Moving away from scheduling: problems in Graphs

Problem: Minimum weight s-t path in a graph (with non-negative edge weights).

Given: A directed graph $G = (V, E)$ and a designated start node s. Assuming that s has a path to all other nodes in G and each edge has a weight $\ell(e) \geq 0$

Find: The minimum weight path P_{sv} where $\ell(P_{sv}) = \sum_{e \in P_{sv}} \ell(e)$ from s to every other node v.

Note: Can easily be made "undirected" by replacing each edge (u,v) in G with a pair (u,v), (v,u) where $\ell((u,v)) = \ell((v,u))$

Alg (due to Dijkstra 1959)

Idea: Start a "traversal" at s, and keep a set S of explored nodes where we know the shortest path length. In each iteration of the algo, add a node adjacent to S to the explored set, for which the distance to this node is smallest. Update the distance to this node, and add it to S.

Alg: Dijkstra (G, s):

Let S be the set of explored nodes + and keep $d(u)$ for $u \in S$

Initialize $S = \{s\}$, $d(s) = 0$

While $V \neq S$:

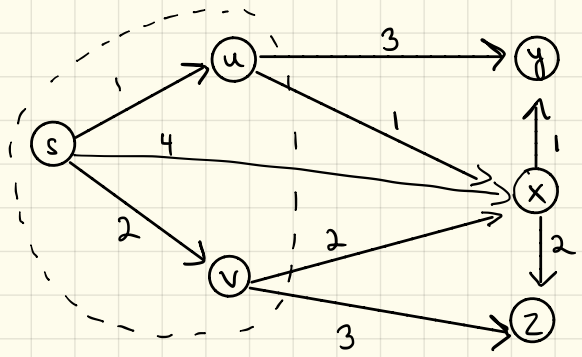
- Select $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v): u \in S} d(u) + l((u,v))$ is as small as possible
- add v to S

End while

Note: This algorithm (as written) gives only the s-v distance for all v. It can be trivially modified to produce the actual paths as follows:

- When v is first added to S via (u,v), store a "backpointer" v→u. Each node remembers the edge it used to join S. Recursively, this remembers the shortest paths from s.

E.g.



Dijkstra's after running for 2 iterations of the while loop.

First, add u, set d(u)=1. Then add v, set d(v)=2. In the next step, x will be added. What if we attempted to add y or z instead?

The distances would be wrong! We only know the provably correct shortest path to the closest node to S at any time.

Proof of correctness uses a "stays ahead" argument similar to IS.

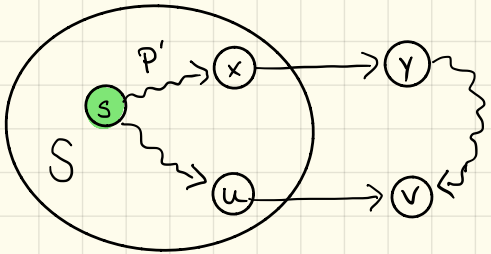
(4.14) Proposition: Consider the set S produced by Dijkstra's alg at any point in execution. For every $u \in S$, the path P_{su} is a shortest s-u path.

Proof:

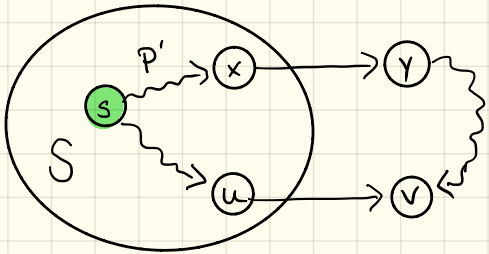
Base case: $S = \{s\}$, $d(s) = 0$ is true

IH: Assume this is true for $|S| = k$

IS: Consider growing $|S|$ to $k+1$ by adding some node v via edge (u,v) - the final edge in P_{sv}. Consider any other s-v path P, we show that it must be at least as long as P_{sv}. To reach v, P must leave S somewhere, let y be the first node on P not in S, and let $x \in S$ be the node just before y. The situation is as below.



