1 Preliminaries

Although we will generalize undirected minimum spanning trees (MSTs) to matroids in the homework, in this lecture we will generalize another direction. Arborescences, also known as branchings, are spanning trees (in the undirected sense) on rooted directed graphs. We will present Chu-Liu/Edmonds/Bock's algorithm for efficiently finding minimum-cost arborescences, and two proofs of correctness [CL65, Edm67, Boc71]. The second proof makes use of linear programming and forecasts some techniques we will use later in the class.

1.1 Arborescences

Consider the setting where we have a graph $G = (V, A)$, with V a set of vertices, and A a set of directed edges, also known as arcs. We call the sizes $|V| = n$ and $|A| = m$. Once we root G at a node $r \in V$, we can define a spanning tree with r as the sink.

Definition 2.1. An r-arborescence is a subgraph $T \subseteq G$ such that

1. T forms a spanning tree in the undirected sense.
2. Each vertex has exactly 1 outgoing arc, except r (which has none).

Remark 2.2. Another property of an arborescence is that every vertex has a directed path to the root r. This property (along with property 2) can alternatively be used to define an arborescence.

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

For optimization, we assign each arc $a \in A$ a weight w_a. We want to find the minimum-weight r-arborescence. We can simplify things slightly by assuming that all of the weights are non-negative. If not, add a large positive constant M to every arc. This will increase the total weight of every arborescence by $M(n-1)$ and leave the structure of the minimum-weight one unchanged. Because no outgoing arcs from r will be part of any arborescence, we can assume no such arcs exist in G either.

2 Chu-Liu/Edmonds/Bock

It's natural to ask if a greedy algorithm similar to the undirected case will work. We can try picking the smallest incoming edge to the component containing r, as in Prim's algorithm, but this fails, for example in Figure 2.1. The algorithm will select the edge with weight 2, and then the edge with weight 3, but the optimal is to choose the edges with weights 3 and 1.

We therefore need a more sophisticated algorithm for the directed case. The algorithm we present here was discovered independently by Chu-Liu [CL65], Edmonds [Edm67], and Bock [Boc71]. We will follow Karp's [Kar72] presentation of Edmonds' algorithm.

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 v and S, respectively.

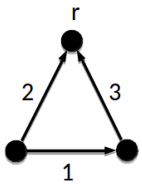

Figure 2.1: An example where the greedy algorithm fails

Definition 2.5. For a vertex $v \in V$, let $M_v = \min_{a \in \partial^+ v} w_a$.

For the first step in the algorithm, we create a new graph G' by setting $w'_a \leftarrow w_a - M_v$ for all $a \in \partial^+ v$ for each $v \in V$. In other words, we subtract some weight from each outgoing arc from a vertex, such that there is at least one arc of weight 0. Another way of thinking about this is that instead of only having weights on arcs, we also have weights on nodes which must be paid when selecting an edge leaving that node.

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

Proof. Every vertex must have exactly one edge leaving it. If we decrease the weight of every exiting arc by M_v, we decrease the weight of every possible arborescence by M_v as well. Thus, we do not affect the min-weight arborescence. □

Each vertex has at least one 0-weight arc leaving it. For each vertex, we pick a 0-weight edge out of it. If this is an arborescence, this must be the minimum weight arborescence, since all edge weights are still nonnegative. Otherwise, the graph consist of some connected components, each of which have one directed cycles along with some acyclic incoming components (for example Figure 2.2).

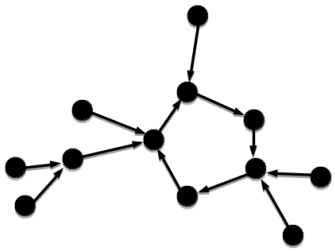

Figure 2.2: An example of a possible component after running the first step of the algorithm

Consider some 0-cost cycle C. In the second step of the algorithm, we construct a new graph, $G'' = G'/C$, which is G' with the cycle C contracted to 1 node, removing arcs within C, and replacing parallel arcs by the cheapest arc.

