2
Arborescences: Directed Spanning Trees

Greedy algorithms worked vey well for minimum weight spanning tree problem, as we saw in Chapter 1. In this chapter, we define arborescences which are a notion of spanning trees for rooted directed graphs. We will see that a naïve greedy approach no longer works, but it requires just a slightly more sophisticated algorithm to efficiently find them. We give two proofs of correctness for this algorithm. The first is a direct inductive proof, but the second makes use of linear programming duality, and highlights its use in analyzing the performance of algorithms. This will be a theme that return to multiple times in this course.

2.1 *Arborescences*

Consider a graph $G = (V, A, w)$: here V is a set of vertices, and A a set of directed edges, also known as ***arcs***. The function $w : A \to \mathbb{R}$ gives a weight to every arc. Let $|V| = n$ and $|A| = m$. Once we root G at a node $r \in V$, we can define a "directed spanning tree" with r being the *sink/root*.

We will use "arcs" instead of "edges" to emphasize the directedness of the graph.

Definition 2.1. An ***r-arborescence*** is a subgraph $T = (V, A')$ with $A' \subseteq A$ such that

A ***branching*** is the directed analog of a forest; it merely drops the connectivity requirement that arborescences have.

1. T forms a spanning tree in the undirected sense, and
2. Each vertex except r has one outgoing arc, r has none.

Remark 2.2. Every vertex in an arborescence has a directed path from itself to the root r. This property (along with either property 1 or property 2) can alternatively be used to define an arborescence. Add argument, or exercise.

Remark 2.3. It's easy to check if an r-arborescence exists. We can reverse the arcs and run a depth-first search from the root. If all vertices are reached, we have produced an r-arborescence.

The focus of this chapter is to find the minimum-weight r-arborescence. We can simplify things slightly by assuming that all of the weights are non-negative. Because no outgoing arcs from r will be part of any arborescence, we can assume no such arcs exist in G either. For brevity, we fix r and simply say arborescence when we mean r-arborescence.

If there are negative arc weights, add a large positive constant M to every weight. This increases the total weight of each arborescence by $M(n-1)$, and hence the identity of the minimum-weight one remains unchanged.

2.1.1 *The Limitations of Greedy Algorithms*

It's natural to ask if greedy algorithms like those in Chapter 1 for the directed case. E.g., we can try picking the lightest incoming arc into the component containing r, as in Prim's algorithm, but this fails, for example in Figure 2.1. Or we could emulate Kruskal's algorithm and consider arcs in increasing order of weight, adding them if they don't close a directed cycle. (Exercise: give an example where it fails.) The problem is that greedy algorithms (that consider the arcs in some linear order and irrevocably add them in) don't see to work. However, the algorithm we eventually get will feel like Borůvka's algorithm, but one where we are allowed to revoke some of our past decisions.

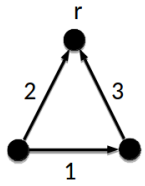

Figure 2.1: A Prim-like algorithm will select the arc with weight 2 and 3, whereas the optimal choices are the arcs with weights 3 and 1.

2.2 *The Chu-Liu/Edmonds/Bock Algorithm*

The algorithm we present was discovered independently by Yoeng-Jin Chu and Tseng-Hong Liu [1], Jack Edmonds, and F. Bock [2]. We will follow Karp's presentation of Edmonds' algorithm.

[1] Edmonds (1967)

[2] Karp (1971)

Definition 2.4. For a vertex $v \in V$ or subset of vertices $S \subseteq V$, let $\partial^+ v$ and $\partial^+ S$ denote the set of arcs leaving the node v and the set S, respectively.

Definition 2.5. For a vertex $v \in V$ in graph G, define $M_G(v) := \min_{a \in \partial^+ v} w(a)$ be the minimum weight among arcs leaving v in G.

The first step is to create a new graph G' by subtracting some weight from each outgoing arc from a vertex, such that there is at least one arc of weight 0. That is, set $w(a') \leftarrow w(a) - M_G(v)$ for all $a \in \partial^+ v$ and each $v \in V$.

Claim 2.6. T is a min-weight arborescence in $G \iff T$ is a min-weight arborescence in G'.

