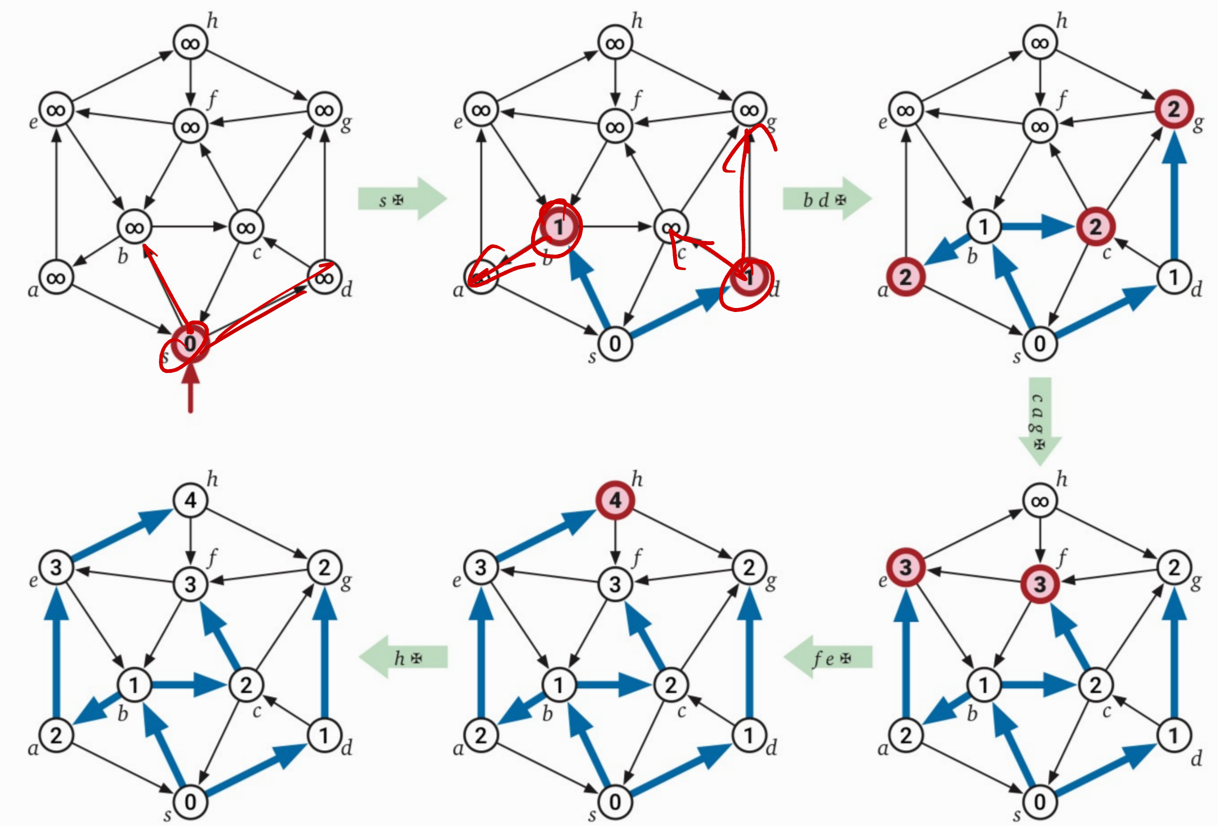
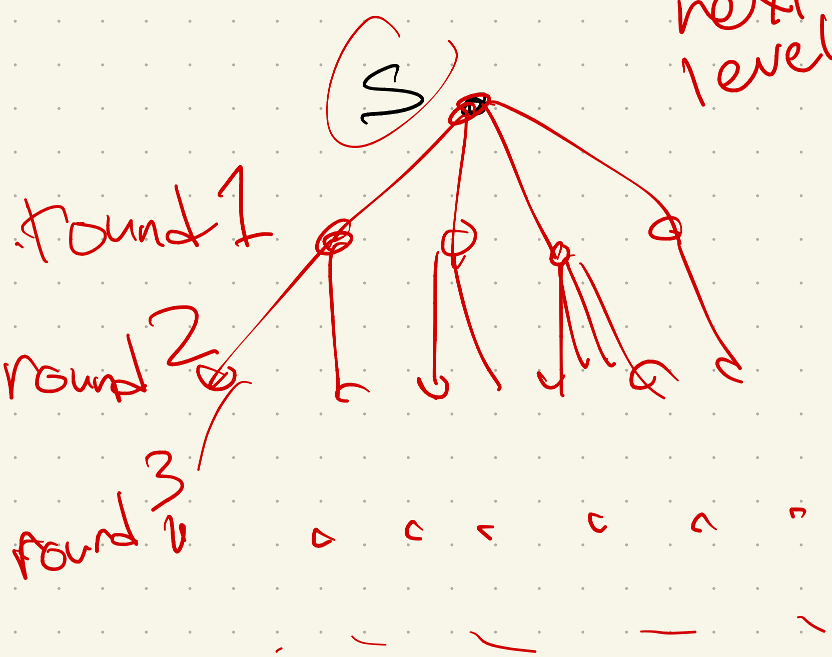