Proof continued:



We know, by IH that P_{sx} is the shortest s-x path with length $d(x)$, so $\ell(P') \geq \ell(P_{sx}) = d(x)$. Thus, the path P to y has length $\ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq d'(y)$, and the full path to v is at least as long as this subpath. But, Dijkstra's gives us that $d'(y) \geq d'(v) = \ell(P_{sv})$ - since it chooses the smallest $d'(v)$ — so that we have $\ell(P) \geq \ell(P') + \ell(x,y) \geq \ell(P_{sv})$ ∎

Problem: MST (Minimum Spanning Tree)

Given: An undirected graph $G = (V, E)$ and a weight function $d(\{u,v\})$ that provides a non-negative weight for each edge.

Find: The subgraph T that connects all vertices (spans the graph) and minimizes

$$\text{cost}(T) = \sum_{\{u,v\} \in T} d(\{u,v\})$$

(4.16) Claim: T will be a <u>tree</u>. (Why?)

<u>Proof</u>: Suppose T contains some cycle C with edge $e \in C$.

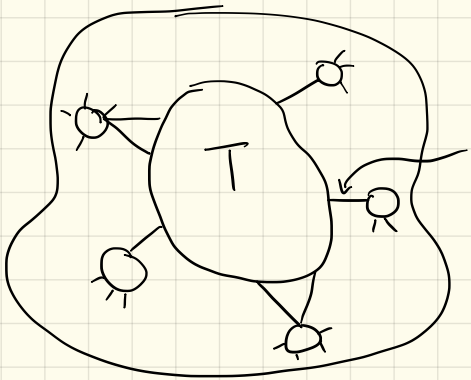
Then $(V, T - \{e\})$ is still connected, and has less cost (or at least cost no greater) than T. $\rightarrow\leftarrow$

Sketch of Prim's algorithm:

(1) Given $G = (V, E)$ and vertex $s \in V$, Let T be a "tree" containing only s.

(2) Repeat the following $|V|-1$ times:

- Add the lowest cost edge $\{u, v\}$ such that $u \in T$ and $v \notin T$ to T.



we examine these "appendage" edges, and pick the cheapest one.

Note the similarity to Dijkstra's

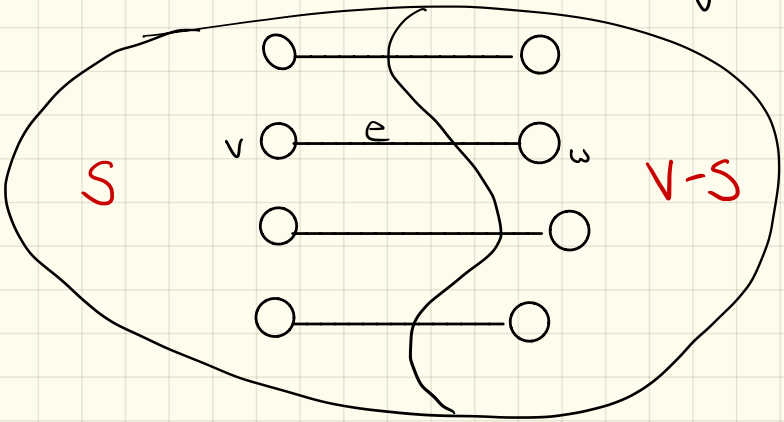
```
Alg Prims (G, s):
    Parent = {}
    For u ∈ V:  distToT[u] = ∞
    u = s
    While u ≠ null:
        distToT[u] = -∞
        for v ∈ Neighbors(u):
            if d({u,v}) < distToT[v]:
                distToT[v] = d({u,v})
                parent[v] = u
            end if
        end for
        u = closestVertex(distToT)
    end while

    return parent
```

* Assumption: Edge weights are distinct. This restriction can be lifted by applying a tiny "perturbation" to the weights of each edge (so long as the perturbation retains the order of non-ties).

To prove correctness of Prim's, we establish the

Cut Property:

(4.17) Theorem (MST cut property). Let S be a subset of nodes with $|S| \geq 1$ and $|S| < |V|$. Every MST contains the edge $e = \{u, v\}$ with $v \in S$ and $u \in V - S$ that has minimum weight.

