Algorithms

Shortest Paths

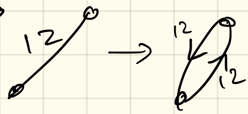
Recap

- HW Posted
- Reading for Mon. & Wed. posted.
- Stay tuned for Friday

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

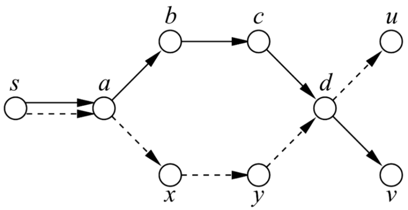
Usually, to solve this, need to solve a more general problem:

Find shortest paths from s to every other vertex.

Called the single-source shortest path Tree.

Some notes:

- Why a tree?



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.

If 2 shortest paths cross twice, subpaths must be tied

- Negative edges?

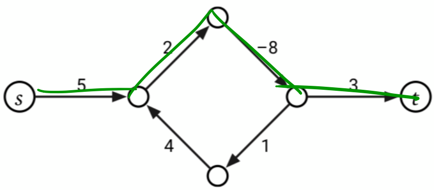
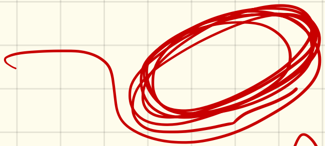


Figure 8.3. There is no shortest walk from s to t.

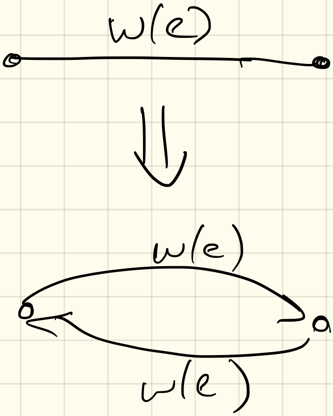
no repeat edges

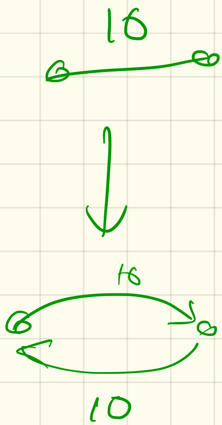


$-\infty$

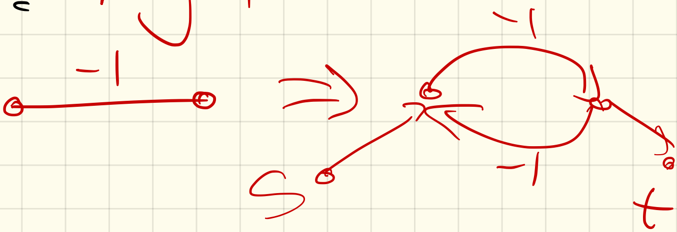
Also: If undirected, can simulate w/ directed.

ie





Unless!! negatives



B/c gets weird:

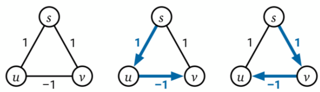


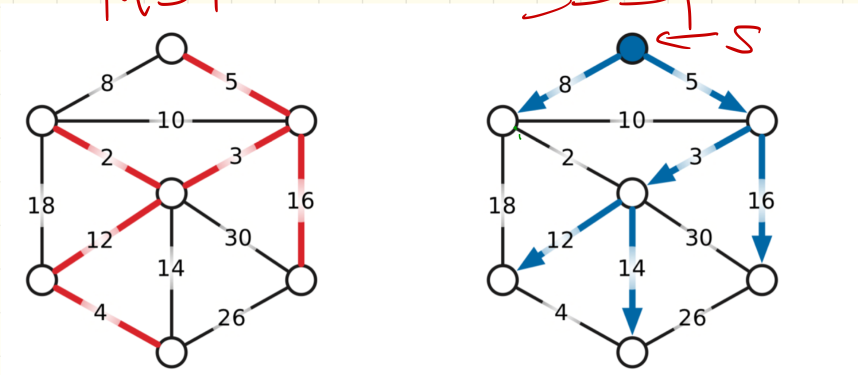
Figure 8.4. An undirected graph where shortest paths from s are unique but do not define a tree.

How to solve?? Flows!