Figure 8.6. A complete run of breadth-first search in a directed graph. Vertices are pulled from the queue in the order s ✠ b d ✠ c a g ✠ f e ✠ h ✠ ✠, where ✠ is the end-of-phase token. Bold vertices are in the queue at the end of each phase. Bold edges describe the evolving shortest path tree.

next level was empty



queue:

~~s's nbrs ✠~~

distance 2 nodes from s

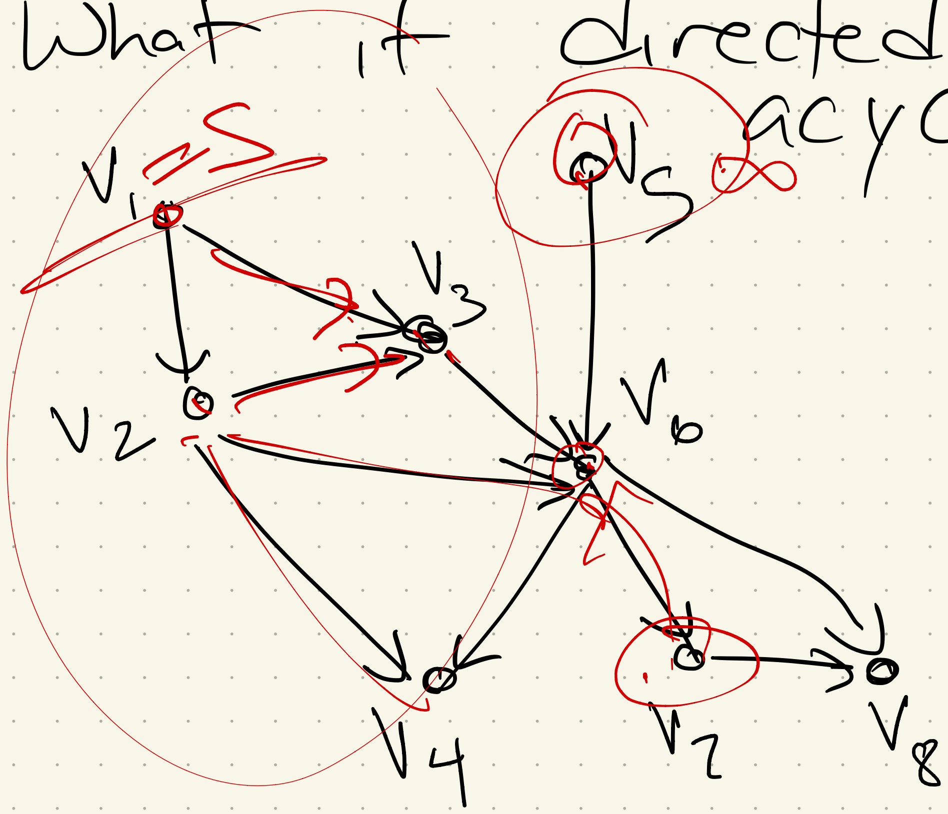
~~✠~~ dist 3 nodes ✠

~~✠~~ empty ✠

no more new nbrs

2^{nd} version (warm-up)

What if directed & acyclic?



