

Tuesday, October 4

[1] Welcome!

[2] HW 2/3 back as soon as I can!

[3] Try to have a topic idea by the end of the week/early next week

[4] Kruskal's HW Question

↳ add $x = []$ to generate_weight_list right after for loop.

[5] Questions

[6] Trees!

[7] Outro

↳ small work: finish your LP summary and look@ topics.

Welcome back to another episode of DiscOp! Today we're really putting the discrete in discrete optimization as we move from learning about linear programming to applying it to ask questions about graph structures. As we go through graph optimization problems, we'll often describe them with linear programs to start but then develop more specialized tools for solving the question based on the structure we're concerned about.

Before we start, let's recall what we know about graphs.

Review: Define as many of the following words from graph theory as you can.

- graph (two set definition)
- vertex, vertices, vertex set, order
- edge, edges, edge set, size
- degree
- path
- cycle
- connected
- tree
- leaf
- spanning tree
- bipartite
- Eulerian
- Hamiltonian
- coloring
- chromatic number
- weighted graph
- directed graph
- arc, directed edge
- directed cycle
- matching
- vertex cover
- cut
- flow

We'll be talking about the majority of these, but we'll start today with trees.

Definition (tree, forest) A tree is a graph which is acyclic and connected. An acyclic (and not necessarily connected) graph is a forest.

Claim: Every tree on n vertices has $n - 1$ edges

have you considered induction??

- base: $n=1$. K_1 , 0 edges, ✓
- hyp: s/p a tree on k vert. has $k-1$ edges
- step: let T be a tree on $k+1$ vert. It has a leaf l . $T-l$ is still a tree. By IH, it has $k-1$ edges. So T has k edges.

Claim: T is a tree if and only if there is a unique path between vertices x and y .


PF: (\Rightarrow) if tree, acyclic / connected. Connected means @ least one path, acyclic means @ most 1 path.

(\Leftarrow) connected follows from unique path, acyclic occurs by unique path.

While trees are plenty interesting on their own (see Math 375), we're going to be most interested in making spanning trees, and in particular ones of minimum total weight. So we better define what that means.

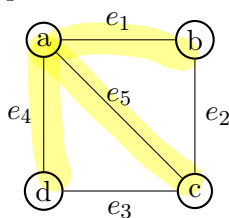
Definition (spanning tree) A spanning tree T of a ^{connected} graph G is a subgraph of G that (1) is a tree and (2) contains every vertex of G .

Claim: Every connected graph has a spanning tree.

Pf: let G be a connected graph. If G is acyclic, then G is a tree, and it itself is a spanning tree. If it has a cycle, find an edge on the cycle. Delete this edge. G is still connected.
 Continue until G is acyclic.

But what about my LPs (or, more helpfully here IPs)! This existence claim can be read as a *feasibility* claim!

Example: (1) Write an integer feasibility program for the spanning tree problem for the graph below.



$$\sum_{i=1}^5 e_i = 3$$

$$e_1 + e_2 + e_5 \leq 2$$

$$e_3 + e_4 + e_5 \leq 2$$

$$e_i \in \{0, 1\}$$

(2) Using your example, write a general integer feasibility program that returns a spanning tree of a graph $G = (V, E)$.

$$\sum_{i=1}^m e_i = n - 1$$

$$\forall x \subset V, x \neq \emptyset, \sum_{e_i \in x} e_i \leq |x| - 1$$

$$e_i \in \{0, 1\}$$

e as a zero/one vector restricted to E

$$e(E) = n - 1$$

$$e(\gamma(s)) \leq |s| - 1$$

$$\forall s \subset V, s \neq \emptyset$$

$$\gamma(s) = \{uv \in E: u \in s, v \notin s\}$$

Excellent! We've got an integer program! We could relax it to an LP, use a branch and bound technique, and we're good, right?

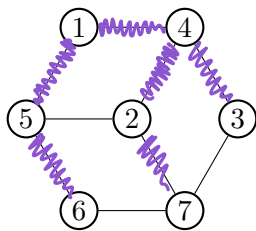
Well...

way too big. 2^n constraints. $|E|$ variables.

So here we run into our first application of a more specialized method to find what we're searching for, in this case a spanning tree. We'll use breadth first search, an excellent way to look at every vertex of a graph.

Breadth First Search: Start every vertex as unencountered. To explore a vertex, we mark all of its unencountered neighbors as seen and add them to a queue (encountered). After exploring a vertex, we mark it as explored, then move to explore the next vertex in the encountered queue.