Claim 2.7. *Let $OPT(G)$ be the cost of the min-weight arborescence on G. We claim $OPT(G') = OPT(G'')$.*

Proof. We first show $\mathrm{OPT}(G') \leq \mathrm{OPT}(G'')$. Suppose we have a min-weight arborescence T'' of G''. There is some node v_c which represents some cycle in G'. We can construct an arborescence T' of G' by expanding the cycle, and removing one edge in the cycle. Since the cycle has weight 0 on all

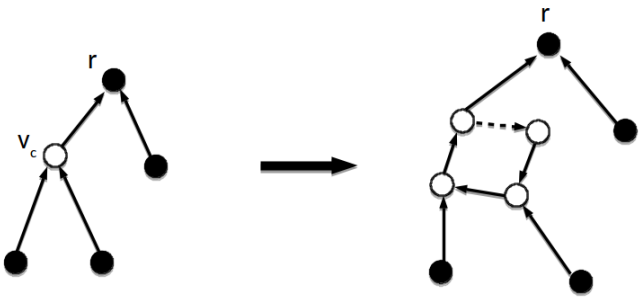

Figure 2.3: Expanding a node to a cycle

its edges, T' has the same weight as T''. For example, in Figure 2.3, the white node is expanded into a 4-cycle, and the dashed arrow is the edge that is removed after expanding.

Now we show $\mathrm{OPT}(G'') \leq \mathrm{OPT}(G')$. Suppose we have a min-weight arborescence T' of G'. After contracting some nodes in G' to obtain G'', if we look at the edges in T', they must still connect every node to the root. Therefore, we can remove some edges to create an arborescence of G''. For example, in Figure 2.4, we contract the two white nodes, and then remove edge b. Since edge weights are non-negative, we can only lower the cost by removing edges. Therefore $\mathrm{OPT}(G'') \leq \mathrm{OPT}(G')$. □

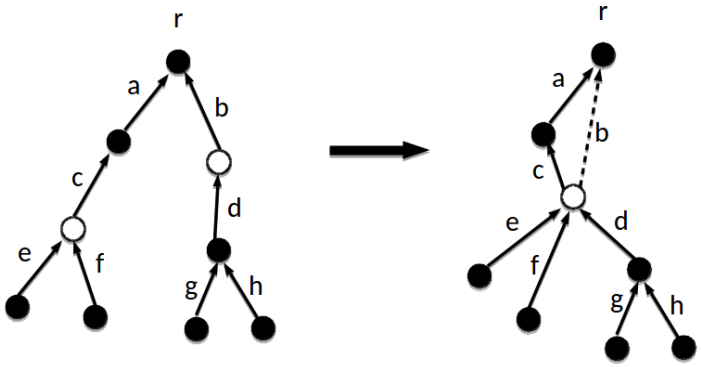

Figure 2.4: Contracting nodes in a cycle

The above proof gives an algorithm for finding the min-weight arborescence on G' given a min-weight arborescence on G'' by expanding the contracted cycle. Since G'' has strictly less vertices than G', we can now run the algorithm from the beginning on G'', and this inductively gives an algorithm for finding the min-weight arborescence on G. The runtime of the algorithm is $O(mn)$. Each contraction step takes $O(m)$ time and reduces the number of vertices by at least one. Because there are n vertices, there are at most n rounds.

Remark 2.8. This is not the best known run-time bound. Tarjan [Tar77] 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 [GGST86] give an algorithm to solve the min-cost 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. [**?**].

3 Linear Programming Methods

In this section, we will present an alternate proof of correctness of the Chu-Liu/Edmonds/Bock algorithm using linear program duality. This proof is based on Edmonds' original proof [Edm67].

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 2.9. For some number of parameters $n \in \mathbb{N}$, number of constraints $m \in \mathbb{N}$, objective vector $c \in \mathbb{R}^n$, constraints $A \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^m$, a *linear program* is

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

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*, a convex intersection of a finite number of half spaces—each half space corresponds to one constraint. If the polyhedron is bounded, we call it a *polytope*.

