Mathematical Foundations of Computer Science

CS 499, Shanghai Jiaotong University, Dominik Scheder

8 Spanning Trees

- Homework assignment published on Monday, 2018-04-16
- Submit questions and first solution by Sunday, 2018-04-22, 12:00, by email to me and the TAs.
- Submit your final solution by Sunday, 2018-04-29.

8.1 Minimum Spanning Trees

Throughout this assignment, let $G = (V, E)$ be a connected graph and $w : E \to \mathbb{R}^+$ be a weight function.

Exercise 8.1. Prove the inverse of the cut lemma: If X is good, $e \notin X$, and $X \cup e$ is good, then there is a cut $S, V \setminus S$ such that (i) no edge from X crosses this cut and (ii) e is a minimum weight edge of G crossing this cut.

Proof. Suppose we color each vertex black or white, and construct a cut $S, V \setminus S$ by defining S as the set of all black vertices and $V \setminus S$ as the set of all white vertices, then an edge crosses the cut if and only if its two vertices has different color. Let $X' = \{e' \mid w(e') < w(e), e' \in E\}$, $e = (u, v)$, then our goal is to find a way to color V such that (i) each edge in $X \cup X'$ has vertices with the same color and (ii) u and v have different color.

Just consider edges in $X \cup X'$ and ignore others, we color u and every vertex connects to u by one or several edges black. Next we'll prove that v is not colored. If there is a path between u and v, since $X \subseteq E(T)$, there must be an $e' \in X'$ in the path. The path and e can form a circle, so delete e from T and add e' will get a smaller minimum spanning tree, which contradicts the supposition "$X \cup e$ is good".

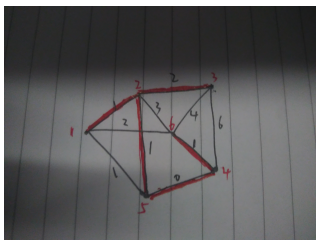
If a vertex is not colored, we color it white, then v is white. In this way, we find the cut. If an edge is in X or its weight is smaller than e, both vertices are either black or white at the same time, so the solution meets the conditions. □

Definition 8.2. *For $c \in \mathbb{R}$ and a weighted graph $G = (V, E)$, let $G_c := (V, \{e \in E \mid w(e) \leq c\})$. That is, G_c is the subgraph of G consisting of all edges of weight at most c.*

Lemma 8.3. *Let T be a minimum spanning tree of G, and let $c \in \mathbb{R}$. Then T_c and G_c have exactly the same connected components. (That is, two vertices $u, v \in V$ are connected in T_c if and only if they are connected in G_c).*

Exercise 8.4. Illustrate Lemma 8.3 with an example!

Answer



The red edges are included in the smallest spanning tree. Take T_1 and G_1 as examples. We can see the correctness of the lemma. It's also true for other values of c.

Exercise 8.5. Prove the lemma.

Proof. First, we will prove that if $u, v \in V$ are connected in T_c, they are connected in G_c. It's obvious, because $\forall e \in T_c$, $e \in G_c$.
Second, we will prove that if $u, v \in V$ are connected in G_c, they are connected in T_c. Now we assume u, v are not connected in T_c. Then we can see there

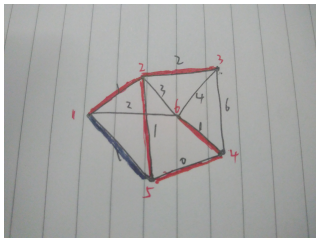
exists an edge e_1 whose weight is $> c$, and if we delete e_1 from T, u, v are not connected. Since u, v are connected in G_c, we can find an edge $e_2 \in G_c$ that keeps u, v connected in G_c. Replace e_1 with e_2, then we get a new spanning tree, with a smaller sum of weights, but it's contradict with T is a smallest spanning tree. So, if $u, v \in V$ are connected in G_c, they are connected in T_c. In conclusion, two vertices $u, v \in V$ are connected in T_c if and only if they are connected in G_c. □

Definition 8.6. *For a weighted graph G, let $m_c(G) := |\{e \in E(G) \mid w(e) \leq c\}|$, i.e., the number of edges of weight at most c (so G_c has $m_c(G)$ edges).*

Lemma 8.7. *Let T, T' be two minimum spanning trees of G. Then $m_c(T) = m_c(T')$.*

Exercise 8.8. Illustrate Lemma 8.7 with an example!