Proof. Each arborescence has exactly one arc leaving each vertex. Decreasing the weight of every arc exiting v by $M_G(v)$ decreases the weight of every possible arborescence by $M_G(v)$ as well. Thus, the set of min-weight arborescences remains unchanged. □

Now each vertex has at least one 0-weight arc leaving it. Now, for each vertex, pick an arbitrary 0-weight arc out of it. If this choice is an arborescence, this must be the minimum-weight arborescence, since all arc weights are still nonnegative. Otherwise, the graph consist of some connected components, each of which has one directed cycle along with some acyclic incoming components, as shown in the figure.

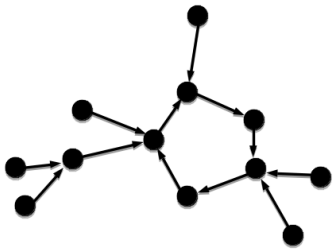

Figure 2.2: An example of a possible component after running the first step of the algorithm

For the second step of the algorithm, consider one such 0-weight cycle C, and construct a new graph $G'' := G'/C$ by contracting the cycle C down to a single new node v_C, removing arcs within C, and replacing parallel arcs by the cheapest of these arcs. Let $\mathrm{OPT}(G)$ denote the weight of the min-weight arborescence on G.

Claim 2.7. $\mathrm{OPT}(G') = \mathrm{OPT}(G'')$.

Proof. To show $\mathrm{OPT}(G') \leq \mathrm{OPT}(G'')$, we exhibit an arborescence in G' with weight at most $\mathrm{OPT}(G'')$. Indeed, let T'' be a min-weight arborescence in G''. Consider arborescence T' in G' obtained by expanding v_C back to the cycle C, and removing one arc in the cycle. Since the cycle has weight 0 on all its arcs, T' has the same weight as T''. (See Figure 2.3.)

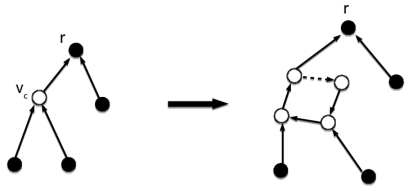

Figure 2.3: The white node is expanded into a 4-cycle, and the dashed arrow is the arc that is removed after expanding.

Now to show $\mathrm{OPT}(G'') \leq \mathrm{OPT}(G')$, take a min-weight arborescence T' of G', and identify the nodes in C down to get a vertex v_C. The resulting graph is clearly connected, with each vertex having a directed path to the root. Now remove some arcs to get an arborescence of G'', e.g., as in Figure 2.4. Since arc weights are non-negative, we can only lower the weight by removing arcs. Therefore $\mathrm{OPT}(G'') \leq \mathrm{OPT}(G')$. □

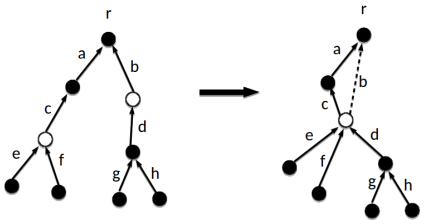

Figure 2.4: Contracting the two white nodes down to a cycle, and removing arc b.

The proof also gives an algorithm for finding the min-weight arborescence on G' by contracting the cycle C (in linear time), recursing on G'', and the "lifting" the solution T'' back to a solution T'. Since we recurse on a graph which has at least one fewer nodes, there are at most n recursive calls. Moreover, the weight-reduction, contraction, and lifting steps in each recursive call take $O(m)$ time, so the runtime of the algorithm is $O(mn)$.

Remark 2.8. This is not the best known run-time bound: there are many optimizations possible. Tarjan [3] presents an implementation of the above algorithm using priority queues in $O(\min(m \log n, n^2))$ time, and Gabow, Galil, Spencer and Tarjan [4] give an algorithm to solve the min-weight arborescence problem in $O(n \log n + m)$ time. The best runtime currently known is $O(m \log \log n)$ due to Mendelson et al. [5].

[3]

[4]

[5]

Open problem 2.9. Is there a linear-time (randomized or deterministic) algorithm to find a min-weight arborescence in a digraph G?

2.3 *Linear Programming Methods*

Let us now see an alternate proof of correctness of the algorithm above, this time using linear programming duality. This is how Edmonds originally proved his algorithm to be optimal.

If you have access to the Chu-Liu or Bock papers, I would love to see them.

2.3.1 *Linear Programming Review*

Before we actually represent the arborescence problem as a linear program, we first review some standard definitions and results from linear programming.

