Algorithms - Spring '25

MSTs
SSSPs

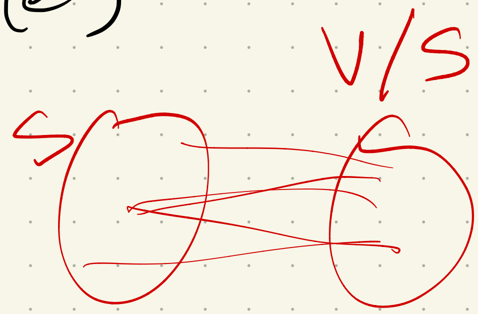
Recap:

- How's this view?
- Office hours:
  Thursday 1-2^{30} pm

Minimum Spanning Trees

Goal: Given a weighted undirected Graph G, $w: E \to \mathbb{R}^+$ the weight function, find a spanning tree T of G that minimizes:

$$w(T) = \sum_{e \in T} w(e)$$



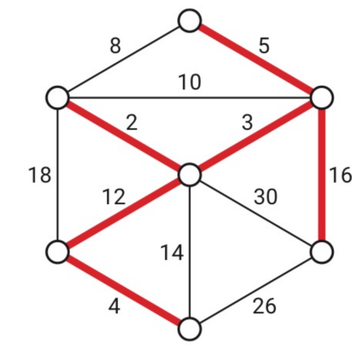


Figure 7.1. A weighted graph and its minimum spanning tree.

Key lemma: Given any $S \subseteq V$, the minimum edge xy from any $x \in S$ to some $y \in V/S$ is in the MST.

$\hookrightarrow$ "safe" edge

So: be greedy!

Boruvka: Build a forest, Initially V disjoint vertices.

While (F is not a tree)

$O(E)$ → Find all safe edges and add them to F

$O(V)$ → count conn. components in F

Runtime:

how many times loop repeats: $\log_2 V$

Worst case: $O(E \log V)$

min one is safe

Prim: Keep one spanning subtree.

Initially, $T = \{v\}$

While $|T| \neq V$:

add next safe edge

Runtime:

$\underbrace{E + V \log V}$

Kruskal's Algorithm :

KRUSKAL: Scan all edges by increasing weight; if an edge is safe, add it to F.