Answer



The red edges are included in the smallest spanning tree T. The blace edges are included in T'. Take $c = 1$ as an example. We can see $m_c(T) = m_c(T') = 4$. It's also true for other values of c.

Exercise 8.9. Prove the lemma.

Proof. By Lemma 8.3, we know T_c, T'_c and G_c share the same connected components. Since T_c and T'_c are both trees, the number of edges equals $|V| - |C|$, in which $|C|$ is the number of connected components. So $m_c(T) = m_c(T')$. □

Exercise 8.10. Suppose no two edges of G have the same weight. Show that G has exactly one minimum spanning tree!

Proof. First, we sort the edges' weight to get a strictly increasing sequence P Call this sequence P. Notice that each value in P represents an edge. Then $m_{P_i}(T)$ actually becomes then number of $P_i \in T,\ i \leq c$. For a smallest spanning tree, we can construct if given the m_c sequence of it with the following algorithm.

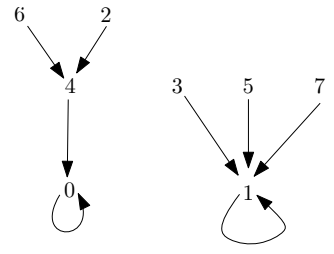
for i=1 to $|E| - 1$
  if $m_c(P_i)! = m_c(P_{i-1})$
    add the corresponding edge to T

With this algorithm, the only corresponding T can be constructed.
By Lemma 8.6, we know that if there are more than one smallest spanning trees, they all share the same sequence of m_c. Then we apply the algorithm to the sequence. We can see the output Ts are all the same. So there is actually only one smallest spanning tree for G. □

8.2 Counting Special Functions

In the video lecture, we have seen a connection between functions $f : V \to V$ and trees on V. We used this to learn something about the number of such trees. Here, we will go in the reverse direction: the connection will actually teach us a bit about the number of functions with a special structure.

Let V be a set of size n. We have learned that there are n^n functions $f : V \to V$. For such a function we can draw an "arrow diagram" by simply drawing an arrow from x to $f(x)$ for every V. For example, let $V = \{0, \ldots, 7\}$ and $f(x) := x^2 \mod 8$. The arrow diagram of f looks as follows:



The *core* of a function is the set of elements lying on cycles in such a diagram. For example, the core of the above function is $\{0, 1\}$. Formally, the core of f is the set

$$\{x \in V \mid \exists k \geq 1 f^{(k)}(x) = x\}$$

where $f^{(k)}(x) = f(f(\dots f(x)\dots))$, i.e., the function f applied k times iteratively to x.

Exercise 8.11. Of the n^n functions from V to V, how many have a core of size 1? Give an explicit formula in terms of n.

Answer There are n^{n-1} functions have a core of size 1.

Proof. There are n^{n-2} trees on n vertices. Choose one of those vertices to make a self-cycle. We have n choices. So the answer is $n * n^{n-2} = n^{n-1}$. □

Exercise 8.12. How many have a core of size 2 that consists of two 1-cycles? By this we mean that $\text{core}(f) = \{x, y\}$ with $f(x) = x$ and $f(y) = y$.

Answer There are $(n-1)*n^{n-2}$ functions have a core of size 2 that consists of two 1-cycles.

Proof. We can remove the edges on the vertebrate and regard the vertices in the core as the root of different trees. We need to prove that the two vertices in the 2-size core cannot in a same tree. If they are in a same tree, they will have the same ancestor node, but it is impossible. There are n^{n-2} trees on n vertices. Choose one of edges to delete it and the vertices of this edge are the core of this function. We have $n-1$ choices. So the answer is $(n-1) * n^{n-2}$. □

