CS3100

MST, shortest path trees (SSSP)

Announcements

- next HW - due next Friday, oral grading
- Midterm: 2 weeks from today

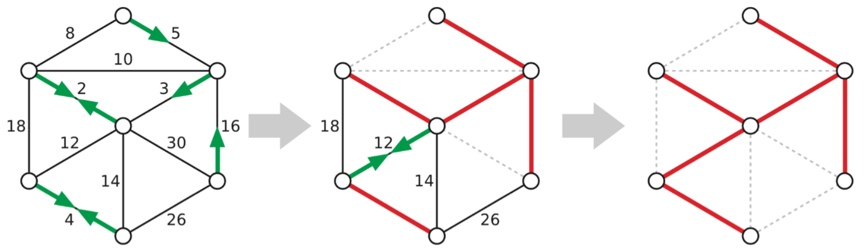
Last time: MST

Key idea: for any vertex cut S & V−S, the smallest edge between S & V−S will be in the MST.

Borůvka's (or Sollin's) algorithm:

Add all ~~such~~ safe edges for each component left in G.

Recurse.



Borůvka's algorithm run on the example graph. Thick edges are in F.
Arrows point along each component's safe edge. Dashed (gray) edges are useless.

More precisely:

- Count components of G using BFS/DFS + as we go, label each vertex w/ its component #

$O(m+n)$

While (# components > 1):

how many iterations?

compute array $S[1..n]$, where $S[i]$ = min weight edge w/ one endpoint in component i

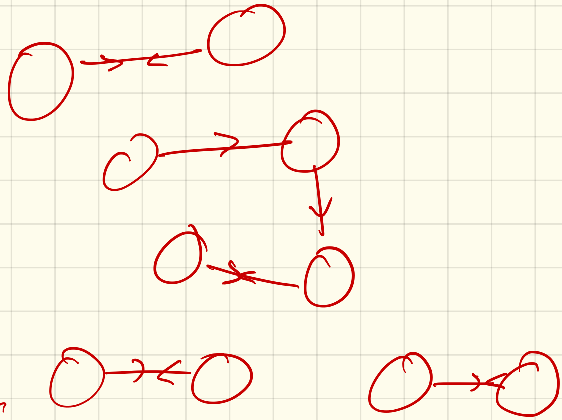
How?

- consider each edge uv:
- if endpts have same label, ignore
- if not, check if $w(uv)$ beats current $S[\text{label}(u)]$ or $S[\text{label}(v)]$

$O(m)$

Runtime:

how many iterations?



↳ worst case, # components divides by 2

$T(n) \le T(\frac{n}{2}) + 1$

(one iteration reduce # comp by half)

$\# \text{rounds} = \underline{O(\log_2 n)}$

Total: $\underline{O(m \log n)}$

Other algorithms: (Prim)

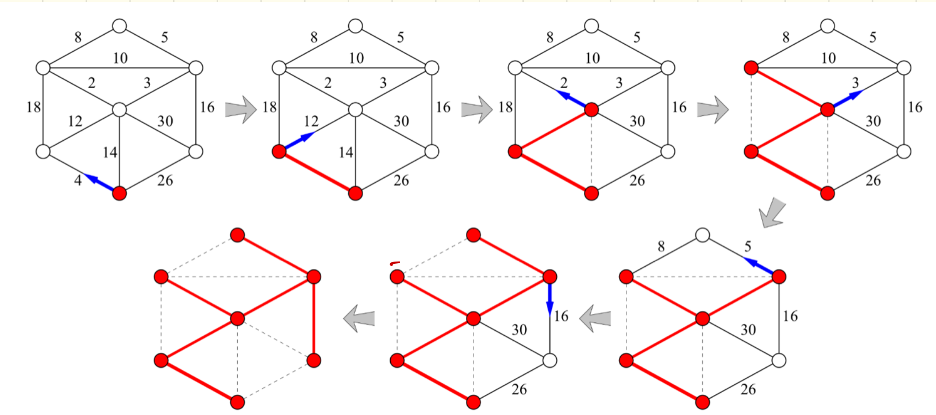
Grow a single T:

start from any vertex, & set T = vertex

while $|T| = n$:

find safe edge from T to V−T & add

(Really Jarník's from 1929)



How to implement?

heap!