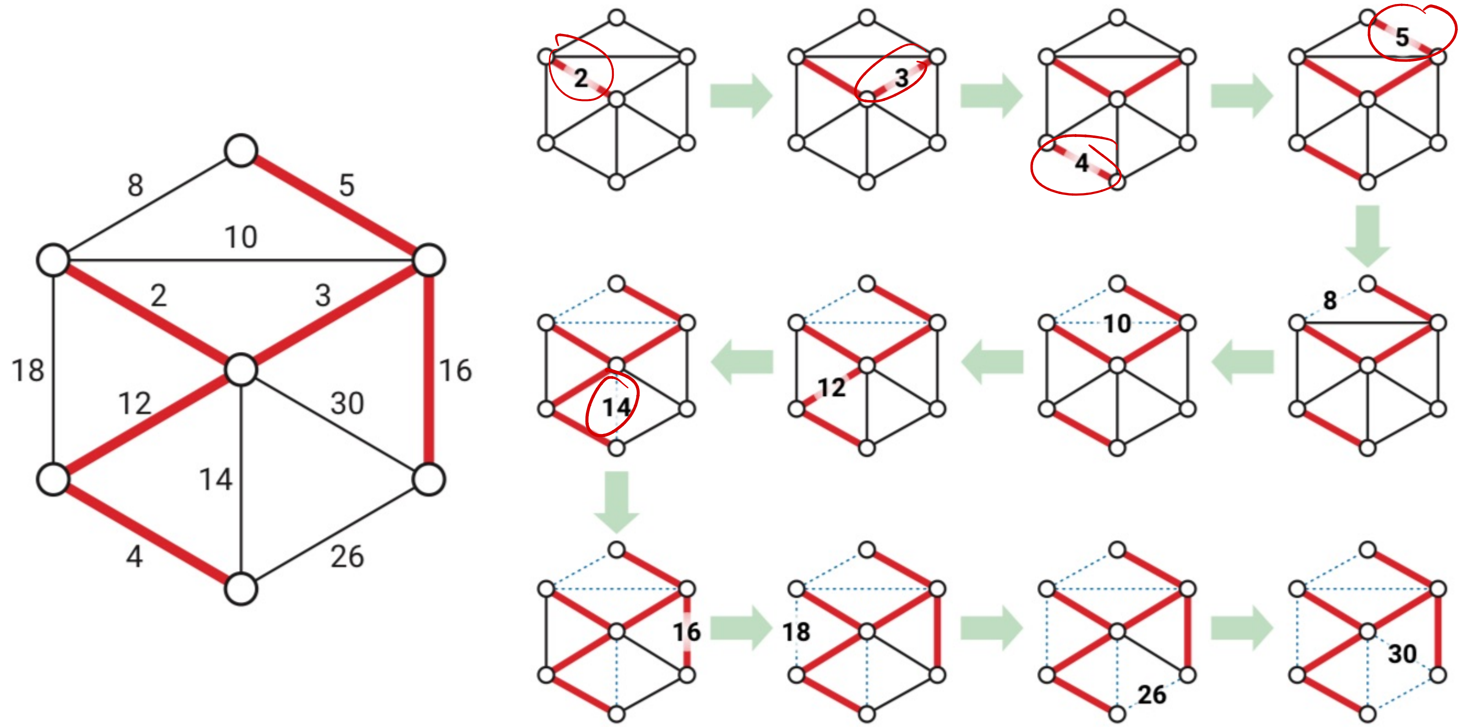


Figure 7.6. Kruskal's algorithm run on the example graph. Thick red edges are in F; thin dashed edges are useless.

How to implement? sort edges & loop over them. How to see if useless?

Data structure: Union find

one element

- MAKESET(v) — Create a set containing only the vertex v.
- FIND(v) — Return an identifier unique to the set containing v.
- UNION(u, v) — Replace the sets containing u and v with their union. (This operation decreases the number of sets.)

Then:

```
KRUSKAL(V, E):
    sort E by increasing weight
    F ← (V, ∅)
    for each vertex v ∈ V
        MAKESET(v)
    for i ← 1 to |E|
        uv ← ith lightest edge in E
        if FIND(u) ≠ FIND(v)
            UNION(u, v)
            add uv to F
    return F
```

Comparison:

- Boruvka: $O(E \log V)$
- Prim: $O(E + V \log V)$
- Kruskal: $O(E \log V)$

$(E \log E \leq E \log V^2 = 2E \log V$

Remember:

- Worst case here, plus hidden constants.
- Also: $E = O(V^2)$ but could be much smaller!

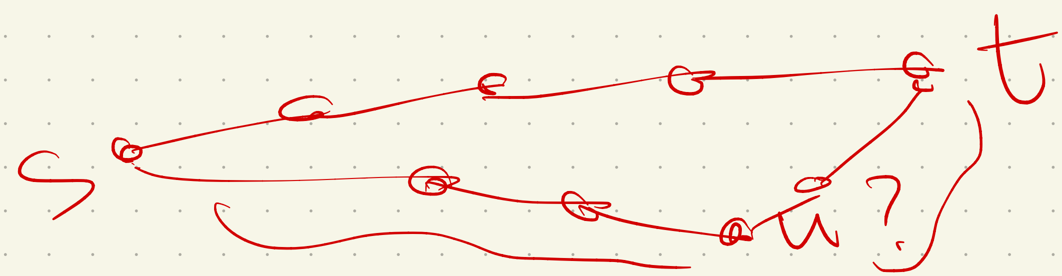
Next: Shortest paths

Goal: given $s, t \in V$, compute the shortest path from s to t.