Example: Run BFS to find a spanning tree on the graph below, starting at vertex 1. Keep track of your spanning tree as you go by noting where you first encounter a vertex!



encountered:

1	2	3	4	5	6	7
T	F	F	F	F	F	F
	T	T	T	T	T	T

explored:

X	4	5	2	3	6	7

↑ ↑ ↑

Algorithm 1 Breadth First Search $B(G, v)$

```

1: for  $v \leftarrow 1$  to  $|G|$  do # of vertices
2:    $EncounterQ[v] \leftarrow \text{False}$  } make our T, F array for encountered.
3: end for
4:  $ExploredQ[1] \leftarrow v$  add start to explored
5:  $EncounterQ[v] \leftarrow \text{True}$ 
→ 6:  $QSize \leftarrow 1$  track size of explored
→ 7:  $k \leftarrow 1$ 
8:  $Tree \leftarrow \text{empty list}$  keeps tree edges
9: repeat
10:   $\hat{v} \leftarrow ExploredQ[k]$ 
11:  for  $w \sim \hat{v}$  do
12:    if not  $EncounterQ[w]$  then
13:       $QSize \leftarrow QSize + 1$  increase encounterQ len counter
14:       $ExploredQ[QSize] \leftarrow w$  add to end of encountered
15:       $EncounterQ[w] \leftarrow \text{True}$  we've encountered it
16:      Add edge  $\hat{v}w$  to  $Tree$  add edge to tree
17:    end if
18:  end for
19:   $k \leftarrow k + 1$  move down the queue.
20: until  $k > QSize$  run through all of ExploredQ

```

Question: Can we build a spanning tree that contains every edge incident to a given vertex?

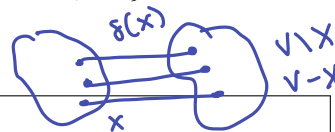
yes! start BFS there!

So we can formulate spanning trees as a linear program, but it's probably more efficient to instead run Breadth First Search to find one. But we originally talked about finding a minimum spanning tree, which means we need to talk weights and cuts.

Definition (edge weight, cost) For a graph G , we define an edge weight function $w: E \rightarrow \mathbb{R}$ (or, more typically \mathbb{R}^+ or \mathbb{N}). This is sometimes also called a *cost*. $w(e), w(\{u,v\}), c(e), c(\{u,v\})$

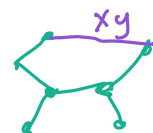
Definition (tree weight) The weight of any tree T is $\sum_{e \in T} w(e)$.

Definition (cut, cut weight) For $X \subset V$, $\delta(X) = \{xy \in E : x \in X, y \notin X\}$ is a cut. The weight of a cut is $\sum_{e \in \delta(X)} w(e)$.



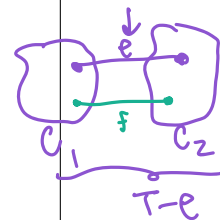
Claim: the following are equivalent.

- (1) T is a minimum spanning tree.
- (2) For every edge xy in $E(G) \setminus E(T)$, no edge on the $x-y$ path in T has a higher weight than $e = \{x,y\}$.
- (3) For every $e \in E(T)$, e is a minimum weight edge of the cut between the components of $T - e$.
- (4) We can order $E(T) = \{e_1 \dots e_{n-1}\}$ such that for each i there exists a set $X \subset V(G)$ such that e_i is the min cost edge of a cut X and no previous edge is in the cut X .

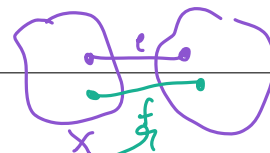


(1) \rightarrow (2) s/p k has a higher weight than e , where k is on $x-y$ path in T .
Then $T - k + e$ is smaller than T , T not MST \nexists (1)

(2) \rightarrow (3) suppose that C is a component of $T - e$, and let $f \in \delta(C)$ s.t. $w(f) < w(e)$. Then the two ends of f contradict (2)



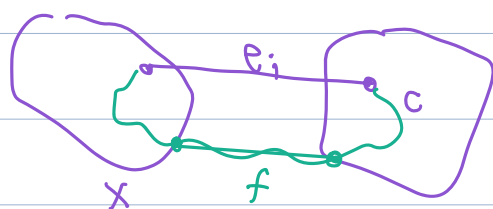
(3) \rightarrow (4) any ordering of the edges will work, where X is one of the components of $T - e$. e_i is min weight in $\delta(X)$ (by (3)) and cut has no other edges from T



can't happen

(4) \rightarrow (i) Let $E(T) = \{e_1, \dots, e_{n-1}\}$ be from (4). Let T^* be a minimum spanning tree st. $i = \min\{h : e_h \notin E(T^*)\}$ is maximized for T^* . In words, the first edge they differ in is as late in the list as possible.

Let X be the cut given by (4) for i , meaning e_i is the min. cost edge in $\delta(X)$ and $e_j \notin \delta(X) \forall j \in [1, 2, \dots, i-1]$. Then $T^* + e$ has a cycle, called C . Note, there is another edge on the cycle that is also in the cut, called f .



Note, $f \neq e_j$ for

$j \in [1, 2, \dots, i-1]$ by the claim of (4).

$T^* + e_i - f$ is a spanning tree, so $w(e_i) \geq w(f)$.

However, since $f \in \delta(X)$, $w(e_i) \leq w(f)$. Thus $T^* + e_i - f$ is an mst that has a later first uncommon edge than T^* ; contradicts our min. Thus $T = T^*$ and T is an MST.