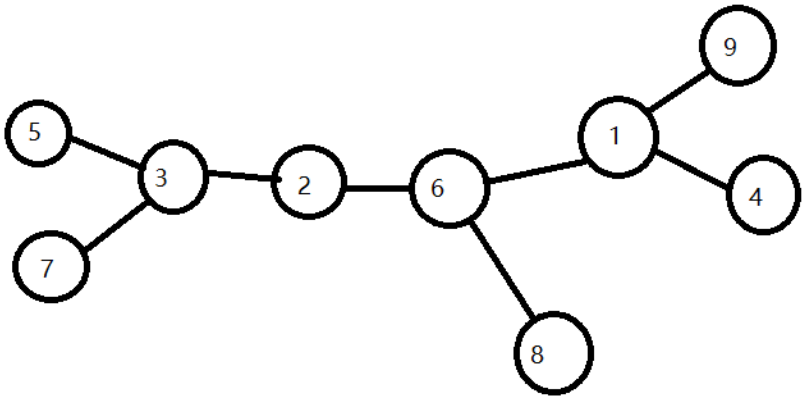
Hint. For the previous two exercises, you need to use the link between functions $f : [n] \to [n]$ and vertebrates (T, h, b) from the video lecture.

8.3 Counting Trees with Prüfer Codes

In the video lecture, we have seen Cayley's formula, stating that there are exactly n^{n-2} trees on the vertex set $[n]$. We showed a proof using *vertebrates*. For this homework, read Section 7.4 of the textbook, titled "A proof using the Prüfer code".

Exercise 8.13. Let $V = \{1, \dots, 9\}$ and consider the code $(1, 3, 3, 2, 6, 6, 1)$. Reconstruct a tree from this code. That is, find a tree on V whose Prüfer code is $(1, 3, 3, 2, 6, 6, 1)$.

Answer The answer is as follow:



Exercise 8.14. Let $\mathbf{p} = (p_1, p_2, \ldots, p_{n-2})$ be the Prüfer code of some tree T on $[n]$. Find a way to quickly determine the degree of vertex i only by looking at $\mathbf{p}$ and not actually constructing the tree T. In particular, by looking at $\mathbf{p}$, what are the leaves of T?

Answer The degree of vertex i equals to $|T(i)|+1$, $T(i) = \{j|p_j = i, 1 \leq j \leq n-2\}$. The proof is as follows:

1. If $|T(i)| = 0$, then vertex i is a leaf, otherwise it should connect to at least two vertices, delete each of which makes i in the Prüfer code.

2. If $|T(i)| \neq 0$ and vertex i is deleted, then vertex i has $|T(i)|$ neighbors deleted before and a neighbor exists while deleting vertex i itself.

3. If $|T(i)| \neq 0$ and vertex i is not deleted, then vertex i has $|T(i)|$ neighbors deleted before and a neighbor exists at last.

To sum up, the leaves of T are all vertex i such that $T(i) = 0$. That is to say, the leaves are all vertex i that i is not in the Prüfer code.

Exercise 8.15. Describe which tree on $V = [n]$ has the

1. Prüfer code $(1, 1, \ldots, 1)$.

2. Prüfer code $(1, 2, 3, \ldots, n-2)$.

3. Prüfer code $(3, 4, 5, \ldots, n)$.

4. Prüfer code $(n, n-1, n-2, \ldots, 4, 3)$.

5. Prüfer code $(n-2, n-3, \ldots, 2, 1)$.

6. Prüfer code $(1, 2, 1, 2, \ldots, 1, 2)$ (assuming n is even).

Justify and explain your answers.

Answer

1. The tree is that 1 connect to other n-1 nodes.

   Because the prüfer code are all 1, so the first n-2 step will delete 1 in prüfer code and the minimum number in [n] except 1, so the number from 2 to n-1 will connect to 1. When the prüfer code are deleted all, there are 1 and n remained, so the last step is connect 1 and n.

2. The tree is that 1 connect 2 and n-1, 2 connect to 3, 3 connect to 4...n-3 connect to n-2, n-2 connect to n.

   The first step will delete 1 in prüfer code and n-1 in [n], and 1 can be delete because there is no 1 in prüfer code, so the second step will delete 2 in prüfer code and 1 in [n]...recursively delete i in prüfer code and i-1 in [n] until i=n-2. There are n-2 and n left in [n], so the last step will delete n-2 and n.