Motivation: roads
routing
cost

To solve this, we need to solve a more general problem:
find shortest paths from s to <u>every</u> vertex.

Why?



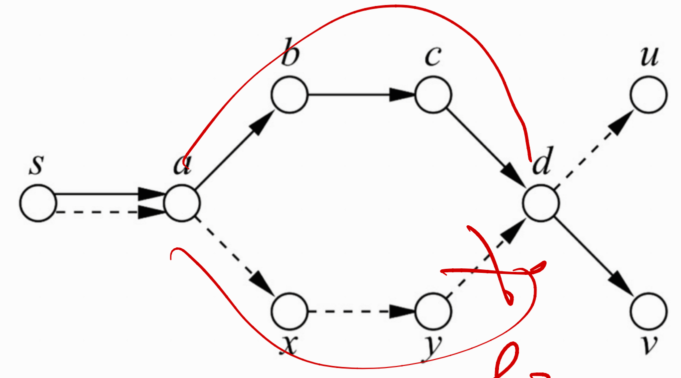
Single source shortest paths (SSSP)

Some notes:

Why a tree?

ℓ_1 ~~$\ell_1 < \ell_2$~~

$\ell_1 = \ell_2$



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.

What about negative cycles or edges?

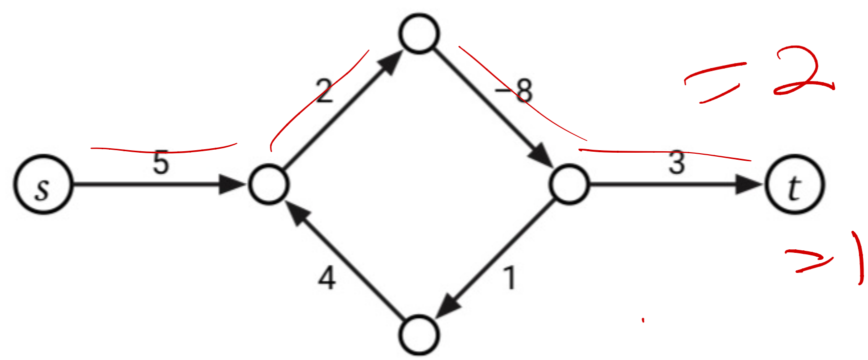
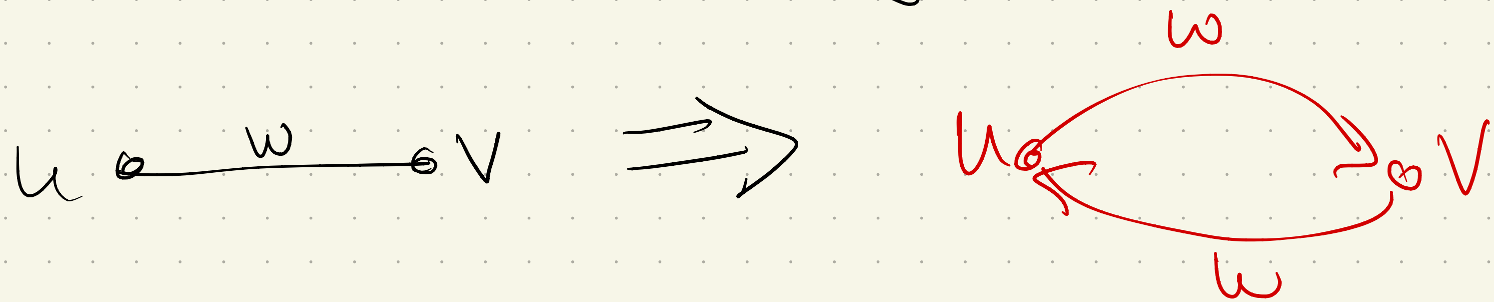


Figure 8.3. There is no shortest walk from s to t.

If neg cycles, "shortest" path could have infinite length.

Also: If undirected, can simulate with a directed graph:



Unless you have negative edges.
(It gets wierd.)