$v_1\ v_2 \to v_3\ v_5$

$v_6\ v_4\ |\ v_7\ v_8$

know prev "best" paths

Remember: helps to have all "closer" vertices done before computing your distance. no cycles!

Well, know something: topological order! (Ch 6)

get an order that does all "closer" edges to s first.

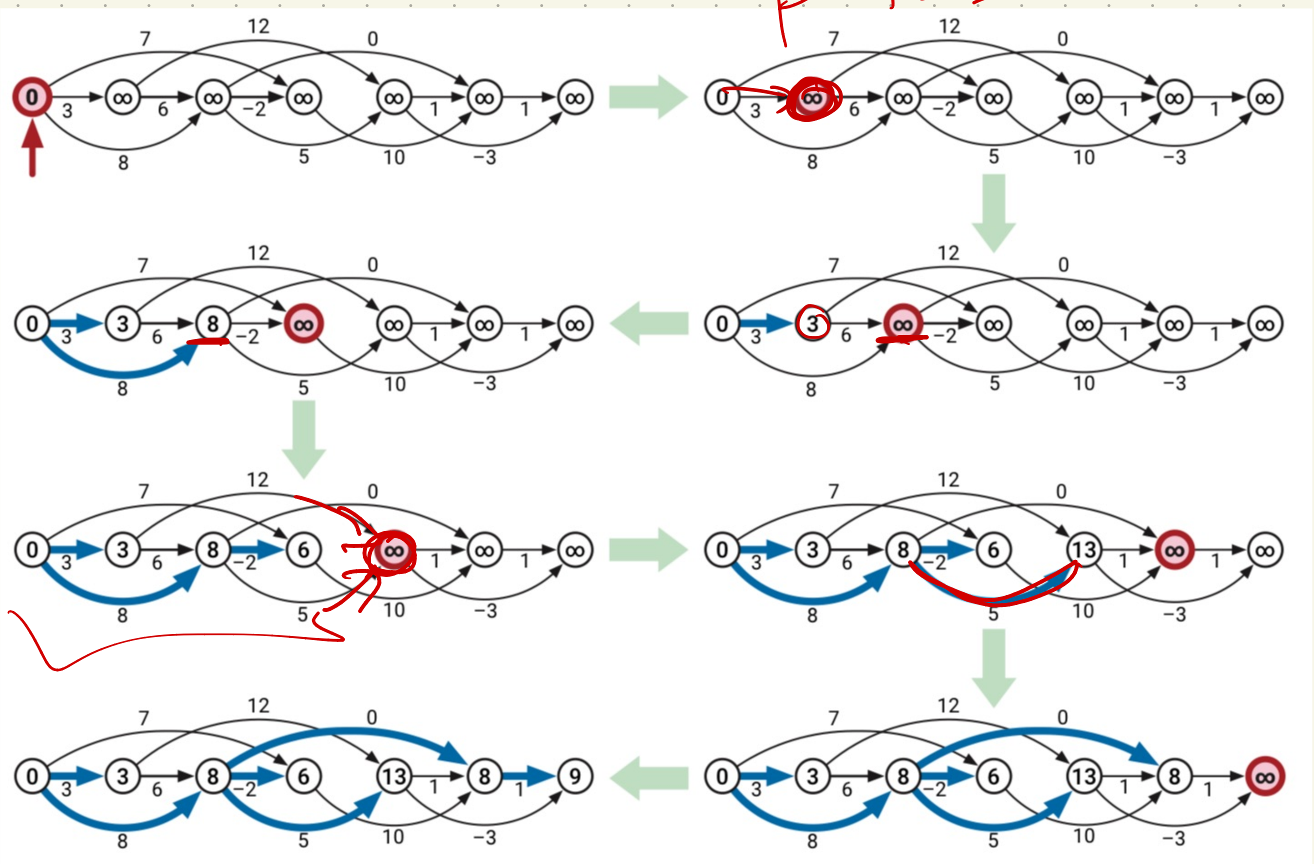
So, use it!

```
DAGSSSP(s):
  INITSSSP(s)
  for all vertices v in topological order
    for all edges u→v
      if u→v is tense
        RELAX(u→v)
```