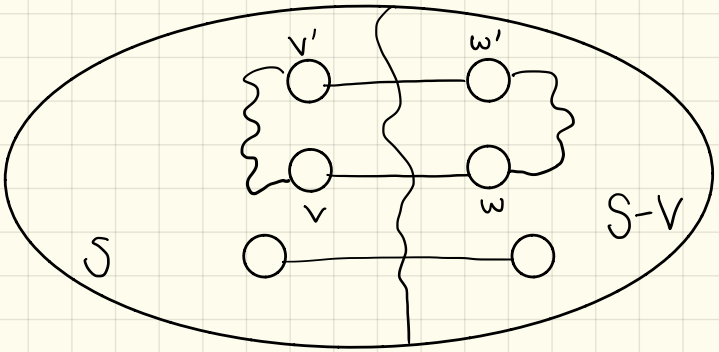


- a cut is just a pair $(S, V-S)$ that partitions or "cuts" the graph into two parts.

Proof: Suppose T does not contain e. Since T is connected, it must contain a path P between v and w, and P must contain some edge f that "crosses" the cut.

But the subgraph $T' = (T - \{f\}) \cup \{e\}$ has lower weight than T.

T' is still connected since any path using P in T can now be "re-routed" using e. T' is acyclic because $T' \cup \{f\}$ is the only cycle, which is eliminated by removing f. Finally, since $d(e) < d(f)$, we have $\text{cost}(T') < \text{cost}(T)$ ∎



(4.18) Claim: Prim's algorithm produces an MST of G.

Proof: At any point $T = (V_T, E_T)$ is a subgraph that is a tree.

T grows by 1 vertex and 1 edge after each iteration, so it stops after $|V_G| - 1$ iterations and at that point, T will be a spanning tree.

The pair $(V_T, V_G - V_T)$ is a cut of G. By the cut property (4.17) the MST contains the lowest cost edge crossing this cut. This is <u>exactly</u> the edge added to T by Prim's, so Prims only adds edges that must be in the MST ∎

Speeding up Prim's algo:

Naive implementation of Prim's is $O(mn)$

Idea: for each node $v \in V - S$, maintain an attachment cost

$$a(v) = \min_{e = \{u,v\} : u \in S} c_e$$

If we keep this in a priority queue, we can select the next node to add via the ExtractMin() operation in $O(\lg n)$ time. Then, we must update the attachment costs using ChangeKey() in $O(\lg n)$.

There are $n-1$ iterations where we do ExtractMin() and we do ChangeKey() at most once per edge $\Rightarrow$ Total running time of prim's is $O(m \lg n)$.

Ch 2 2.5

Recall: ExtractMin(·) operation of a PQ returns the element with the smallest key

ChangeKey(·) allows altering the key associated with an element

Another algorithm to find MST: Kruskal's algorithm.

Sketch of algorithm: Add edges in increasing order of weight, skipping any edge that would create a cycle. Note: cycles happen when one adds an edge to an already connected component.

(4.18)

Theorem: Kruskal's alg. produces an MST of G

Lemm 1: Every edge added by Kruskal's algorithm is justified by the cut property.

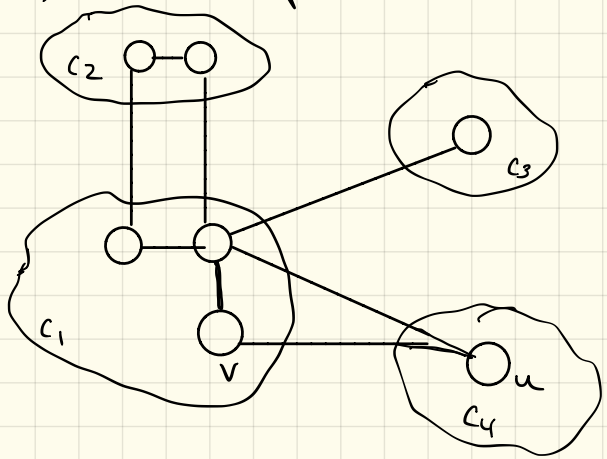
Proof: Consider $e=(v,w)$ added by kruskal's. Let S be the set connected to v before e is added. We have $v \in S$ and $w \in V-S$ so that adding e creates no cycle. Further, no edge from S to $V-S$ has been seen yet (or Kruskal's would have added it). So, e is the cheapest edge in the $(S, V-S)$ cut, and is in the MST. ∎

Lemma 2: The output of Kruskal's is a spanning tree.

Proof: Let (V, T) be the Kruskal's output. By design (V, T) contains no cycles. If (V, T) was not connected, there would be some $S \neq V$ with an edge from S to $V - S$. But, in that case, Kruskal's wouldn't have stopped. Thus, (V, T) is a connected graph with no cycles and is a spanning tree of G. ∎

By Lemma 1 + 2, Kruskal's produces an MST of G.