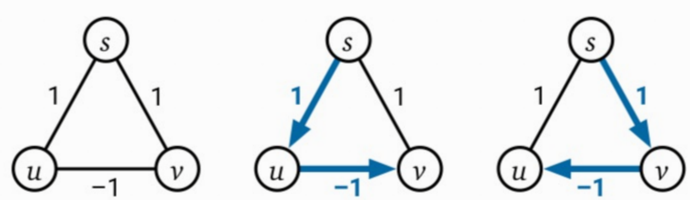
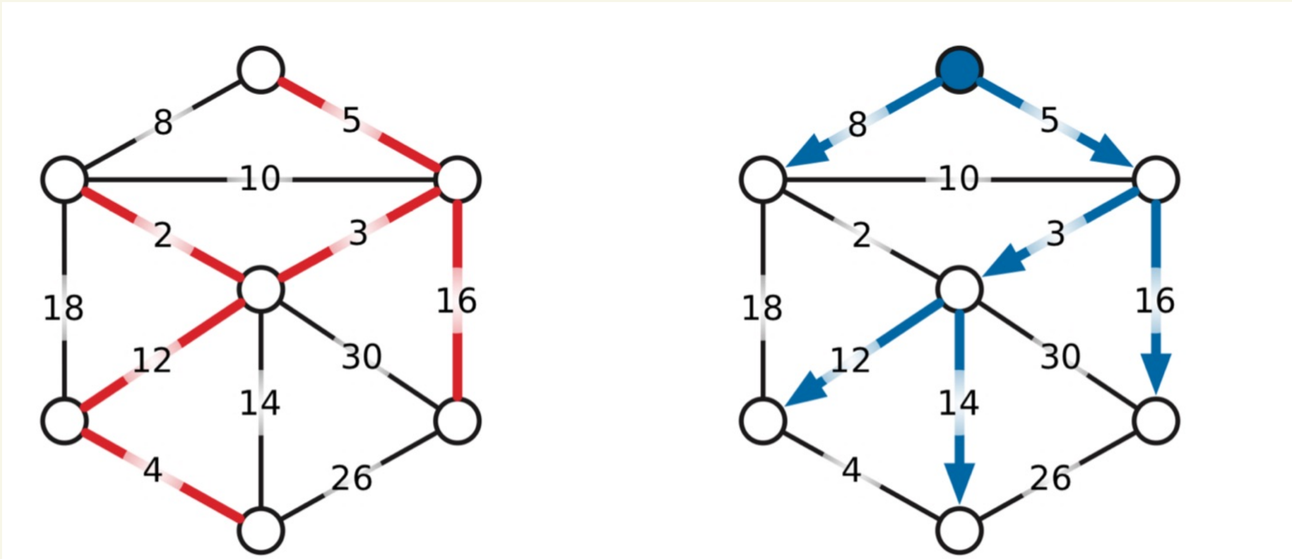


Figure 8.4. An undirected graph where shortest paths from s are unique but do not define a tree.

not a tree!

Important to realize:

MST ≠ SSSP



Why?

- MST is globally min
  - ↳ but that doesn't mean every s ⇝ t path in it is min.

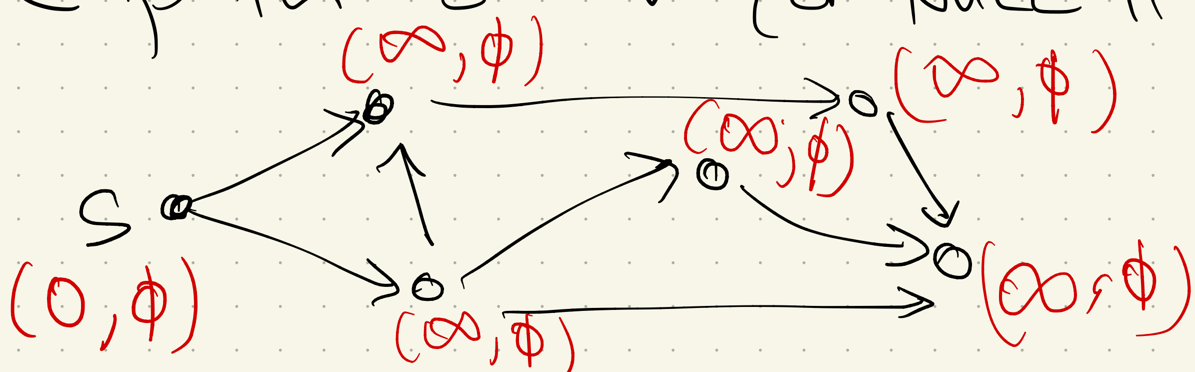
Computing a SSSP.