O(1)

note p. # of code in book

try all possible paths



Runtime: $O(\text{top sort}) + \underbrace{\sum_v d(v)}_{= 2E}$

$O(V+E)$

$= \underline{O(V+E)}$

impressive!

Dijkstra ('59)

(actually Leyzorek et al '57, Dantzig '58)

Make the bag a priority queue:

Keep "explored" part of the graph, S.

Initially, $S = \{s\}$ + dist(s)=0 (all others NULL + ∞)

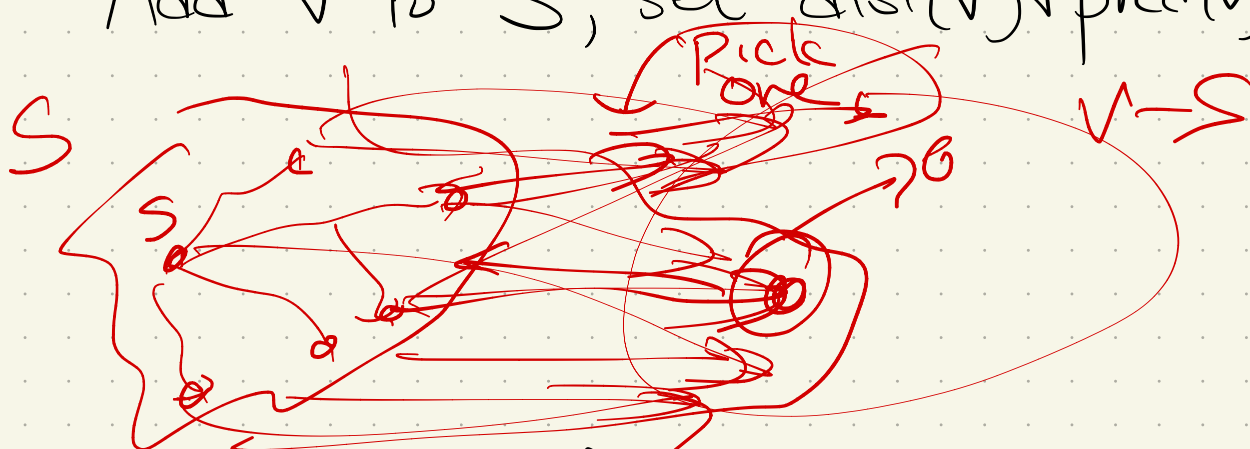
While $S \neq V$:

