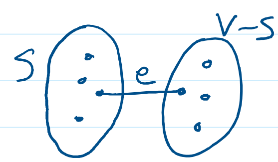
Minimum Spanning Tree (MST)

- Given an edge weighted graph $G=(V,E)$, $\omega: E \longrightarrow \mathbb{R}^+$ find a spanning tree T of minimum weight $\omega(T) = \sum_{e \in T} w(e)$
- We have seen algorithms for MST in CMPUT204.
- without loss of generality (WOLG), we can assume the graphs edge weights, $w(e)$, are all distinct; we can simply fix a tie breaking rule
- Proposition: This implies (prove it!) that the MST is unique.
- We also assume the graph is simple (no loops & parallel edges).

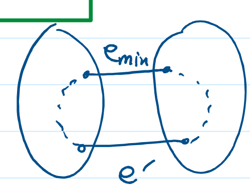
Basic rules for MST

For G, any non-trivial set $S \subset V$ ($\neq \emptyset$) defines a cut

$$\delta(S) = \{ e \in E \mid e \text{ is between } S \text{ and } V-S \}$$



cut rule: In any graph G for any cut S, the min edge $e \in \delta(S)$ belongs to MST.



Proof: Suppose not, say $e_{min} \notin T$

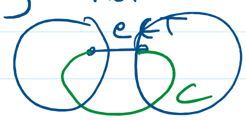
Consider adding e_{min} to T.

creates a cycle C; $\exists$ an $e' \in C$ across the cut with $w(e') > w(e_{min})$.

Remove e' from T and add e_{min}

cycle rule: For any cycle C, the heaviest edge on C cannot be in MST.

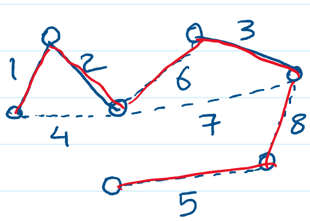
Proof: By way of contradiction, suppose heaviest edge e of C is in MST T. So $T-e$ has two components. $\exists\, e' \in C - \{e\}$ that connects these two components and so $T' = T \cup \{e'\} - \{e\}$ is a spanning tree (why?); by choice of e, $w(T') < w(T)$.



Recall classics: Kruskal / Prim / Boruvka

Kruskal: Starts by sorting the edges in increasing order $w(e_1) \le w(e_2) \le \cdots \le w(e_m)$

- takes $O(m \log m) = O(m \log n)$ $\quad (m \in O(n^2))$
- Starts from each vertex as a component
- Considering each edge e, if it connects two different components adds e to T and merges the two components.



Proof of correctness: each ignored edge by the algorithm creates a cycle; based on the order this must be the heaviest of the cycle $\longrightarrow$ cannot be in a MST.

Union-Find data structure:

A union-find is a data structure supporting these operations on input set $\{1, 2, \ldots, n\}$:

- initialize: starts the data structure by creating n disjoint sets $s_1, \ldots, s_n$ where $s_i := \{i\}$;
- union(i, j): Given index of two sets i, j, replace s_i and s_j with $s_i \cup s_j$ and the index of the new set with $\min\{i, j\}$;
- find(e): given an element $e \in [n]$, returns the index of the set

- Easy implementations of union-find can be done with $O(n)$ time for initialization and $O(\log n)$ update time. Using this data structure one can implement Kruskal's in time $O(m \log n)$

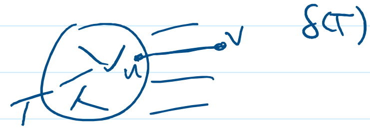
- There are efficient implementation of union-find that allow m operations (find/union) in amortized time $O(m.\alpha(m))$ where $\alpha(\cdot)$ is the inverse Ackermann function (grows even slower than $\log^*(\cdot)$ where $\log^* n$ is the # of iterated log one needs to take to get to 1. e.g. $\log^*(2^{2^2}) = 3$, $\log^*(2^{65536}) = 6$)

- Total time for kruskal's alg: $O(m \log n + m\alpha(m))$

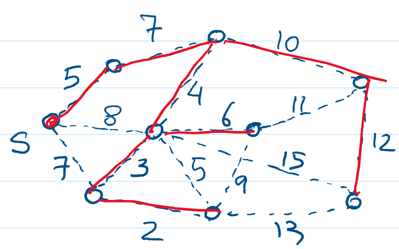
Prim: Unlike Kruskal that grows trees from each node & merges them, Prim's grows one single tree.

Prim's MST

- Start from an arbitrary vertex s;
- let T be tree with zero edges on s alone.
- repeat n-1 times:
    - Pick the cheapest edge between T and V-T, say e=(u,v) u∈T
    - add e to T
- return T



- We can use a priority Queue to implement.



Prim's Algorithm using PQ:

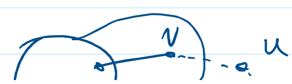
(i) Start from a node s; Let T=(s,{}).
Let Q be a PQ with keys being vertices and key-values being each edge(v) and its weight.