Definition 2.10. For some number of variables (a.k.a. dimension) $n \in \mathbb{N}$, number of constraints $m \in \mathbb{N}$, objective vector $c \in \mathbb{R}^n$, constraint matrix $A \in \mathbb{R}^{n \times m}$, and right-hand side $b \in \mathbb{R}^m$, a (minimization) *linear program* (LP) is

This form of the LP is called the *standard form*. More here.

$$\text{minimize } c^\intercal x \quad \text{subject to } Ax \geq b \text{ and } x \geq 0$$

Note that $c^\intercal x$ is the inner product $\sum_{i=1}^{n} c_i x_i$.

The constraints of a linear program form a ***polyhedron***, which is the convex body formed by the intersection of a finite number of half spaces. Here we have $m + n$ half spaces. There are m of them corresponding to the constraints $\{a_i^\intercal x \geq b_i\}_{i=1}^{m}$, where $a_i \in \mathbb{R}^n$ is the vector corresponding to the i^{th} row of the matrix A. Moreover, we have n non-negativity constraints $\{x_j \geq 0\}_{j=1}^{n}$. If the polyhedron is bounded, we call it a ***polytope***.

Whenever we write a vector, we imagine it to be a column vector.

Definition 2.11. A vector $x \in \mathbb{R}^n$ is called *feasible* if it satisfies the constraints: i.e., $Ax \geq b$ and $x \geq 0$.

Definition 2.12. Given a linear program $\min\{c^\intercal x \mid Ax \leq b, x \geq 0\}$, the *dual linear program* is

$$\text{maximize } b^\intercal y \quad \text{subject to } A^\intercal y \leq c \text{ and } y \geq 0$$

The dual linear program has a single variable y_i for each constraint in the original (primal) linear program. This variable can be thought of as giving an importance weight to the constraint, so that taking a linear combination of constraints with these weights shows that the primal cannot possibly surpass a certain value for $c^\intercal x$. This purpose is exemplified by the following theorem.

Theorem 2.13 (Weak Duality)**.** *If x and y are feasible solutions to the linear program $\min\{c^\intercal x \mid Ax \leq b, x \geq 0\}$ and its dual, respectively, then $c^\intercal x \geq b^\intercal y$.*

Proof. $c^\intercal x \geq (A^\intercal y)^\intercal x = y^\intercal A x \geq y^\intercal b = b^\intercal y$. □

See the ***strong duality theorem*** in Here for a converse to this theorem. For now, weak duality will suffice.

This principle of weak duality tells us that if we have feasible solutions x, y where $c^\intercal x = b^\intercal y$, then we know that both x and y are optimal solutions. Our approach will be to give a linear program that models min-weight arborescences, use the algorithm above to write a feasible solution to the primal, and then to exhibit a feasible solution to the dual such that the primal and dual values are the same—hence both must be optimal!

2.3.2 *Arborescence Linear Program*

To analyze the algorithm, we first need to come up with a linear program that "captures" the min-weight arborescence problem. Since we want to find a set of arcs forming an arborescence T, we have one variable x_a for each arc $a \in A$. Ideally, each variable will be an *indicator* for the arc being in the arborescence: i.e., it will binary values: $x_a \in \{0,1\}$, with $x_a = 1$ if and only if $a \in T$. This choice of variables allows us to express our objective to minimize the total weight: $w^\intercal x := \sum_{a \in A} w(a) x_a$.

Next, we need to come up with a way to express the constraint that T is a valid arborescence. Let $S \subseteq V - \{r\}$ be a set of vertices not containing the root, and some vertex $v \in S$. Every vertex must be able to reach the root by a directed path. If $\partial^+ S \cap T = \emptyset$, there is no arc in T leaving the set S, and hence we have no path from v to r. We conclude that, at a minimum, $\partial^+ S \cap T \neq \emptyset$. We represent this constraint by ensuring that the number of arcs out of S is non-zero, i.e.,

$$\sum_{a \in \partial^+ S} x_a \geq 1.$$

We write an integer linear programming (ILP) formulation for min-weight arborescences as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{a \in A} w(a) x_a \\ \text{subject to} \quad & \sum_{a \in \partial^+ S} x_a \geq 1 \quad \forall S \subseteq V - \{r\} \\ & \sum_{a \in \partial^+ v} x_a = 1 \quad \forall v \neq r \\ & x_a \in \{0,1\} \qquad \forall a \in A. \end{aligned} \tag{2.1}$$

The following lemma is easy to verify:

Lemma 2.14. *T is an arborescence of G with $x_a = \mathbf{1}_{a \in T}$ if and only if x is feasible for the integer LP (2.2). Hence the optimal solution to the ILP (2.1) is exactly the min-weight arborescence.*

