Algorithms - Spring '25

MSTs
SSSPs

Recap:

- How's this view?
- Office hours:

  Thursday 12^{30} - 2pm

Minimum Spanning Trees

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

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

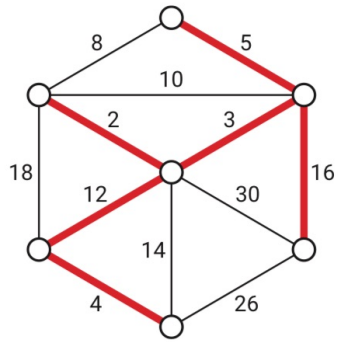


Figure 7.1. A weighted graph and its minimum spanning tree.

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.

↳ "safe" edge

So: Be greedy!

Boruvka: Build a forest, Initally V disjoint vertices.

While (F is not a tree)

Find <u>all</u> safe edges and add them to F

Runtime:

Prim: Keep one spanning sub tree.

While $|T| \neq V$:

add next safe edge

Runtime:

Kruskal's Algorithm:

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

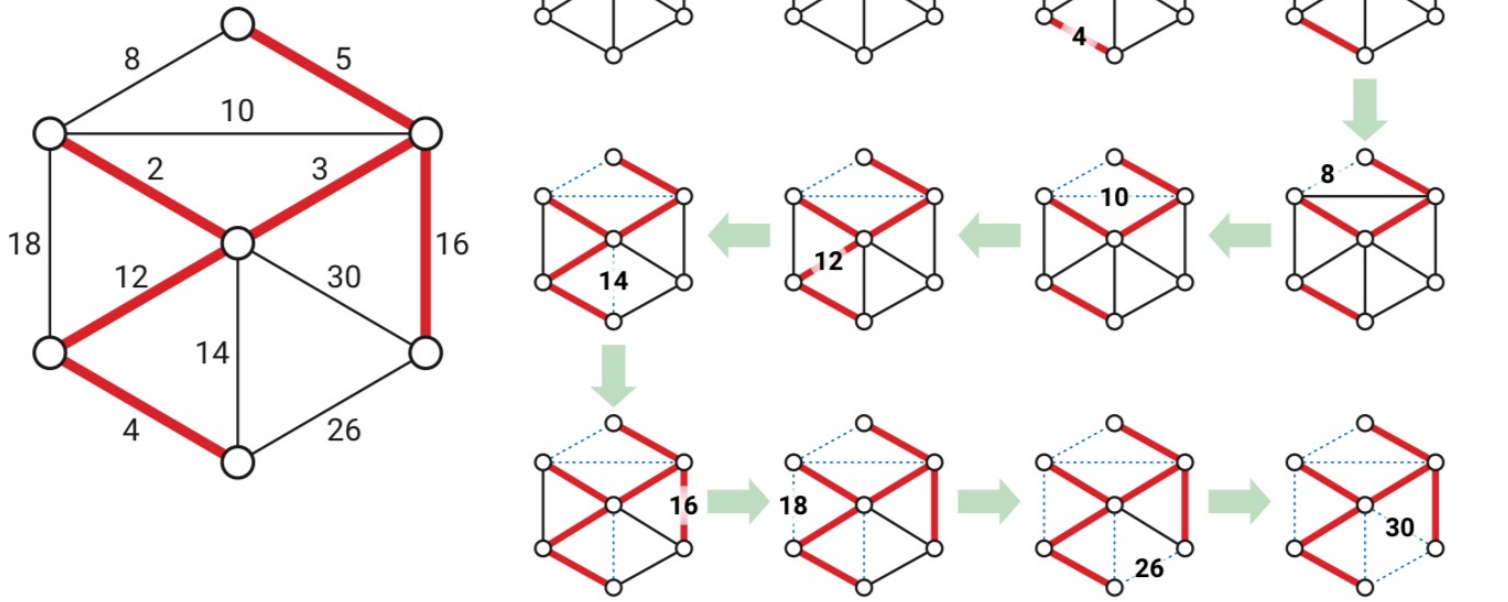


Figure 7.6. Kruskal's algorithm run on the example graph. Thick red edges are in F; thin dashed edges are useless.

How to implement?