(ii) For each v ∈ N (s) set edge(v) = (s,v) and run Q.insert(v, w(edge(v))).

(iii) Repeat n-1 times:

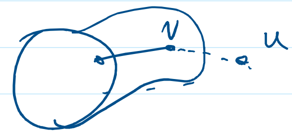
(a) Pick v with smallest value in Q (Q.extract-min and edge(v) be the edge of v). Add vertex v and the edge edge(v) to T .

(b) For each u ∈ N (v), if u∉T update Q.decrease-key(u, w(uv)) (if u is not in Q, we instead run Q.insert(u, w(uv))).



w(uv))).

(iv) Return T as the MST of G



Proof of correctness: first note that it adds n-1 edges & does not create a cycle ⟶ returns a spanning tree

To prove it returns a MST we use the cut-rule: any time we add e to T, the set of vertices in T are connected & all edges between them & V-T are added to Q; so Q contains all $\delta(T)$ and e is the smallest across the cut.

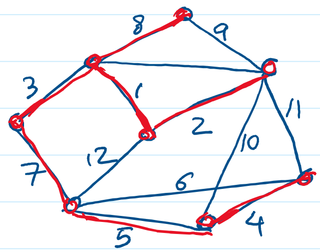
Runtime: - O(n) insert operations (iterations of loop)
- O(m) decrease-key (twice for each edge)
- O(n) extract-min to add the next edge.
- Using min-heap implementation of PQ : O(m log n)

Boruvka's Algorithm:

- one of the oldest MST and is somewhat in between Kruskal's & Prim's
- Starts from each node v being a component.
- At each iteration finds cheapest edge connecting a component to another and adds them all; merge the components that get connected.

Boruvka's MST

- Start from $S(v) = \{v\}$ for each node v and $T = \phi$
- while there are more than one sets do:
  - for each set S find the cheapest edge $e \in \delta(S)$; call it e_S
  - Add all edges e_S to T and merge all sets that these edges run between
- return T



Correctness: Easy to see it returns a spanning tree (why?) to show it is MST, we can use the cut rule: any edge selected by the algorithm is a min-cost edge going out of a component and hence is in MST.

Time: The number of rounds is $O(\log n)$ as the number of components goes down by a factor of two each time. In each round we spend $O(m)$ time $\longrightarrow$ total $O(m \log n)$

Advanced MST Algorithms:

* Fredman and Tarjan's $O(m \log^* n)$ algorithm

Fibonacci heaps: Is an advance implementation of P.Q's

Fibonacci heaps: Is an advance implementation of P.Q's using heaps. It supports:

- $O(1)$ amortized time for insert(·) and decreas_key(·)
- $O(\log n)$ amortized time for extract_min(·) where n is the maximum size of heap at any time.

(See CLRS for details)

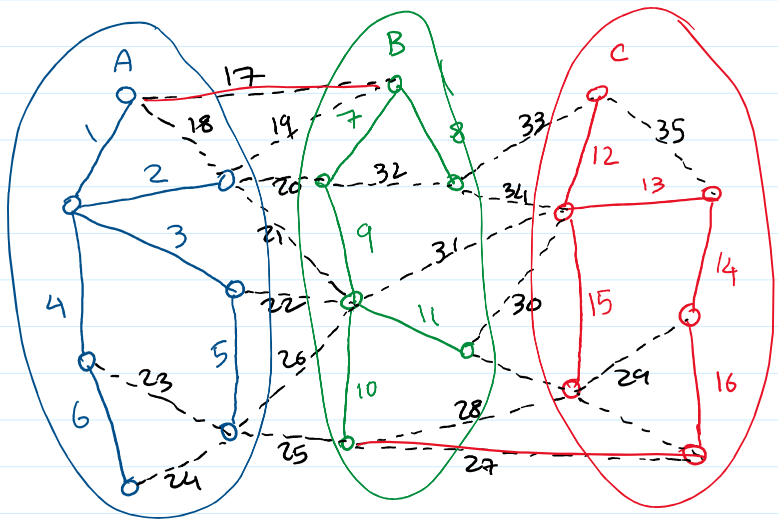
So using F.H. in Prim's alg, since we:

- $O(n)$ insert(·) operations
- $O(m)$ decrease_key(·) operations
- $O(n)$ iterations & extract_min(·) operations

So total time becomes $O(m + n\log n)$

Idea of improved time: In Prim's algorithm, each iteration we have a P.Q of edges going out of the current tree. What if we ensure the current tree has small boundary & use a Fibonacci heap?

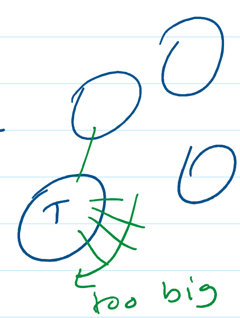
- run prim's as long as the boundary (and so the heap size) is bounded by a constant.
- once it becomes too big start growing (Prim) tree from another node.
- once every vertex belongs to one of these trees, say we have a forest with components $C_1, C_2, C_3, \dots$ We contract them each into a single node and recurse on this new graph.