Relaxing the Boolean integrality constraints gives us the linear programming relaxation:

$$\begin{aligned} \text{minimize} \quad & \sum_{a \in A} w(a) x_a \\ \text{subject to} \quad & \sum_{a \in \partial^+ S} x_a \geq 1 \quad \forall S \subseteq V - \{r\} \\ & \sum_{a \in \partial^+ v} x_a = 1 \quad \forall v \neq r \\ & x_a \geq 0 \qquad \forall a \in A. \end{aligned} \tag{2.2}$$

Since we have *relaxed* the constraints, the optimal solution to the (fractional) LP (2.2) can only have less value than the ILP (2.1), and hence the optimal value of the LP is at most $\mathrm{OPT}(G)$. In the following, we show that it is in fact *equal* to $\mathrm{OPT}(G)$!

Exercise 2.15. Suppose all the arc weights are non-negative. Show that the optimal solution to the linear program remains unchanged even if drop the constraints $\sum_{a \in \partial^+ v} x_a = 1$. Add some clean combinatorial solution?

2.3.3 *Showing Optimality*

The output T of the Chu-Liu/Edmonds/Bock algorithm is an arborescence, and hence the associated solution x (as defined in Lemma 2.14) is feasible for ILP (2.1) and hence for LP (2.2). To show that x is optimal, we now exhibit a vector y feasible for the dual linear program with objective equal to $w^\intercal x$. Now weak duality implies that both x and y must be optimal primal and dual solutions.

The dual linear program for (2.2) is

$$\begin{aligned} \text{maximize} \quad & \sum_{S \subseteq V - \{r\}} y_S \\ \text{subject to} \quad & \sum_{S : a \in \partial^+ S} y_S \leq w(a) \quad \forall a \in A \\ & y_S \geq 0 \quad \forall S \subseteq V - \{r\}, |S| > 1. \end{aligned} \tag{2.3}$$

Observe that y_S is unconstrained when $|S| = 1$, i.e., S corresponds to a singleton non-root vertex.

We think of y_S as *payments* raised by vertices inside set S so that we can buy an arc leaving S. In order to buy an arc a, we need to raise $w(a)$ dollars. We're trying to raise as much money as possible, while not overpaying for any single arc a.

Lemma 2.16. *If arc weights are non-negative, there exists a solution for the dual LP (2.3) such that $w^\intercal x = \mathbb{1}^\intercal y$, where all y_e values are non-negative.*

Proof. The proof is by induction over the execution of the algorithm.

- The base case is when the chosen zero-weight arcs out of each node form an arborescence. In this case we can set $y_S = 0$ for all S; since all arc weights are non-negative, this is a feasible dual solution. Moreover, both the primal and dual values are zero.

- Suppose we subtract $M := M_G(v)$ from all arcs leaving vertex v in graph G so that v has at least one zero-weight arc leaving it. Let G' be the graph with the new weights, and let T' be the optimal solution on G'. By induction on G', let y' be a non-negative solution such that $\sum_{a \in T'} w'_e = \sum_S y'_S$. Define $y_v := y'_v + M$ and $y_S = y'_S$ for all other subsets; this is the desired feasible dual solution for the same tree $T = T'$ on the original graph G. Indeed, for one of the arcs $a = (v, u)$ out of the node v, we have

$$\begin{aligned}\sum_{S: a \in \partial^+ S} y_S &= \sum_{S: a \in \partial^+ S, |S|=1} y_S + \sum_{S: a \in \partial^+ S, |S| \geq 2} y_S \\ &= (y'_{\{u\}} + M) + \sum_{S: a \in \partial^+ S, |S| \geq 2} y'_S \\ &\leq M + w'(a) = M + (w(a) - M) = w(a).\end{aligned}$$

  Moreover, the value of the dual increases by M, the same as the increase in the weight of the arborescence.