grow S

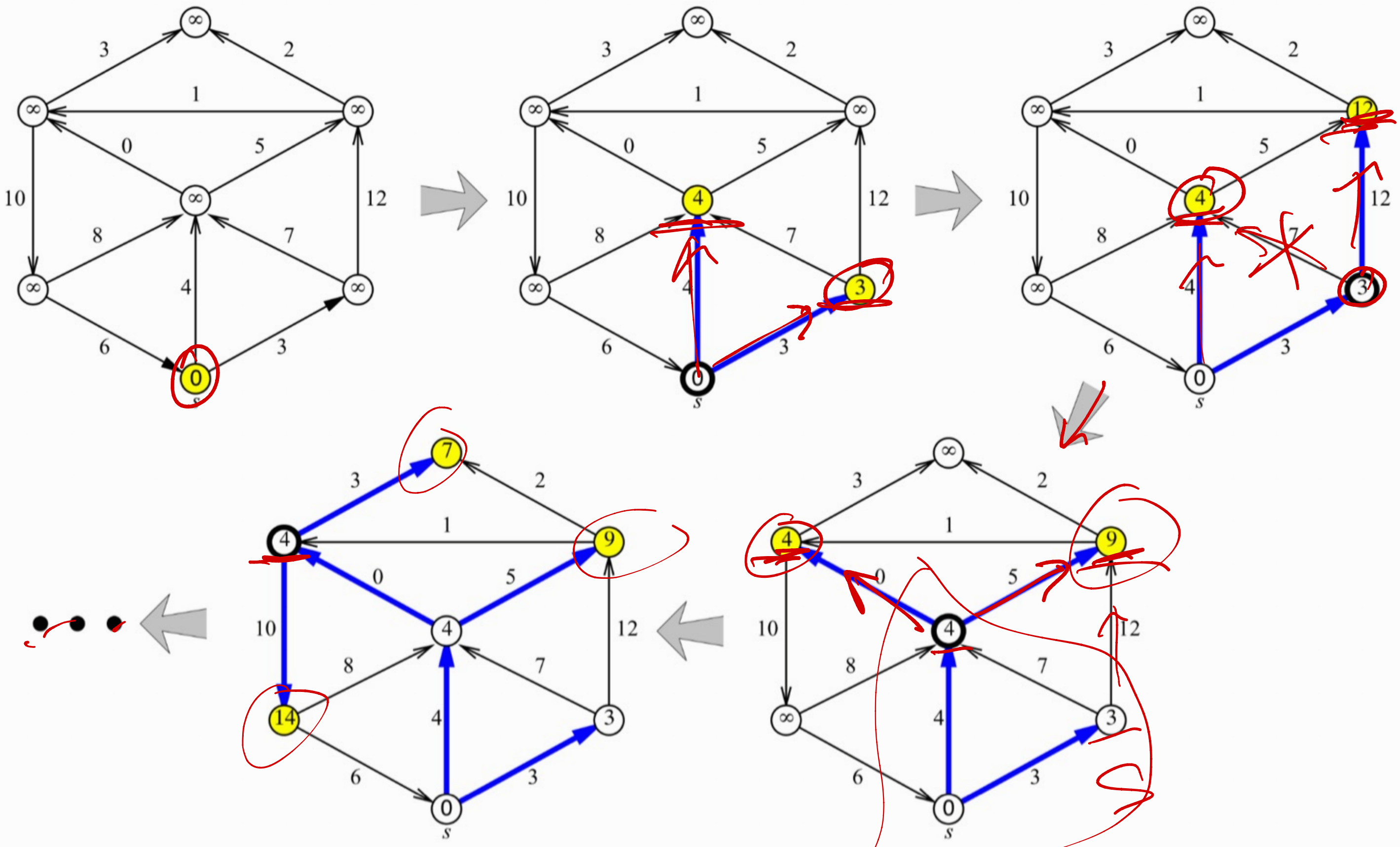
Select node $v \notin S$ with one edge from S to v with:

greedy! $\min_{e=(u,v),\, u \in S}$ dist(u) + w(u→v) "tensest" edge

Add v to S, set dist(v) + pred(v)



Picture ⟶



Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

(find nice animations!)

Correctness

Thm: Consider the set S at any point in the algorithm. For each $u \in S$, the distance $dist(u)$ is the shortest path distance (so $pred(u)$ traces a shortest path).

Pf: Induction on $|S|$:

Base case: if $|S| = 1$:

then $S = \{s\}$ (one vertex)