Diagram



Each C_i is a different component, and edges between them have not been added. Kruskal's picks the cheapest merge and adds the edge.

Efficient implementation of Kruskal's

(1) Sort the edges by cost: this is $O(m \lg m)$, but since $m \leq n^2$ we have $\lg m \leq \lg n^2 \leq 2 \lg n$ so that $O(m \lg m) = O(m \lg n)$

$$\Rightarrow O(m \lg n)$$

(2) Use Union-Find to maintain connected components of (V, T) as we add edges. To merge 2 components, compute Find(u) and Find(v). If Find(u) = Find(v), then u and v are in the same connected component and we cannot merge. Otherwise, Union(Find(u), Find(v)) is the new expanded component.

At most $2m$ "Find" operations ($O(\lg n)$ each)
At most $n-1$ "Union" operations ($O(1)$ each) $= O(n)$ total

$$\Rightarrow m \lg n + 2m \lg n + n = O(m \lg n)$$

* assume $m \geq n$ if G is connected and not already a tree.

Interlude: Union-Find (A data structure for maintaining disjoint sets efficiently).

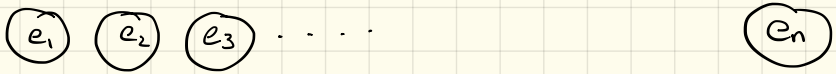
Idea: We want to maintain a collection of disjoint subsets so that we can do the following efficiently.

(1) For some element e, determine the subset to which e belongs

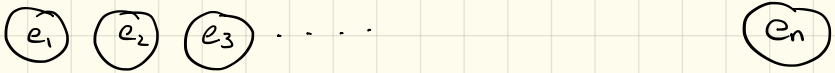
(2) Given two subsets S_1 and S_2, create a new subset equal to their union.

S = our initial set

Start with every $e \in S$ in its own set

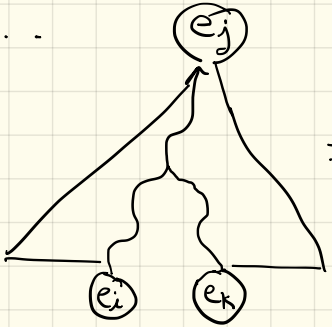


define the operation Find(e_i) that returns the "representative" of e_i. Initially, this is e_i itself.



now, imagine we want to union 2 elements (e_i and e_j), we do this by simply having e_i point to e_j (or vice versa).

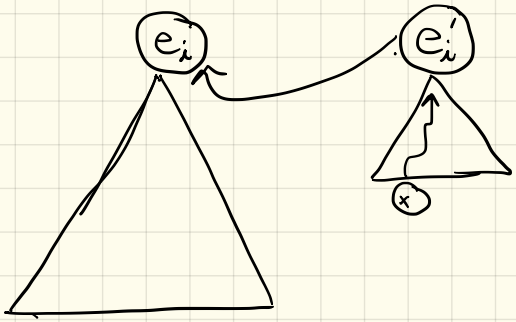
now, e_i's "representative" is simply obtained by walking the pointer until we reach the "root" of it's component. Consider the following scenario.



$$\text{Find}(e_i) = \text{Find}(e_k) = \text{Find}(e_j) = e_j$$

How do we join 2 components efficiently?

Consider



where we wish to union the component with representative e_i with that with representative e_i'.

We simply point e_i' at e_i. Now, all elements that previously had e_i' as their representative (including e_i' itself) now have e_i as their representative.

Was there a motivation for pointing e_i' at e_i instead of vice versa?

Yes!

When we union 2 components, we always point the smaller one at the larger one.

This increases the number of parents we must traverse to find the representative for nodes in the smaller subtree, but not those originally in the larger subtree. If trees are the same height, the number of traversals increases in that tree by one.

This rule (union by size or union by rank) guarantees that a series of m operations will have complexity of $O(m \lg n)$. This is because it limits the tree heights. The only time the height of the large tree grows is when both trees have the same height, but that can happen at most $\lg n$ times.

When tree heights are $O(\lg n)$, Find() takes $\lg n$ time.

Can also use path-compression to make operations amortized $O(\alpha(n))$

$\alpha()$ is the inverse Ackermann function — grows so slowly that $\alpha(n)$ is essentially constant for any n we will encounter.