- Else, suppose the chosen zero-weight arcs contain a cycle C, which we contract down to a node v_C. Using induction for this new graph G', let y' be the feasible dual solution. For any subset S' of nodes in G' that contains the new node v_C, let $S = (S' \setminus \{v_C\}) \cup C$, and define $y_S = y'_{S'}$. For all other subsets S in G' not containing v_C, define $y_S = y'_S$. Moreover, for all nodes $v \in C$, define $y_{\{v\}} = 0$. The dual value remains unchanged, as does the weight of the solution T obtained by lifting T'. The dual constraint changes only arcs of the form $a = (v, u)$, where $v \in C$ and $u \notin C$. But such an arc is replaced by an arc $a' = (v_C, u)$, whose weight is at most $w(a)$. Hence

$$\sum_{S: a \in \partial^+ S} y_S = y'_{\{v_C\}} + \sum_{S': a' \in \partial^+ S', S' \neq \{v_C\}} y'_S \leq w(a') \leq w(a).$$

  This completes the inductive proof

Notice that the sets with non-zero weights correspond to singleton nodes, or to the various cycles contracted during the algorithm. Hence these sets form a *laminar* family; i.e., any two sets S, S' with non-zero value in y are either disjoint, or one is contained within the other. □

By Lemma 2.16 and weak duality, we conclude that the solution x and the associated arborescence T is optimal. It is easy to extend the argument to potentially negative arc weights.

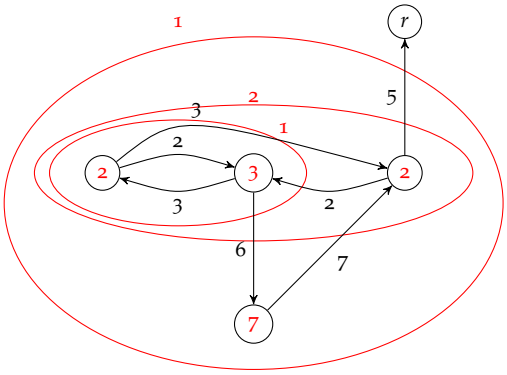

Figure 2.5: An optimal dual solution: vertex sets are labeled with dual values, and arcs with costs.

Corollary 2.17. *There exists a solution for the dual LP (2.3) such that $w^\intercal x = \mathbb{1}^\intercal y$. Hence the algorithm produces an optimal arborescence even for negative arc weights.*

Proof. If some arc weights are negative, add M to all arc weights to get the new graph G' where all arc weights are positive. Let y' be the optimal dual for G' from Lemma 2.16; define $y_S = y'_S$ for all sets of size at least two, and $y_{\{v\}} = y'_{\{v\}} - M$ for singletons. Note that the weight of the optimal solution on G is precisely $M(n-1)$ smaller than on G'; the same is true for the total dual value. Moreover, for arc $e = (u, v)$, we have

$$\sum_{S: a \in \partial^+ S} y_S = \sum_{S: a \in \partial^+ S, |S| \geq 2} y'_S + (y'_{\{u\}} - M) \leq (w_e + M) - M = w_e.$$

The inequality above uses that y' is a feasible LP solution for the graph G' with inflated arc weights. Finally, since the only non-negative values in the dual solution are for singleton sets, all constraints in (2.2) are satisfied for the dual solution y, this completes the proof. □

2.3.4 *Integrality of the Polytope*

Introduce a little more?

The result of Corollary 2.17 is quite exciting: it says that no matter what the objective function of the linear program (i.e., the arc weights $w(a)$), there is an optimal *integral* solution to the linear program, which our combinatorial algorithm finds. In other words, the optimal solutions to the LP (2.2) and the ILP (2.1) are the same.

We will formally discuss this later in the course, but let us start playing with these kinds of ideas. A good start is to visualize this geometrically: let $\mathcal{A} \subseteq \mathbb{R}^{|A|}$ be the set of all solutions to the ILP (which correspond to the characteristic vectors of all valid r-arborescences). This is a finite set of points, and let K_{arb} be the convex hull of these points. (It can be shown that K_{arb} is a polytope, though we don't do it here.) If we optimize a linear function given by some weight vector w over this polytope, we get the optimal arborescence for this weight. This is the solution to ILP (2.1).

Moreover, let $K \subseteq \mathbb{R}^{|A|}$ be the polytope defined by the constraints in the LP relaxation (2.2). Note that each point in $\mathcal{A}$ is contained within K, therefore so is their convex hull K. I.e.,

$$K_{arb} \subseteq K.$$

Picture. In general, the two polytopes are not equal. But in this case, Corollary 2.17 implies that for this particular setting, the two are indeed equal. Indeed, a geometric hand-wavy argument is easy to

make — if K were strictly bigger than K_{arb}, there would be some direction in which K extends beyond K_{arb}. But each direction corresponds to a weight-vector, and hence for that weight vector the optimal solution within K (which is the solution to the LP) would differ from the optimal solution within K_{arb} (which is the solution to the ILP). This contradicts Corollary 2.17.

2.4 *Matroid Intersection*

Blah