Definition 2.10. We say that a solution x is *feasible* if it satisfies the constraints: $Ax \geq b$ and $x \geq 0$.

Definition 2.11. 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. It tries to come up with a linear combination of constraints proving that the primal cannot possibly attain a certain value for $c^\intercal x$. This purpose is exemplified by the following theorem.

Theorem 2.12 (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.

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

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.

3.2 Arborescence Linear Program

Before we can analyze the algorithm, we first need to come up with a linear program for the min-cost arborescence problem. We are trying to find a set of edges forming an arborescence T, so we would like to have one variable x_e for each arc $e \in A$. We can think of these variables as being binary: $x_e \in \{0, 1\}$ and $x_e = 1$ if and only if $e \in B$. This choice of variables enables us to easily express our objective to minimize the total cost: $\sum_{e \in A} w_e x_e$.

Now, 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. We know that every vertex must be able to reach the root by a directed path. If $\partial^+ S \cap T = \emptyset$, then there is no edge leaving S. This is a problem because we have

to have some way to get to r. We conclude that, at a minimum, $\partial^+ S \cap T \neq \emptyset$. We can represent this constraint by ensuring that the number of edges out of S is non-zero.

$$\sum_{e \in \partial^+ S} x_e \geq 1$$

We represent our arborescence linear program as follows:

$$\begin{aligned} \text{minimize} & \sum_{e \in A} w_e x_e \\ \text{subject to} & \sum_{e \in \partial^+ S} x_e \geq 1 \quad \forall S \subseteq V - \{r\} \\ & x_e \geq 0 \quad \forall e \in A \end{aligned} \tag{2.1}$$

Lemma 2.13. *Let G be a rooted, weighted, directed graph. T is an r-arborescence of G with $x_e = \mathbf{1}_{e \in T}$ if and only if $x_e \in \{0, 1\}$, $\sum_{e \in A} x_e = n - 1$, and x is feasible for LP 2.1.*

Proof. Say we have some arborescence T. The vector x clearly satisfies the first two conditions, and satisfies $x \geq 0$. Now, consider some set $S \subseteq V - \{r\}$ and vertex $v \in S$. Find the first edge $e \in \partial^+ S$ on the path to the root from v. We know that $x_e = 1$, so $\sum_{e \in \partial^+ S} x_e \geq 1$. Thus, x is feasible.

Conversely, let x be a feasible solution. Because $\sum_{e \in A} x_e = n - 1$, we can construct the set of $n - 1$ edges contained in T. Suppose for contradiction that there exists some vertex v without a path to the root. Consider the set of vertices S reachable from v. By feasibility of x, we know that $\sum_{e \in \partial^+ S} x_e \geq 1$. We have found another edge leaving S, contradicting the definition of S. Thus, T must be an arborescence. □

3.3 Algorithm Analysis

Finally, we can show correctness of the Chu-Liu/Edmonds/Bock algorithm using the LP 2.1. It is clear the output T of the algorithm is an arborescence: every non-root vertex has exactly one outgoing edge and there are no cycles. Thus, the corresponding indicator vector x is feasible for the linear program. In order to show that x is optimal, we exhibit a vector y feasible for the dual 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 2.1 is

$$\begin{aligned} \text{maximize} & \sum_{S \subseteq V - \{r\}} y_S \\ \text{subject to} & \sum_{S : e \in \partial^+ S} y_S \leq w_e \quad \forall e \in A \\ & y_S \geq 0 \quad \forall S \subseteq V - \{r\} \end{aligned} \tag{2.2}$$

You can think of y_S like payments raised by vertices in S in order to buy an edge leaving S. In order to buy an edge e, we need to raise w_e dollars. Actually, we're trying to raise as much money as possible while not overpaying for any single edge e.

Proof. The proof of correctness will proceed slowly, building up solutions in the dual and the primal simultaneously as the algorithm executes. We will maintain that the dual is always feasible by never

overpaying for any edge, and we will maintain that the value of the dual and the primal are always equal. At the end of the algorithm, we arrive at a feasible primal solution. Because the dual is feasible as well and their objective values are equal, weak duality proves the solution is optimal.