Important to realize:

MST $\neq$ SSSP

MST SSSP $\leftarrow s$



Why? MST is optimizing some global structure

SSSP is local.

Computing a SSSP:

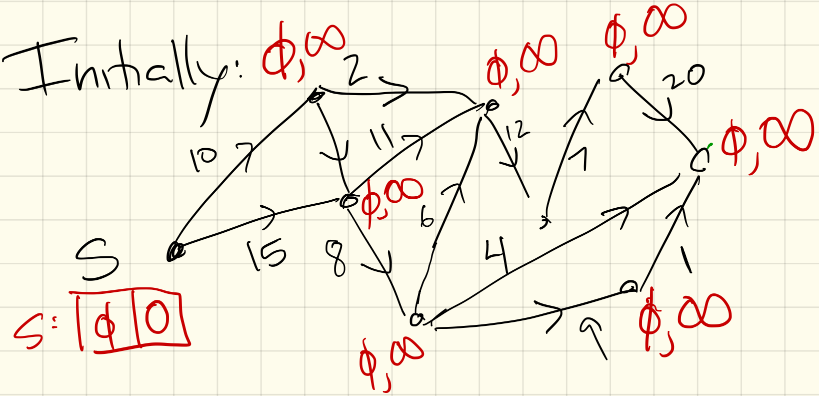
(Ford 1956 & Dantzig 1957)

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)

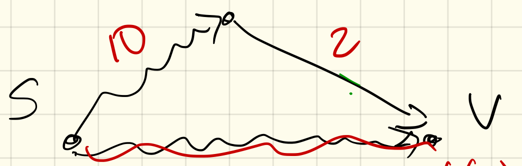
Initially:



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

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

pred(u), dist(u)



pred(v), $\underline{\text{dist}(v)}$

$\emptyset$ 20

If $u \to v$ is tense:

path via u is better

so: $\text{pred}(v) = u$

$\text{dist}(v) = \text{dist}(u) + w(u \to v)$

So, relax:

<u>RELAX($u \to v$):</u>
$\quad dist(v) \leftarrow dist(u) + w(u \to v)$
$\quad pred(v) \leftarrow u$

Algorithm:

Repeatedly find tense edges & relax them.