Prim/Jarník:

$\log x^y = y \log x$

```
TRAVERSE(s):
    put s into the bag  heap
    while the bag is not empty
        take v from the bag  <- extractMin
        if v is unmarked
            mark v
            for each edge vw
                put w into the bag
```

(in the pseudocode: "~~the bag~~ heap" in lines 1, 2 and 7; $O(\log m)$ written beside "put w into the heap")

Runtime:

if heap is size m,

$$O(\log m) \leq O(\log n^2) = O(\log n)$$

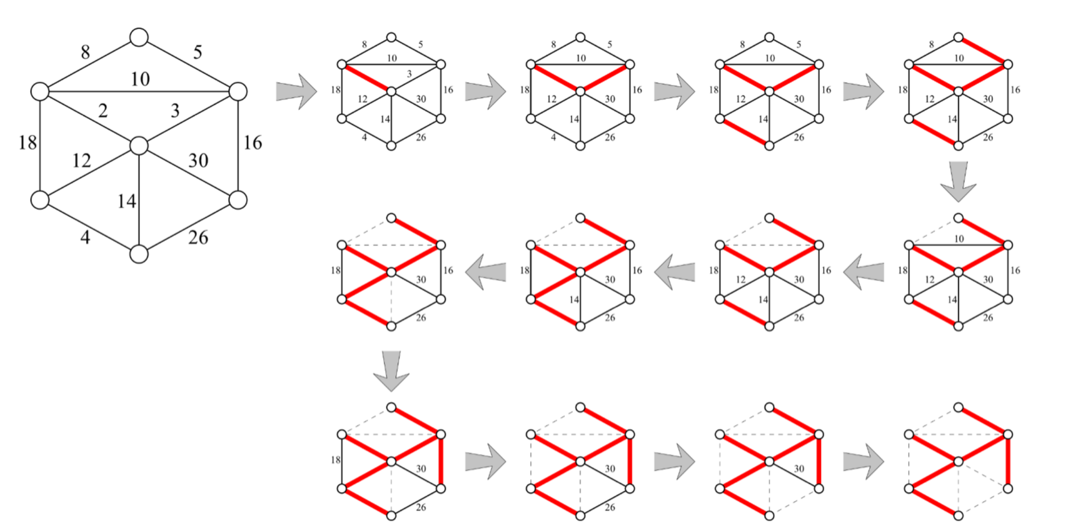
each vertex, v, gets added $d(v)$ times:

$$\sum_v d(v) \cdot \lg n = \lg n \left(\sum d(v)\right) = O(m \lg n)$$

Kruskal's algorithm (1956, motivated by Borůvka)

Scan all edges in increasing order.

If edge is safe, add it.



Kruskal's algorithm run on the example graph. Thick edges are in F. Dashed edges are useless.

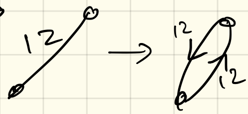
Implementation:

A bit more complex – uses Union-Find data structure (more to come...)

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:

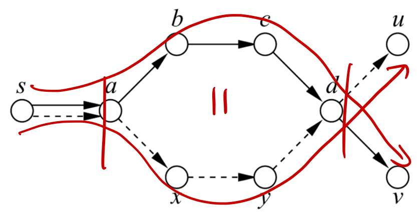
Find shortest paths from s to every other vertex.

Called the single-source shortest path tree.

SSSP

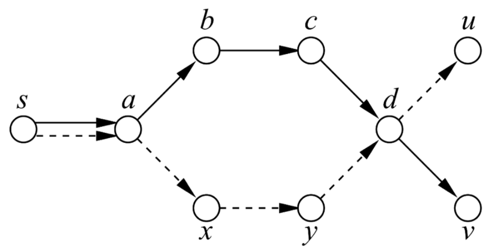
Some notes:

- 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.

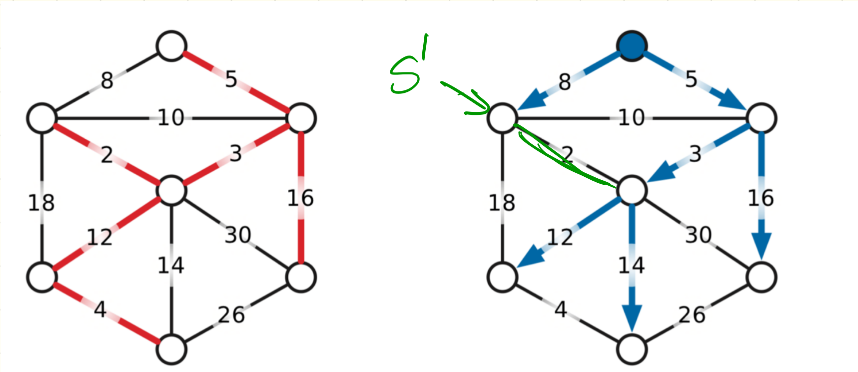
- Negative edges?