Let d_e be the weight decrease for edge e. At the start of the algorithm, $d_e = 0$. We will increment d_e when we raise money.

$$d_e = \sum_{S : e \in \partial^+ S} y_S$$

If we denote the current weights used by the algorithm as $\hat{w}_e$, then the weight decrease should satisfy $\hat{w}_e = w_e - d_e$. See Figure 2.5 for how the payments are made to decrease all weights to 0.

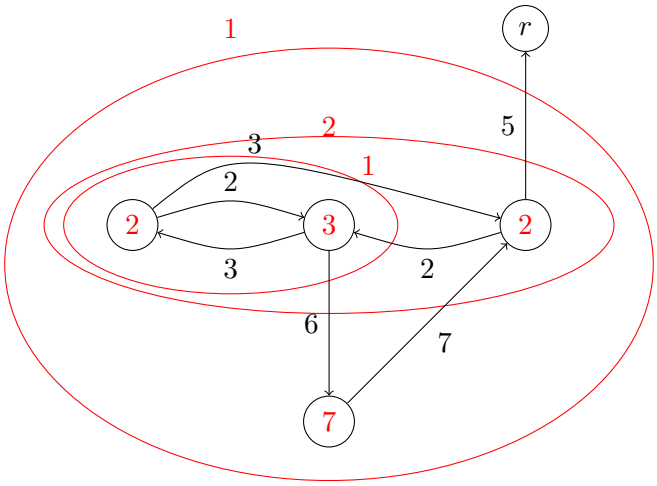

Figure 2.5: Vertex sets are labeled with their payments and edges are labeled with costs.

Step 1: Decreasing weights If we can, we pick a vertex v whose minimum outgoing edge m is non-zero and decrease the weight of all edges in $\partial^+ v$ by $M_v = \hat{w}_m$.

We've updated the values of $\hat{w}_e$, so we need to update d_e to maintain that $\hat{w}_e = w_e - d_e$. When performing the weight decrease, add M_v to $y_{\{v\}}$. Note that we will only ever update payments for singleton sets in this step. This will increase d_e for all $e \in \partial^+ v$, thereby keeping $\hat{w}_e = w_e - d_e$. Because we make sure that $\hat{w}_e \geq 0$ after this decrease, $d_e \leq w_e$ and the dual remains feasible.

Upon adding M_v to $y_{\{v\}}$, we increased the objective of the dual by M_v. We need to make a corresponding increase to the objective of the primal. We have not added to the payments of any non-singleton sets, so $d_m = 0$ before the decrease. Thus, $w'_m = w_m = M_v$ and incrementing x_m will add the necessary M_v to the objective of the primal.

Step 2: Contracting cycles When the algorithm finds a zero-weight cycle C, it contracts the cycle to a single vertex and recursively invokes itself. For the recursive invocation, we have new edge weights $w'_e = \hat{w}_e$ and cleared payments.

Now, suppose we have inductively built up a solution with a feasible dual y', a feasible, integral primal x', and equal objectives $\sum_{e \in A'} w'_e x'_e = \sum_{S \subseteq V' - \{r\}} y'_S$.

We can build up our solution by reassigning x_e in the way corresponding to how the algorithm constructs the resulting arborescence. This inserts the edges along C except for one (the one sharing the vertex with the edge out of C) and keeps all outside edges as they were in the result of the recursive call. Reassign the payments by combining the payments from before the recursive

call and after the recursive call.

$$y''_S = \begin{cases} y_S + y'_S & C \cap S = \emptyset \\ y'_{S-C+\{v_C\}} & C \cap S = C \\ y_S & \text{otherwise} \end{cases}$$

We don't need to add the y_S term if $C \subseteq S$ because we only paid for singleton sets before the recursive call (so $y_S = 0$). By constructions, this implies

$$d''_e = d_e + d'_e$$

and similarly

$$\sum_{S \subseteq V - \{r\}} y''_S = \sum_{S \subseteq V' - \{r\}} y'_S + \sum_{S \subseteq V - \{r\}} y_S$$

Because y' is feasible, we know that $d'_e \leq \hat{w}_e$. Then, $d''_e \leq d_e + \hat{w}_e = d_e + w_e - d_e = w_e$, and y'' is a feasible solution.