Data structure: Union find

- 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: $O(E \log V)$

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:

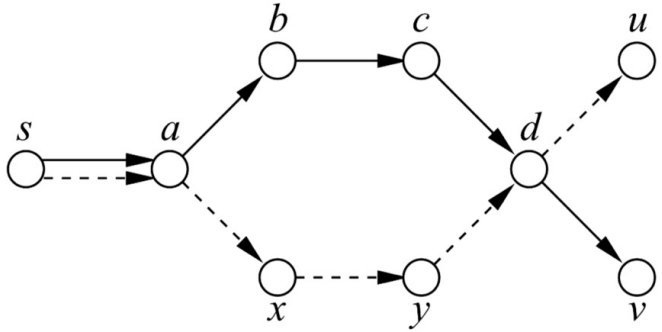
To solve this, we need to solve a more general problem:
find shortest paths from s to every vertex.

Why?

Single Source Shortest paths (SSSP)

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.

What about negative cycles or edges?

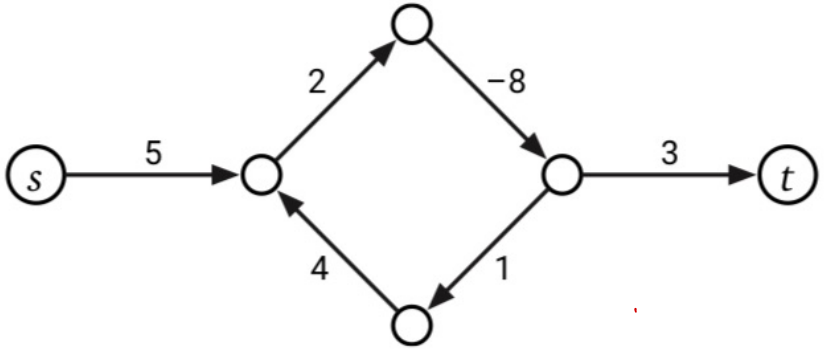
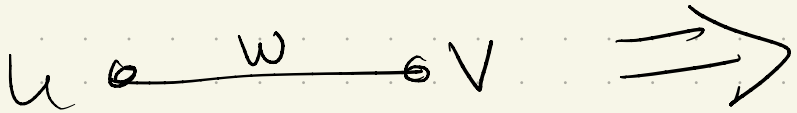


Figure 8.3. There is no shortest walk from s to t.

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



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

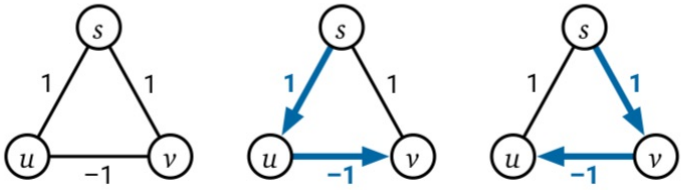
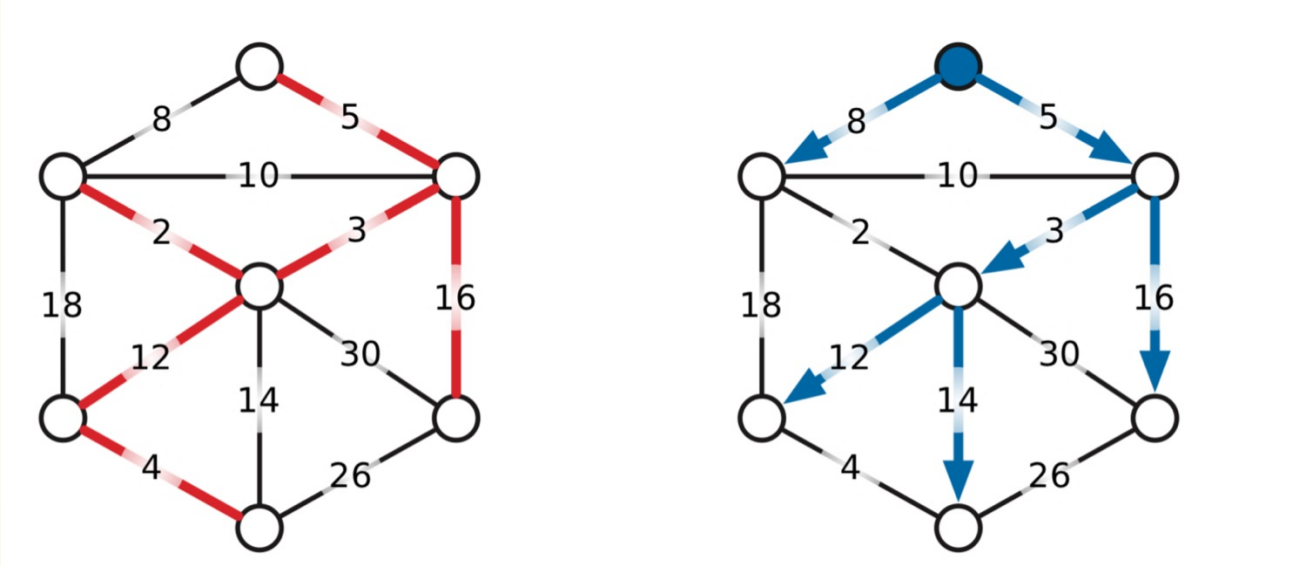