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

```
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"?

(+ proof)

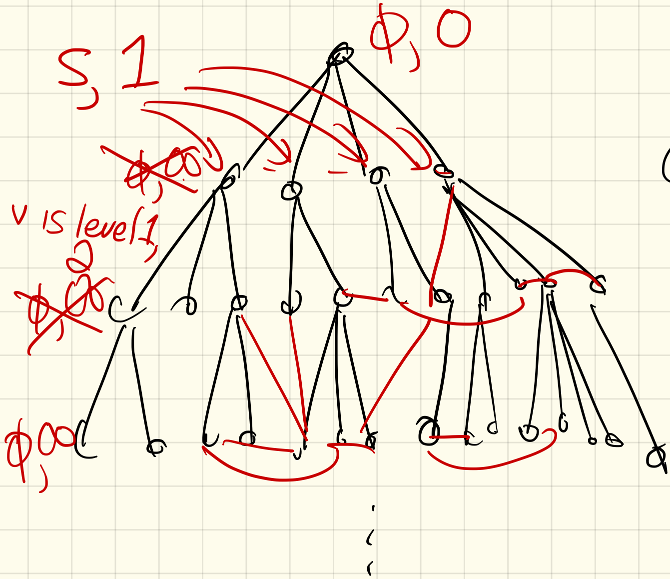
"Easy" (??) warm-up:

What if unweighted?

→ use a queue

How does "tense" work?

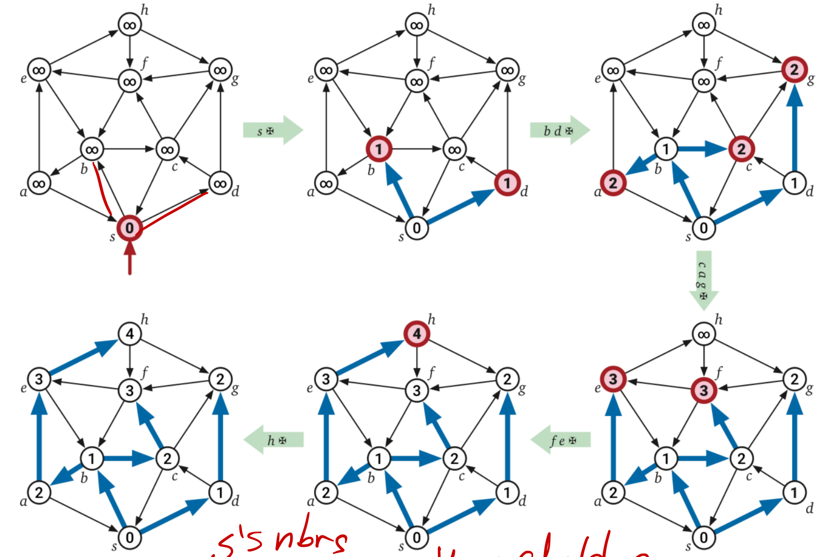
(Hint: think BFS!)



Other edges?

(ie can anything be tense?)

What the heck is his token??



s's nbrs

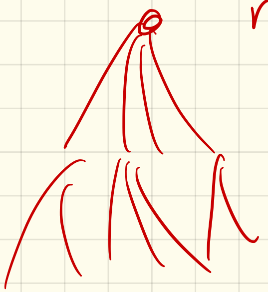
their children

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.

nbrs of s's children

round 1

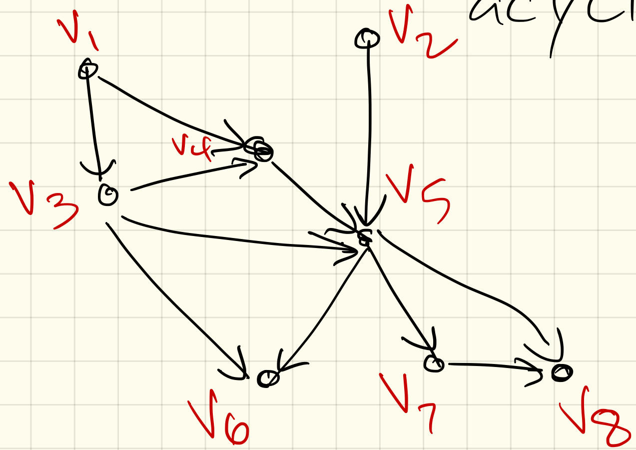
round 2



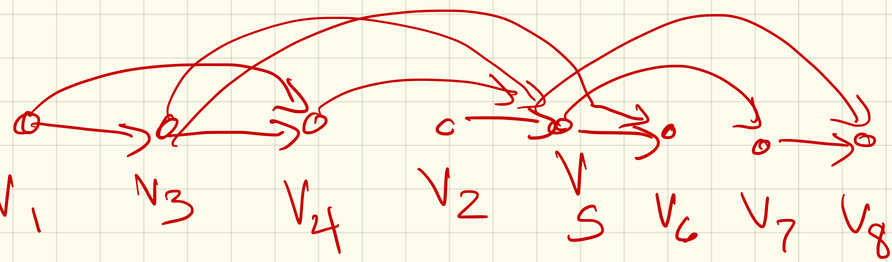
round 3

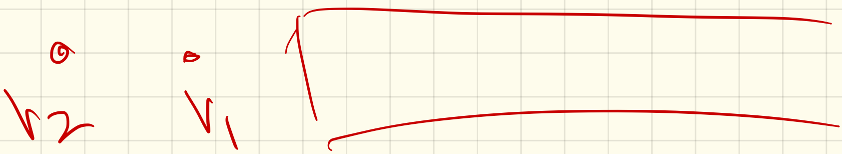
2^{nd} version

What if directed & acyclic?



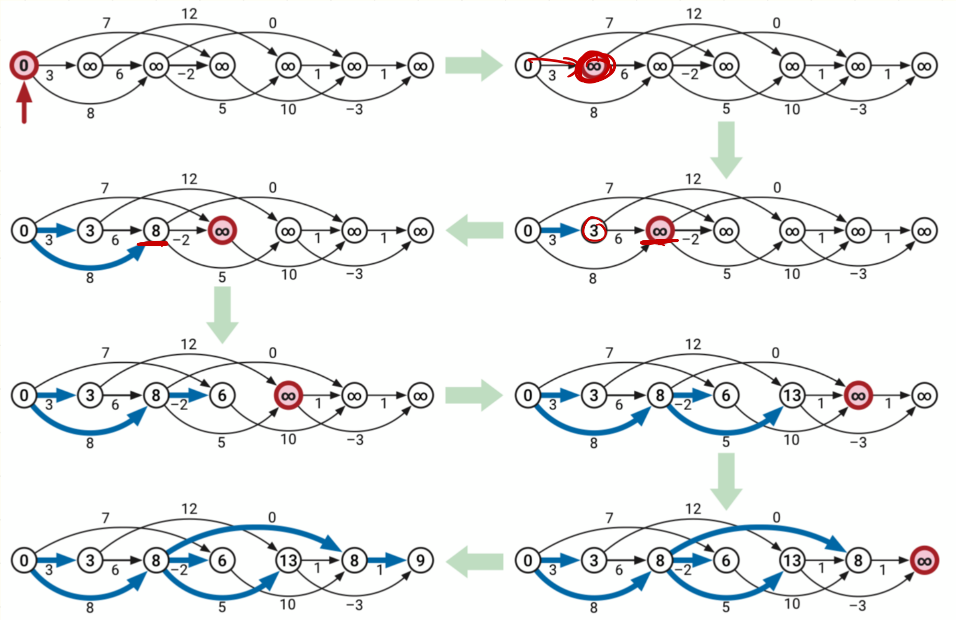
Well, know something: topological order!





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)
```