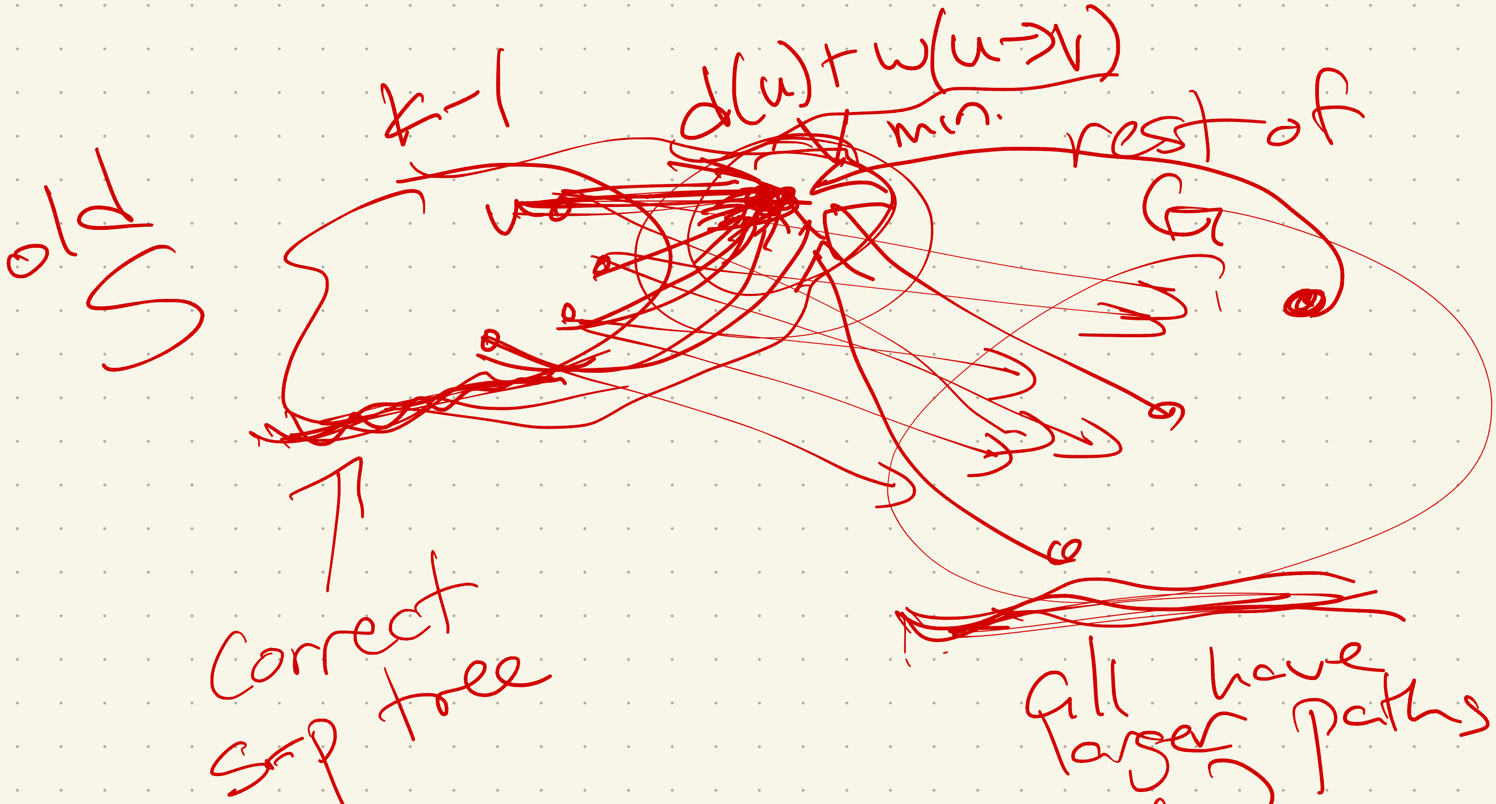
s has distance 0 to itself!

so $(s, \emptyset)$ is correct

IH: Spps claim holds when $|S| = k-1$.

IS: Consider $|S| = k$:

algorithm is adding some v to S



Why is this correct?

all other vertices would give larger s-p to v (since not tense)

Back to implementation + run time:

For each $v \in S$, could check each edge + compute $D[v] + w(e)$

runtime?

Better: a heap!

When v is added to S:

- look at v's edges and either insert w with key dist(v) + w(v→w) or update w's key if dist(v) + w(v→w) beats current one

Runtime:

- at most m ChangeKey operations in heap
- at most n inserts/removes