Figure 8.4. An undirected graph where shortest paths from s are unique but do not define a tree.

Important to realize:

MST ≠ SSSP



Why?

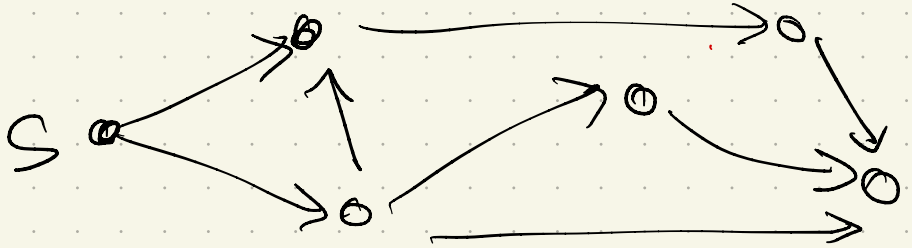
Computing a SSSP.

(Ford 1956 & Dantzig 1957)

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

- dist(v) is length of tentative shortest 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)

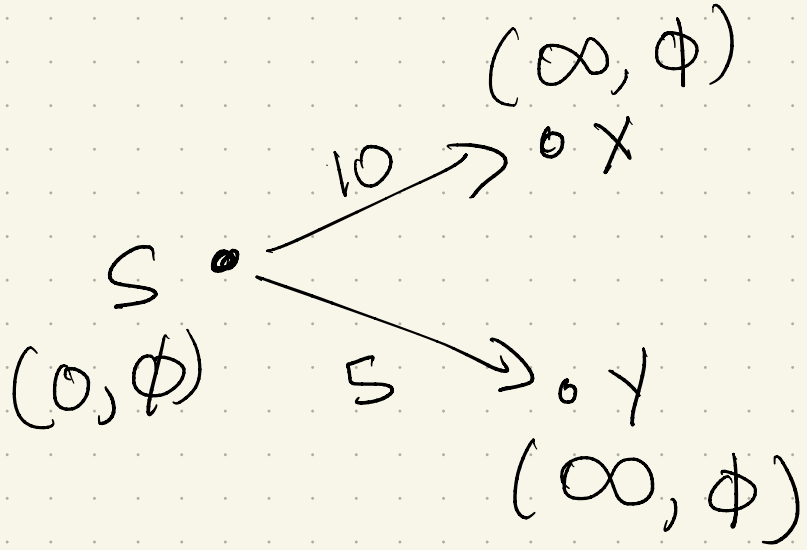
Initally:



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

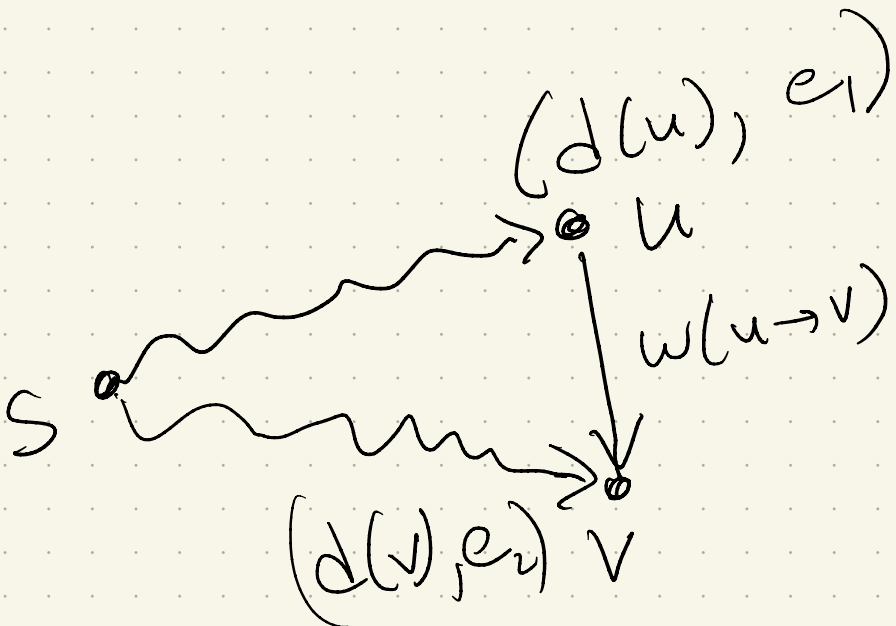
$$dist(u) + w(u \to v) < dist(v)$$

Initially:



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, dist(v) is either ∞ or the length of some $s \rightsquigarrow v$ walk.

Proof: Induction on while loop iterations.

Base case:

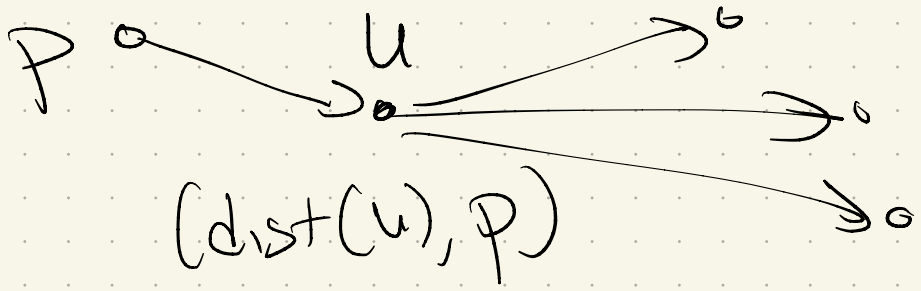
Ind hyp:

In iteration k-1, the claim is true

Ind Step:

In iteration k:

take out some vertex u.



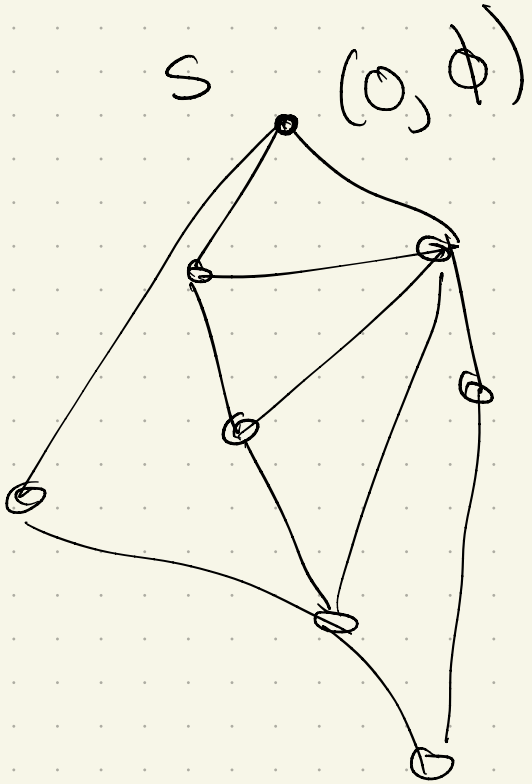
If u→v is tense:

Warm-up: Unweighted graphs

→ use a queue

How does "tense" work?

(Hint: think BFS!)



What the heck is his token??