(Ford 1956 + Dantzig 1957)

Each vertex will store 2 values.
(Think of these as tentative shortest paths.) (dist, pred)

- dist(v) is length of tentative shortest path $s \rightsquigarrow v$
  (or ∞ if don't have an option yet)
- pred(v) is the predecessor of v on that tentative path $s \rightsquigarrow v$ (or NULL if none)

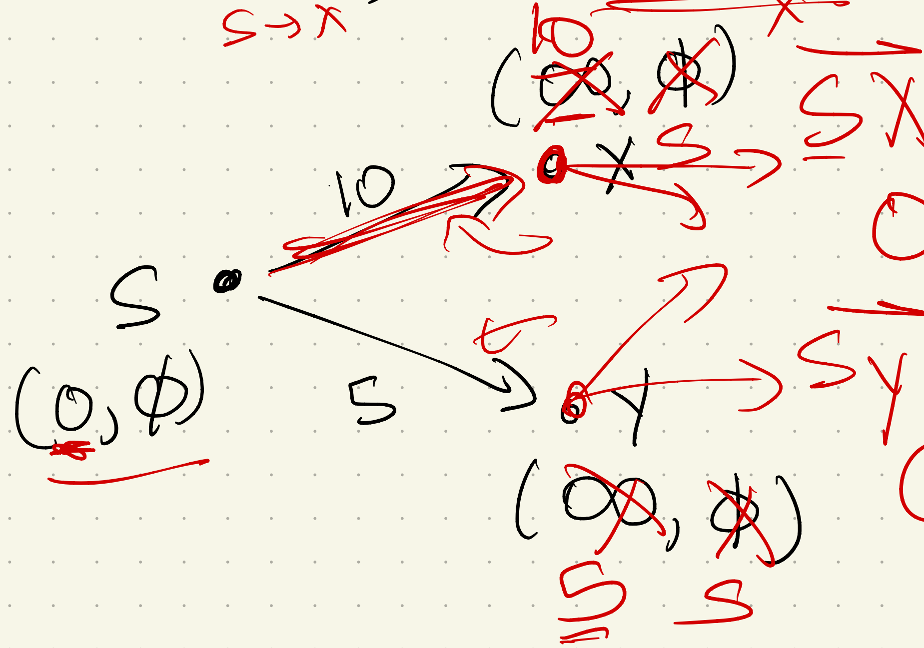
Initially:



We say an edge $\overrightarrow{uv}$ is tense if

$$\underset{s}{\mathrm{dist}(u)} + \underset{s \to x}{w(u \to v)} < \underset{x}{\mathrm{dist}(v)}$$

Initially:

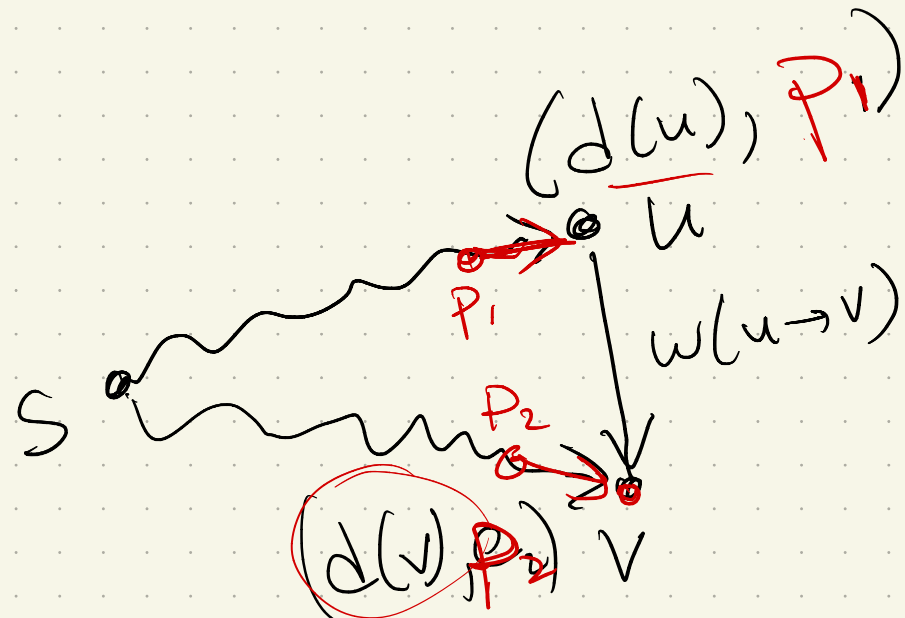


$\overrightarrow{sx}$: $0 + 10 < \infty$

$\overrightarrow{sy}$: $0 + 5 < \infty$

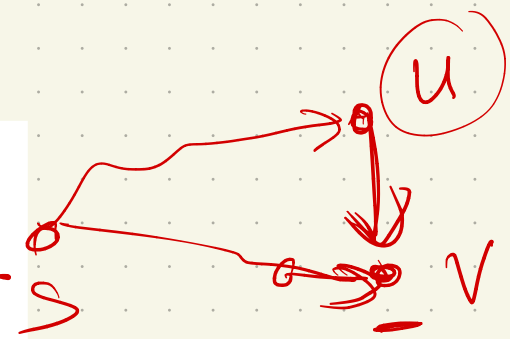
Here:

In general:



Key idea for algorithm:
Find tense edges & relax them:

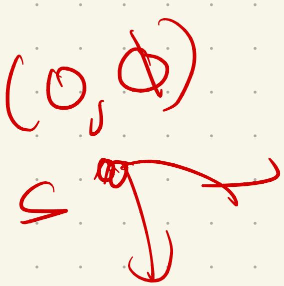
```
RELAX(u→v):
    dist(v) ← dist(u) + w(u→v)
    pred(v) ← u
```



Then:

```
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
```



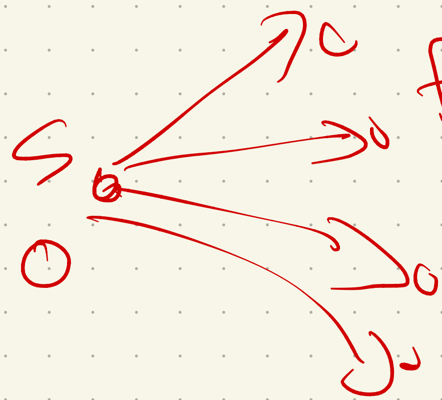
Claim: At any point in time, $\text{dist}(v)$ is either ∞ or the length of some $s \rightsquigarrow v$ walk.

Proof: Induction on while loop iterations.

Base case:

First round of loop:

$\text{dist}(s) = 0$



finite distance for s's nbrs

all other vertices are ∞.

