15-850: Advanced Algorithms **CMU, Spring 2017**
Lecture #2: Randomized MST and Directed MSTs January 27, 2017
Lecturer: Anupam Gupta Scribe: Yu Zhao, Xinyu Wu

1 Preliminaries

In this lecture we introduce the Karger-Klain-Tarjan randomized MST algorithm [KKT95] and Chu-Liu/Edmonds/Bock's algorithm for minimum weight arborescences [CL65, Edm67, Boc71]. Our setting is the same as last lecture. Let $G = (V, E)$ to be an undirected graph with vertex set V and edge set E, where $|V| = n$ and $|E| = m$. We also assume that all the weights of edges are distinct. In Lecture 1 we saw some deterministic algorithms finding MST in time $O(m \log n)$, $O(m+n \log n)$ and $O(m \log^* n)$. Our goal for the first part of this lecture is to present a randomized algorithm which finds a MST in the graph in $O(m + n)$ expected time.

1.1 Heavy & light edges

Recall the two rules mentioned in Lecture 1 which characterize MSTs.

Definition 2.1 (Cut Rule)**.** For any cut of the graph, the minimum-weight edge across the cut must be in the MST. This is also know as the Blue rule, and edges satisfying the blue rule will be colored blue.

Definition 2.2 (Cycle Rule)**.** For any cycle of the graph, the maximum-weight edge in the cut must not be in the MST. This is also know as the red rule, and edges satisfying the red rule will be colored red.

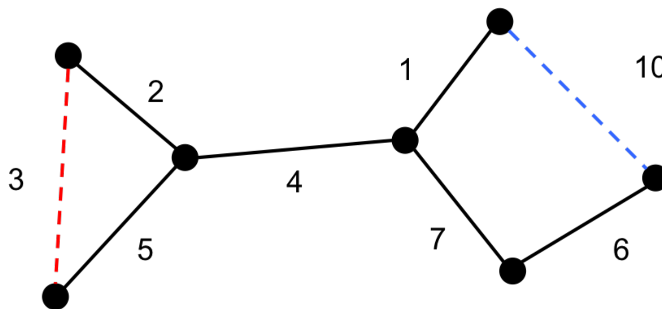

Figure 2.1: Example of heavy and light edges

Take any tree[1] in a graph. For example, in Figure 2.1 we pick the tree with black edges. The tree might not be an MST, or even not be connected (i.e. it is a forest). Now look at any other edge in the graph. The blue edge on the right is heavier than any other edge in the cycle. Then by the cycle rule this edge will not be in the MST of the graph. What about the red edge on the left? This edge is not the heaviest edge in the cycle, and in fact this edge appears in the MST of the graph. Therefore we can discard some edges which will never appear in the MST based on any given tree of a graph. This brings us to a definition.

[1] Here when we talk about trees, in fact we mean trees or forests. And MST refers to the minimum spanning forest if the graph is not connected.

Definition 2.3. Let T be a forest that is a subgraph of a graph G and $e \in E(G)$. If e creates a cycle when added to T and e is the heaviest edge in the cycle, then we say edge e is *T-heavy*. Otherwise, we say edge e is *T-light*.

Every edge is either T-heavy or T-light. Notice that if an edge e does not belong to a cycle, then e is T-light. If e is in the tree T, e is also T-light. In Figure 2.1, the red edge is T-light, the blue edge is T-heavy, and all the black edges are T-light.

Fact 2.4. *Edge e is T-light $\iff e \in MST(T \cup \{e\})$.*

Fact 2.5. *If T is a MST of G then edge $e \in E(G)$ is T-light $\iff e \in T$.*

Therefore our idea is that in order to pick a good tree/forest T, we find all the T-heavy edges and get rid of them. Hopefully the number of edges remaining is small. The MST must be in the remaining edges. To make this idea work, we want to find some tree T such that there are a lot of T-heavy edges.

1.2 MST Verification

How should we find all the T-heavy edges? Here we first assume a magic blackbox: MST Verification. This was skipped during lecture, but we'll show how to implement MST Verification later in Section 3.

Theorem 2.6 (MST Verification)**.** *Given a tree $T \subseteq G$, we can output all the T-light edges in $E(G)$ in time $O(|V| + |E|)$.*

With this, we can find all the T-light edges in linear time.

2 Karger-Klain-Tarjan Algorithm

Suppose we have a graph $G = (V, E)$ with n vertices and m edges.

Algorithm 1 KKT(G)

1: Run 3 rounds of Borůvka's Algorithm on G to get a graph $G' = (V', E')$ with $n' \leq n/8$ vertices and $m' \leq m$ edges.
2: $E_1 \leftarrow$ a random sample of edges E' of G' where each edge is picked independently with probability 1/2.
3: $T_1 \leftarrow$ KKT($G_1 = (V', E_1)$).
4: $E_2 \leftarrow$ all the T_1-light edges in E'.
5: $T_2 \leftarrow$ KKT($G_2 = (V', E_2)$).
6: Return T_2 (combine with the edges chosen in Step 1).

Here the idea is that we randomly choose half of the edges and find the MST on those edges. We hope this tree T will have a lot of T-heavy edges therefore we can discard these edges and find the MST on the remaining graph.