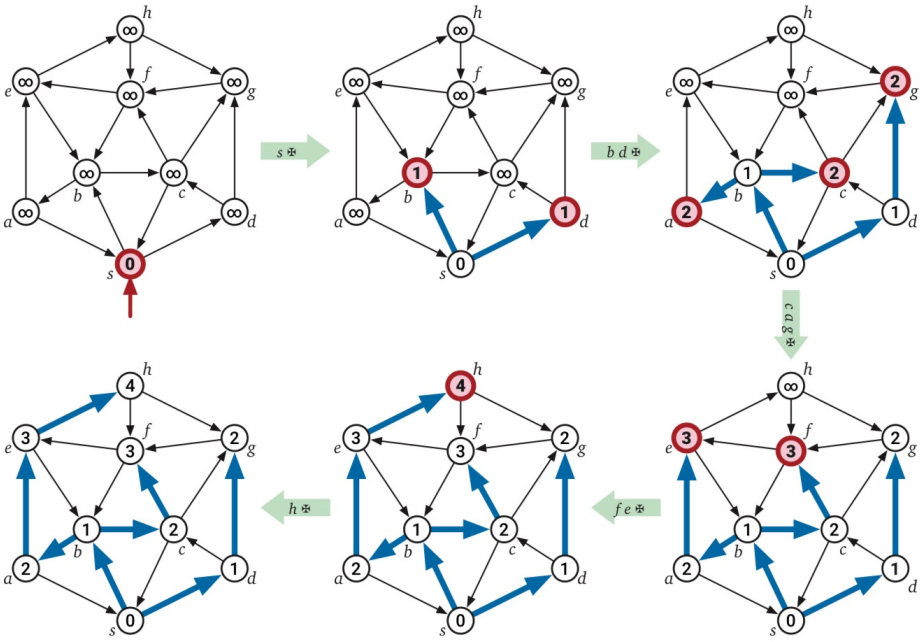


Figure 8.6. A complete run of breadth-first search in a directed graph. Vertices are pulled from the queue in the order s ✠ b d ✠ c a g ✠ f e ✠ h ✠ ✠, where ✠ is the end-of-phase token. Bold vertices are in the queue at the end of each phase. Bold edges describe the evolving shortest path tree.

queue:
s ✠

u =

```
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)
```

Lemma

At the end of the i^{th} phase (when v comes off the queue), for every vertex v, either

- $d(v) = \infty$ (not found yet)

or

- $d(v) \leq i$ (and v is only in queue $\Leftrightarrow d(v) = i$).

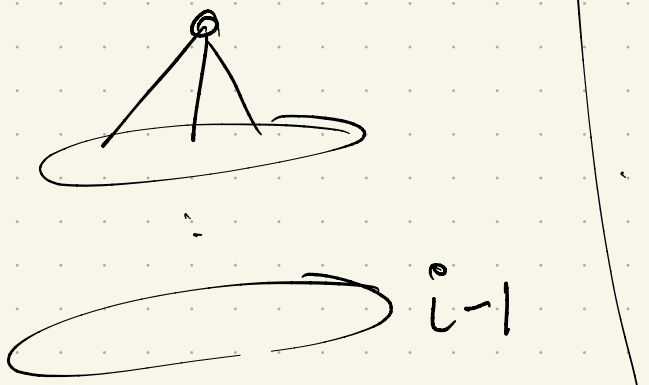
Proof: induction on phase

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



Q: $v_k \cdots v_e$

dist = $i-1$

What now?

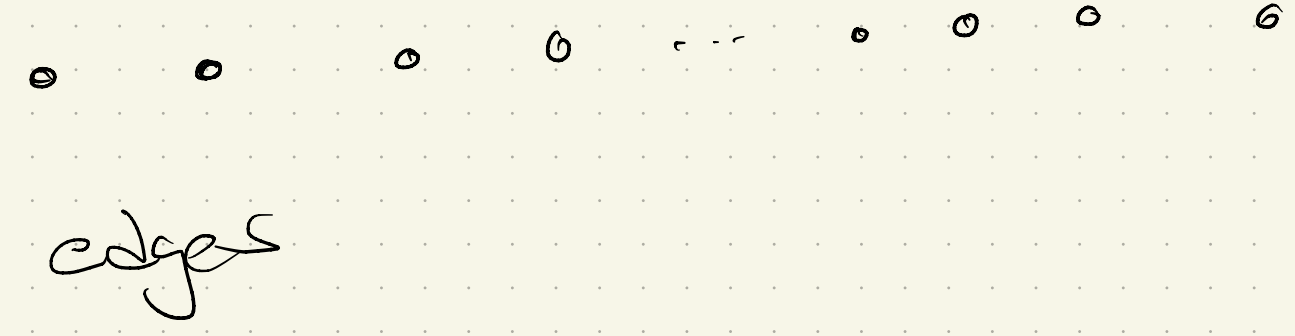
2nd version: DAGs

What if directed + acyclic?

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

Well, know something about DAG-orders:

↳ topological order!



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)
```