3. The tree is that there are two parts, the one part consist of odd number and connect with the order of size, the another part consist of even number and connect with the order of size. The two parts are connected by n-1 connect to n.

   The first step is delete 3 in prüfer code and 1 in [n], the second step is delete 4 in prüfer code and 2 in [n]. Recursively the ith step will delete i+2 in prüfer code and i in [n] until i=n-2. Because there are n-1 and n remained in [n], the last step will delete n-1 and n and connect them.

4. The tree is that 2 connect to n-1, n-1 connect to n-2, n-2 connect to n-3...3 connect to n, n connect to 1.

   The first step will delete n in prüfer code and 1 in [n], the second step will delete n-1 in prüfer code and 2 in [n], after the second step there are n and n-1 can be deleted in [n], and each delete will bring the number which is smaller than n that can be deleted in next step, so recursively

the ith step will delete n-i+1 in prüfer code and n-i+2 in [n] until i=n-2. Because there are 3 and n remained in [n], the last step will delete 3 and n and connect them.

5. The tree is that n-1 connect to n-2, n-2 connect to n-3...2 connect to 1, and 1 connect to n.

   The first step will delete n-1 in prüfer code and n-2 in [n], each delete will bring the number which is smaller than n that can be deleted in next step, so recursively the ith step will delete n-i-1 in prüfer code and n-i in [n] until i=n-2. Because there are 1 and n remained in [n], the last step will delete 1 and n and connect them.

6. The tree is that 1 connect to all other odd numbers, 2 connect to all other even numbers, and 1 connect to 2.

   Once delete 1 in prüfer code, we will delete the smallest odd number except 1, and once delete 2 in prüfer code, we will delete the smallest even number except 2 until the last three steps. The last third step will delete 1 in prüfer code and n-1 in [n], after that 1 can be delete in [n], so the last second step will delete 2 in prüfer code and 1 in [n], then delete 2 and n in the last step and connect them.

The next two exercises use a bit of probability theory. Suppose we want to sample a random tree on $[n]$. That is, we want to write a little procedure (say in Java) that uses randomness and outputs a tree T on $[n]$, where each of the n^{n-2} trees has the same probability of appearing.

Exercise 8.16. Sketch how one could write such a procedure. Don't actually write program code, just describe it informally. You can assume you have access to a random generator `randomInt(`n`)` that returns a function in $\{1, \ldots, n\}$ as well as `randomReal()` that returns a random real number from the interval $[0, 1]$.

Solution

We can use `randomReal()` to generate $n-2$ random integers in $[n]$, and then decode the sequence as a Prüfer code into a tree. The sequence is uniformly random, so the corresponding tree has same probability.

Clearly, a tree T on $[n]$ has at least 2 and at most $n-1$ leaves. But how many leaves does it have on average? For this, we could use your tree sam-

pler from the previous exercise, run it 1000 times and compute the average. However, it would be much nicer to have a closed formula.

Exercise 8.17. Fix some vertex $u \in [n]$. If we choose a tree T on $[n]$ uniformly at random, what is the probability that u is a leaf? What is the expected number of leaves of T?

Answer

1. If u is a leaf, it'll not appear in the tree's Prüfer code. The probability of u not appearing in the sequence is $P_{leaf} = \frac{(n-1)^{n-2}}{n^{n-2}} = (1 - \frac{1}{n})^{n-2}$.

Note that if n is large, the probability tend to be $\frac{1}{e}$

2. Let random variable $X_i \sim B(P_{leaf}, 1)$ represents whether the vertex v_i is a leaf or not. Because of the linearity of expectation, we can know:

$$\begin{aligned} E(\#leaves\ of\ a\ tree) &= E[\sum_{i=1}^{n} X_i] \\ &= \sum_{i=1}^{n} E(X_i) \\ &= nP_{leaf} \\ &= \frac{(n-1)^{n-2}}{n^{n-3}} \end{aligned}$$

(Thanks for Dominik's suggestion and explanation!)

Exercise 8.18. For a fixed vertex u, what is the probability that u has degree 2?

Answer

u has degree 2 means u appears exactly once in the Prüfer code. So the probability is

$$P = \frac{(n-2)(n-1)^{n-3}}{n^{n-2}}$$