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.

Important to realize:

MST ≠ SSSP



Why? SSSP is rooted & directed

- SSSP for every vertex (these are different!)

Computing a SSSP:

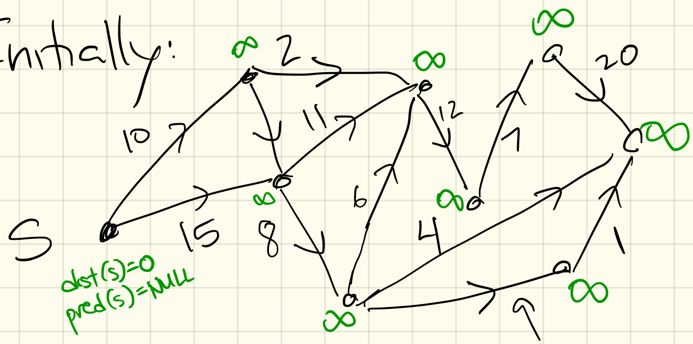
(Ford 1956 & Dantzig 1957)

Each vertex will store 2 values.

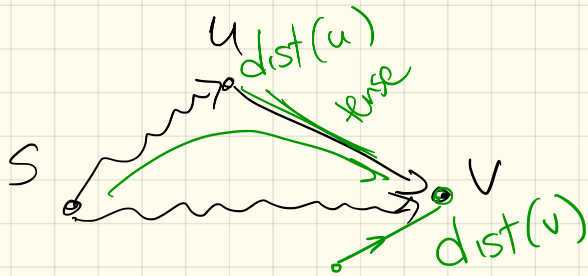
(Think of these as tentative shortest paths)

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

Initially:



We say an edge $\overrightarrow{uv}$ is <u>tense</u>
if $dist(u) + w(u \to v) < dist(v)$



If $u \to v$ is tense:
use the better path!

So, relax:

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

Algorithm:

Repeatedly find tense edges & relax them.

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

```
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"?

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 + ∞)

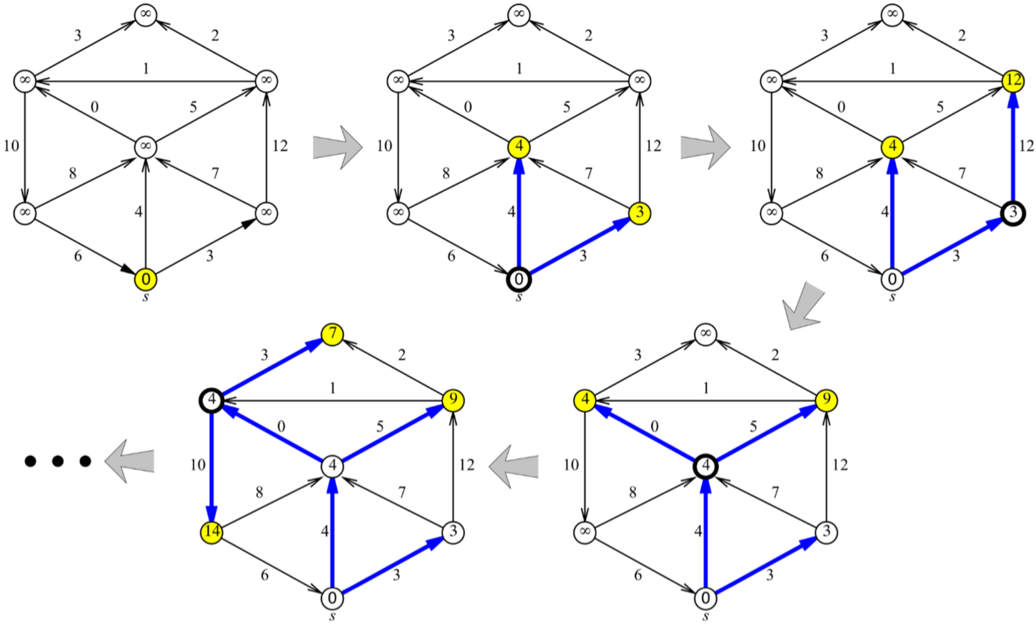
While $S \neq V$:

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 ⟶



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

Runtime:

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