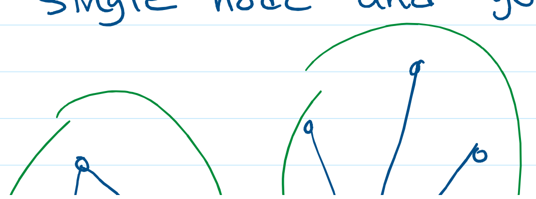
Fredman–Tarjan MST

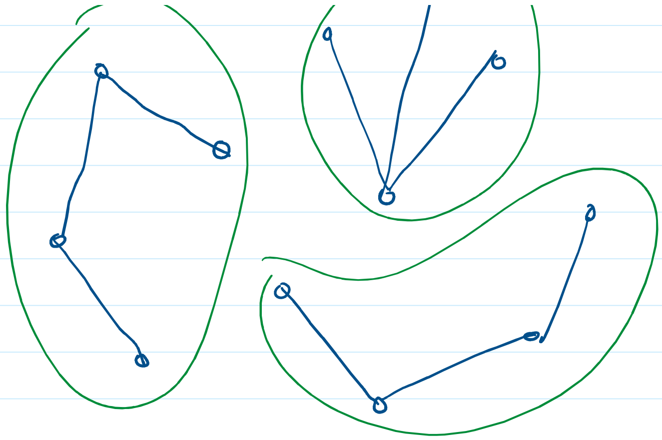
- The algorithm runs in rounds; in round i we have graph G_i with n_i nodes and m_i edges, obtained by contracting some trees in previous round. $G_1 = G$. We also have threshold $k_i = 2^{2m/n_i}$
- in each round do the following:

1) Each vertex $v \in V_i$ is unmarked.
2) Pick an unmarked (→ not belonging to any tree in this round) vertex and run Prim to grow a tree T. Keep track of lightest edges going out of T to vertices in $N(T) = \{u \mid \exists v \in T \text{ and } uv \in E\}$
3) if the size of $N(T) \geq k_i$ or if added an edge to a marked vertex stop, mark all nodes in T, and go to step 2.



4) If no unmarked node left contract each tree into a single node and go to the next round





Proof: First note that it returns a Tree (why?)
To see it returns a MST follows from using cut rule (same as Prim's algorithm)

Runtime details: Note that time for each round is $O(m + n_i \log k_i)$:

- each edge is considered twice
- each time we grow a tree # of components decreases; so $O(n)$ times
- the P.Q operations take $O(\log k_i)$

Observation 1: For every component C: $\sum_{v \in C} \deg(v) \geq k_i$

Proof: note that when v gets added to a component C at most $\deg(v)$ edges are added to the heap.
So size of the heap is bounded by sum of degrees of vertices added. So when the size of heap i.e. $N(T) \geq k_i$ we must have $\sum_{v \in C} \deg(v) \geq k_i$.

So if $C_1, \dots, C_\ell$ are the trees/components at the end of current round then:

$$\sum_{i=1}^{\ell} \sum_{v \in C_i} \deg(v) \geq \ell \cdot k_i$$

assuming we have m_i edges

$$2m_i = \sum_{v} \deg(v) \geq \ell \cdot k_i \longrightarrow \overbrace{\ell}^{=n_{i+1}} \leq \frac{2m_i}{k_i} \leq \frac{2m}{k_i}$$

Recall $k_i = 2^{2m/n_i} \longrightarrow k_{i+1} = 2^{\frac{2m}{n_{i+1}}} \longrightarrow \frac{2m}{n_{i+1}} = \log k_{i+1}$

Thus $n_{i+1} \leq \frac{2m_i}{k_i} \longrightarrow k_i \leq \frac{2m_i}{n_{i+1}} \leq \lg k_{i+1}$ $\longrightarrow k_{i+1} \geq 2^{k_i}$

So the threshold k_i exponentiates in each round

$\longrightarrow$ # of rounds is bounded by $\log^* n$

$\longrightarrow$ total running time is $O(m \log^* n)$.

Linear time MST

Next we see a randomized $O(m+n)$ time MST algorithm by Karger-Klein-Tarjan (1995), called KKT.

Definition: Suppose $F \subseteq G$ is a forest. An edge $e \in E$ is *F-heavy* if e creates a cycle in $F \cup \{e\}$ and it is the heaviest edge in that cycle. Otherwise e is *F-light*

observation:

i) e is F-light $\Longleftrightarrow$ $e \in MST(F \cup \{e\})$

ii) if T is an MST then e is T-light $\Longleftrightarrow$ $e \in T$

iii) For any forest F, the F-light edges contain MST of G.

iii) For any forest F, the F-light edges contain MST of G.
i.e. for any F-heay edge e, $MST(G-e) = MST(G)$.

- How to use this? if F is a forest, we can discard F-heavy edges (from G). Goal: find a forest with many F-heavy edges (so F is close to an MST!) Then we can recurse on the remaining edges.

issues: 1) how to find good forest F?
2) how to quickly determine/classify F-heavy edges?