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 + ∞)

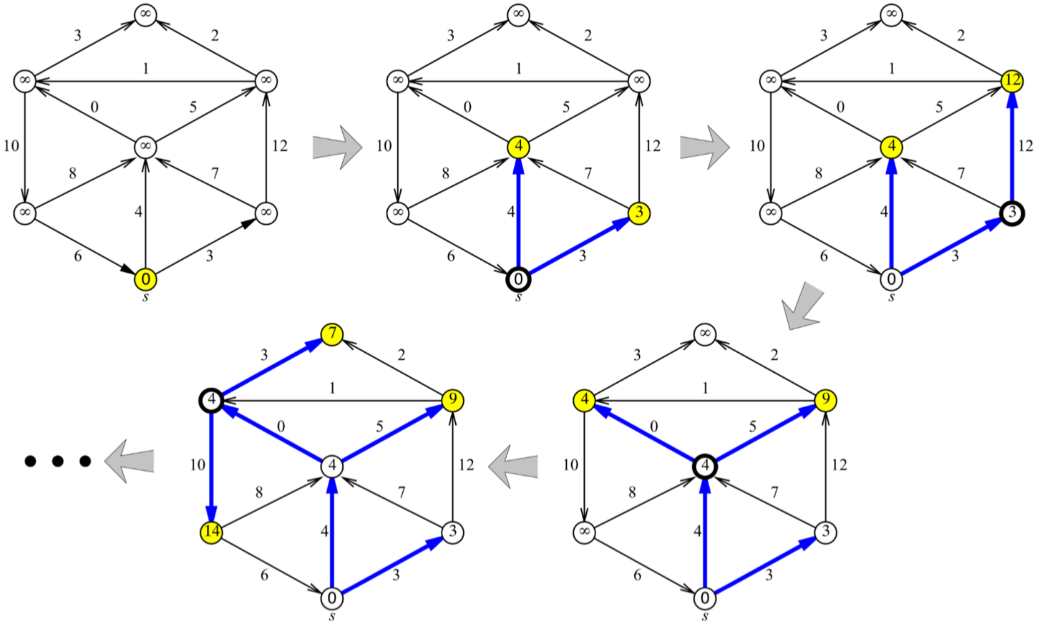
While $S \neq V$:

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

$$\min_{e=(u,v),\, u \in S} dist(u) + w(u \to v)$$

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

Picture $\longrightarrow$



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.

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:

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

IS: Consider $|S| = k$:

algorithm is adding some v to S

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 $\text{dist}(v) + w(v \to w)$ or update w's key if $\text{dist}(v) + w(v \to w)$ beats current one

Runtime:

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