Let's split the total cost of the edges in the primal into the amount paid before the recursive call and the cost during the recursive call:

$$\sum_{e \in A} w_e x''_e = \sum_{e \in A} w'_e x''_e + \sum_{e \in A} d_e x''_e$$

We only make singleton-node payments before the recursive call. Because every node has exactly one edge leaving it before and after the recursive call, the d_e portion of the total cost stays the same. In particular,

$$\sum_{S \subseteq V - \{r\}} y_S = \sum_{e \in A} d_e x_e = \sum_{e \in A} d_e x''_e$$

We know that $w'_e = 0$ for all $e \in C$, so adding edges along C has no effect on the w'_e cost. Because we keep all of the other edges,

$$\sum_{e \in A} w'_e x''_e = \sum_{e \in A'} w'_e x'_e = \sum_{S \subseteq V' - \{r\}} y'_S$$

We conclude that the objectives are equal between the primal and the dual.

$$\sum_{e \in A} w_e x''_e = \sum_{S \subseteq V' - \{r\}} y'_S + \sum_{S \subseteq V - \{r\}} y_S = \sum_{S \subseteq V - \{r\}} y''_S$$

At the end, we have a feasible primal solution x, a feasible dual solution y, and equal objectives

$$\sum_{e \in A} w_e x_e = \sum_{S \subseteq V - \{r\}} y_S$$

By weak duality, we conclude that the solution x is optimal. □

3.4 Integrality of the Polytope

This result is actually very exciting: no matter the objective direction (i.e., arc weights w_e), we will always arrive at an optimal *integral* solution to the linear program. This tells us that every vertex (extreme point) of the arborescence polytope is integral! Indeed, we now show that the converse is also true, so there is a bijection between arborescences of the graph and the vertices of the polytope.

Lemma 2.14. *Let G be an rooted, weighted, directed graph. If T is an arborescence of G, then its corresponding vector $x_e = \mathbf{1}_{e \in T}$ is a vertex of the arborescence polytope 2.1.*

Proof. Choose weights $w_e = \mathbf{1}_{e \notin T}$. Clearly, the optimal solution is the one where $x_e = \mathbf{1}_{e \in T}$. In addition, moving any of the value from one of the edges within T to one of the edges outside clearly increasing the value. Thus, this solution is unique. We conclude that it must be a vertex of the polytope. □

Thus, the arborescence polytope is exactly the polytope that contains all of the arborescences of a graph as its vertices. We will see more about integral polytopes later in the course, when we talk about matchings in graphs.

Remark 2.15. Here we worked with non-negative weights to make our analysis easier. However, you can generalize the above results to work with general weights by including the extra condition in the LP that every vertex has to have exactly one edge out of it. This prevents the LP from giving huge values to edges with negative weights.

Acknowledgments

These lecture notes were scribed by Corwin de Boor, based on previous scribe notes of Yu Zhao and Xinyu Wu.

References

[Boc71] Frederick Bock. An algorithm to construct a minimum directed spanning tree in a directed network. *Developments in operations research*, pages 29–44, 1971. 1, 2

[CL65] Yoeng-jin Chu and Tseng-hong Liu. On the shortest arborescence of a directed graph. *Sci. Sinica*, 14:1396–1400, 1965. 1, 2

[Edm67] Jack Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards Sect. B*, 71B:233–240, 1967. 1, 2, 3

[GGST86] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986. Theory of computing (Singer Island, Fla., 1984). 2.8

[Kar72] R. M. Karp. A simple derivation of Edmonds' algorithm for optimum branching. *Networks*, 1:265–272, 1971/72. 2

[Tar77] R. E. Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977. 2.8