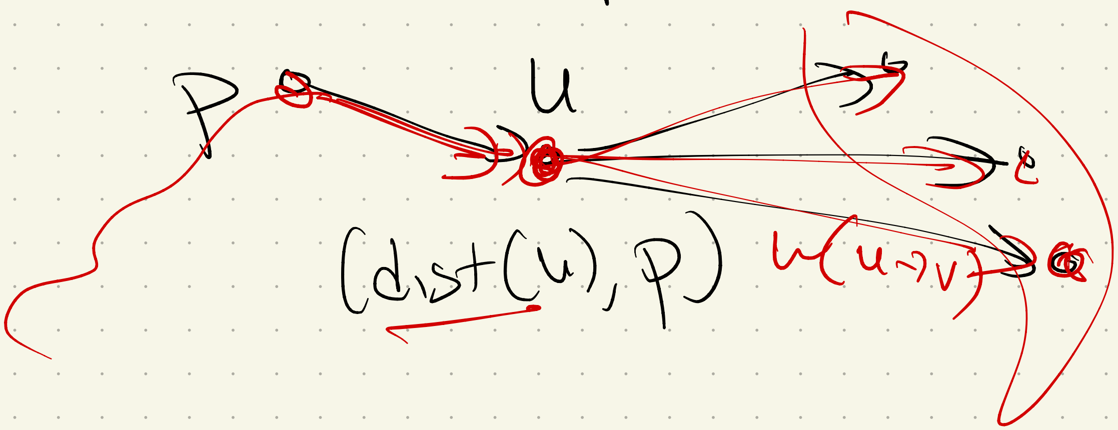
Ind hyp:

In iteration $k-1$, the claim is true

Ind Step:

In iteration k:

take out some vertex u.



If $u \to v$ is tense:

relax:

u's nbrs have finite dist

by IH: u stores some $s \rightsquigarrow u$ walk

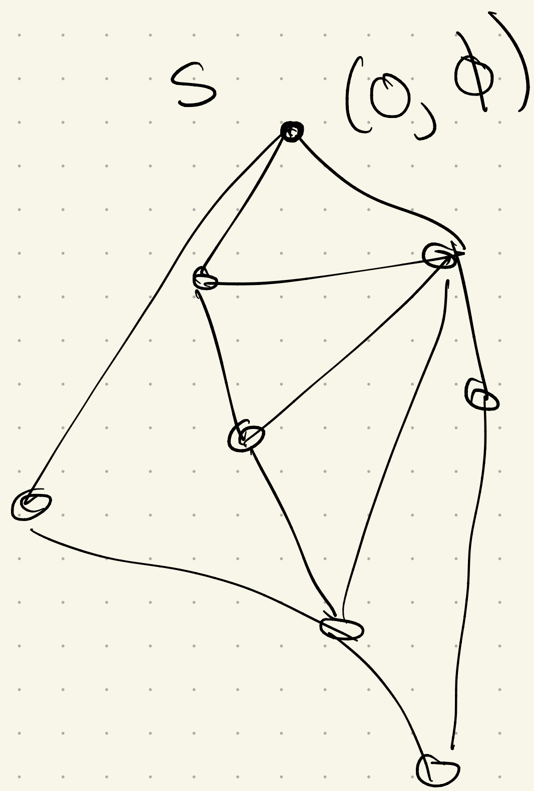
so v stores that walk + one more edge.

Warm-up: Unweighted graphs

→ use a queue

How does "tense" work?

(Hint: think BFS!)



What the heck is his token??

queue:

s ✠

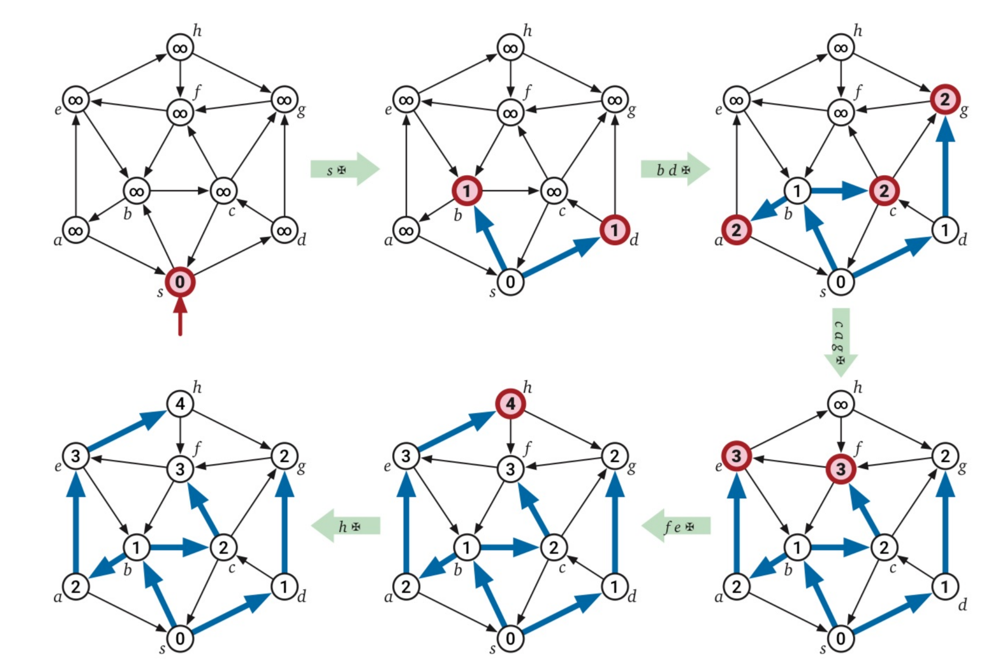


Figure 8.6. A complete run of breadth-first search in a directed graph. Vertices are pulled from the queue in the order s ✠ b d ✠ c a g ✠ f e ✠ h ✠ ✠, where ✠ is the end-of-phase token. Bold vertices are in the queue at the end of each phase. Bold edges describe the evolving shortest path tree.

```
BFSWithToken(s):
   InitSSSP(s)
   Push(s)
   Push(✠)                    ⟨⟨start the first phase⟩⟩
   while the queue contains at least one vertex
      u ← Pull()
      if u = ✠
         Push(✠)              ⟨⟨start the next phase⟩⟩
      else
         for all edges u→v
            if dist(v) > dist(u) + 1        ⟨⟨if u→v is tense⟩⟩
               dist(v) ← dist(u) + 1
               pred(v) ← u                   ⟨⟨relax u→v⟩⟩
               Push(v)
```

u =

Lemma

At the end of the ith phase (when $\mathbb{I}$ comes off the queue), for every vertex v,

either $\bullet$ $d(v) = \infty$
(not found yet)

or $\bullet$ $d(v) \leq i$
(and v is only in queue $\iff d(v) = i$).

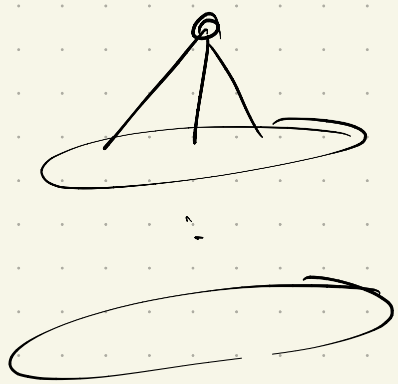
Proof: induction on phase

Base case:

Inductive Hyp: lemma holds for phases $\leq i-1$

IS: phase i: we know by the IH, when last phase ended:

BFS tree



$i-1$

$Q: \underbrace{v_k \cdots v_\ell}_{dist = i-1}, \nexists$

What now?

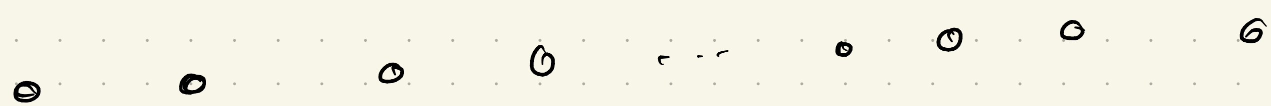
2^{nd} version: DAGs

What if directed + acyclic?

Remember: helps to have all "closer" vertices done before computing your distance.

Well, know something about DAG-orders:

↳ topological order!



edges

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)
```