Example: We can easily take our spanning tree feasibility program to a minimum spanning tree programs by setting an appropriate objective function. Write that program below.

$$\begin{aligned}
 \min \quad & C^T x_e \longrightarrow \text{decision variable, indexed by edges.} \\
 \text{st} \quad & \sum x_e = |V| - 1 \quad \neq! \\
 & \sum_{e \in H} x_e \leq |H| - 1 \quad \forall H \neq V \\
 & x_e \geq 0
 \end{aligned}$$

edges fully contained in H

Example: Now that we have a program, we can start thinking about the dual. Formulate the dual and consider what (if anything) it may mean.

$$\begin{aligned}
 \max \quad & \sum_{H \subseteq V} (|H| - 1) y_H \\
 \text{st.} \quad & \sum_{H \ni e} y_H \geq c_e \quad \forall e \in E \\
 & y_H \geq 0 \text{ for } H \neq V \\
 & y_V \text{ free}
 \end{aligned}$$

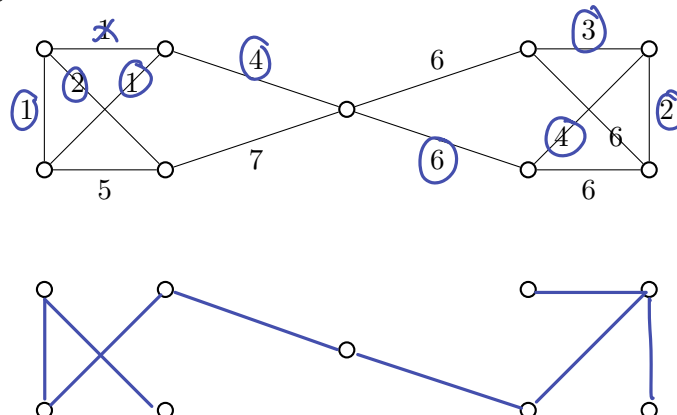
notes: Primal!

$$\begin{aligned}
 & H \neq V \quad \left[\begin{array}{c} \text{edges} \\ \uparrow \\ x_e \geq 0 \end{array} \right] \leq \left[\begin{array}{c} |H| - 1 \\ |V| - 1 \end{array} \right] \\
 & H = V \quad \left[\begin{array}{c} \text{edges} \\ \uparrow \\ x_e \geq 0 \end{array} \right] = \left[\begin{array}{c} |H| - 1 \\ |V| - 1 \end{array} \right] \\
 & \text{edges} \left[\begin{array}{c} H \\ \uparrow \\ y_H \end{array} \right] \left[\begin{array}{c} y_e \\ y_H \end{array} \right]
 \end{aligned}$$

This program suffers from the same problem that the feasibility one did. It's just too many constraints. Fortunately, we have another (or, rather, a few other) approach(es). We'll start with a (possibly) familiar one: Kruskal's Algorithm.

Kruskal's Algorithm: Sort all edges into a list in nondecreasing order and start with an empty graph H where $V(H) = V(G)$. Attempt edges from the list in order: if an edge causes a cycle to be added to H , do not add it. Stop when you arrive at $n - 1$ edges.

Example: Run Kruskal's Algorithm on the graph below and calculate the weight of the minimum spanning tree.



This is a slightly different formulation than what we described. Let's verify why it's equivalent.

Algorithm 2 Kruskal's Algorithm: $KRUSKAL(G)$

```

initialize  $Tree \leftarrow$  empty list
for  $u \in V(G)$  do
    initialize  $T_u$  to contain only  $u$ 
end for
 $t \leftarrow 0$ 
while  $t < |G| - 1$  do
    Pick next edge  $xy$  of minimum weight.
    Determine components of  $x$  and  $y$ .
    if  $T_x \neq T_y$  then
        Merge  $T_x$  and  $T_y$ .
        Add  $xy$  to  $Tree$ 
         $t \leftarrow t + 1$ 
    end if
end while

```

girth = size of smallest cycle in a graph

$G.girth \rightarrow +\infty$ if G is a tree.

Space for discussion...

Claim: Kruskal's algorithm produces a minimum spanning tree.

Pf: let T be a Kruskal's tree, $T = \{e_1, e_2, \dots, e_{n-1}\}$ where $w(e_i) \leq w(e_{i+1})$
 s/p T is not an MST. Pick $T^*^{(MST)}$ to contain the most edges
 in common w/ T . s/p e_k is the first edge in T not in T^*
 $T^* + e_k$ has a cycle. Let e_0 be an edge on the cycle not in T .
 Consider $T^* + e_k - e_0$

$$w(T^* + e_k - e_0) = w(T^*) + w(e_k) - w(e_0) \geq w(T^*) \rightarrow w(e_k) \geq w(e_0)$$

Cl: actually equality.

• if $k=1$: we're looking at least expensive edge, so equality follows.

• if not: e_0 doesn't form a cycle $\{e_1, \dots, e_{k-1}\}$ in T b/c it didn't form one in T^* , but then Kruskal's would take e_0 instead!

so $T^* + e_k - e_0$ is an MST, but more edges in common than T^* , \square .

So we know that Kruskal's will indeed create a minimum spanning tree, but can we verify that this minimum spanning tree creates a feasible solution to the program that we constructed earlier?

Yes! We totally can. Next ~~time~~.