Theorem 2.7. *KKT algorithm will return MST(G).*

Proof. We proceed by induction. The base case is trivial, because we will find the MST in Step 1. Otherwise in Step 3 we find a MST T_1 of graph G_1, and in E_2 we discard all the T_1-heavy edges

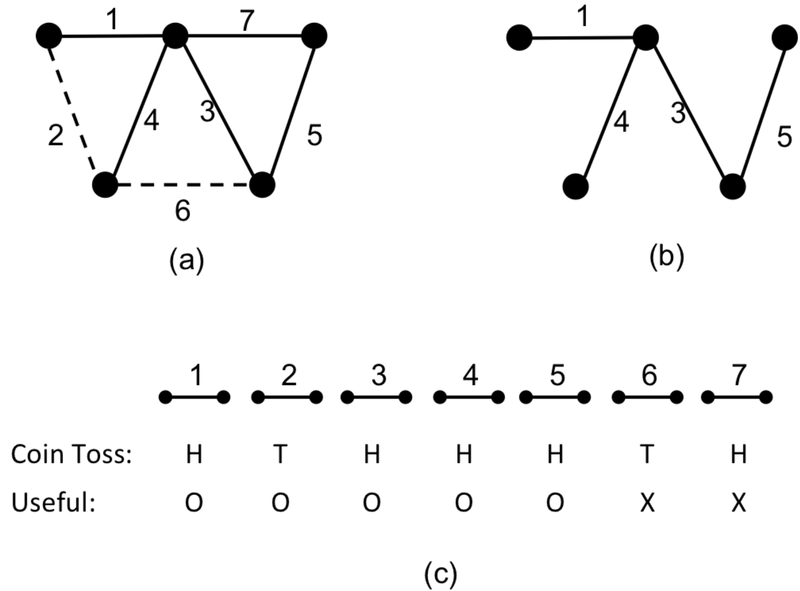

Figure 2.2: Illustration of another order of coin tossing

from E' (not only from E_1!!) which cannot possibly be in the MST of G' due to the cycle rule. Or in other words, $\text{MST}(G') \subseteq E_2$. Therefore what Step 5 returns is the MST of G_2, which is also the MST of G'. Combining it with the edges we chose in Step 1, we get the MST of graph G. □

The key idea of this proof is that discarding heavy edges of any tree in a graph will not change the MST.

Now we need to deal with the complexity.

Claim 2.8. $\mathbf{E}[\#E_1] = \frac{1}{2}m'$.

Proof. This claim is trivial since we pick each edge with probability 1/2. □

Claim 2.9. $\mathbf{E}[\#E_2] \leq 2n'$. *This means graph G_2 will hopefully be very sparse.*

How should we analyze $\mathbf{E}[\#E_2]$? In Step 2 of the KKG algorithm, we first flip a coin on each edge, then take all the edges where we got heads and finally in Step 3 we use KKG algorithm on this new graph to get the MST of G_1.

Let's think of this process in another order. The purpose of Step 3 is to calculate the MST of the new graph. It does not matter if we use the KKT algorithm or some other method – we will still get the same MST. So suppose we use Kruskal's algorithm instead. Then instead of first flipping coins for all edges then calculating the MST, we can do these two things simultaneously: We flip the coins for the edges in an increasing order by weight, and then only consider adding the edge to the graph as in Kruskal's algorithm if we get heads (see Algorithm 2).

We define a coin toss is *useful*, if Kruskal's algorithm will add the edge to the MST if we get heads, and define a coin toss is *useless* if not. For example, Figure 2.2(a) gives a graph with coin toss result. The solid edges are the edges where the coin toss resulted in heads, and the dashed edges are the edges which got tails. Figure 2.2(b) is the MST on all the edges where the coin toss was heads. Now let's check the usefulness of all these coin flips, as in Figure 2.2(c). In the beginning

Algorithm 2 ModifiedKruskals($G = (V, E)$)

1: Sort E in increasing order
2: **for all** $e \in E$ **do**
3: $p \leftarrow 0$ w.p. $\frac{1}{2}$, 1 otherwise
4: **if** $p = 0$ **then**
5: **if** e does not form a cycle **then**
6: Add e to MST
7: **end if**
8: **end if**
9: **end for**

the MST is an empty set. Then the coin flip of the first edge is useful, since it should be added into the MST. The result is heads, so we add this edge into MST. The coin flip of the second edge is useful, since it should be added into the MST. However the coin toss result is tails, so sadly we can not add this edge into MST. The coin flip of the third, forth, fifth edges are all useful, and the toss results are all heads, so we add all of them into the MST. The coin toss of sixth edge is useless, because we will not add this edge into MST regardless of the coin toss since it will create a cycle, and so is the seventh edge.

Claim 2.10. *$T_1 = MST(G_1)$. $e \in E'$ is T_1-light if and only if the coin flip of e is useful.*

Proof. If the coin flip of e is useful, then right before flipping the coin, we will not create a cycle by adding e to the current MST, which means either $e \in T_1$, or there is no cycle in $T_1 \cup \{e\}$, or edge e is not the heaviest edge in the cycle of $T_1 \cup \{e\}$. In any of these cases, edge e is T_1-light. On the other hand, if the coin flip of e is useless, then right before we flip the coin there would be a cycle if we add e into the MST, therefore edge e is T_1-heavy. □

Now we can prove Claim 2.9.

Proof of Claim 2.9.

$$\mathbf{E}[\#E_2] = \mathbf{E}[\#\text{useful coin flips}] \leq \frac{n' - 1}{1/2} \leq 2n'$$

The first inequality holds since by each useful coin flip we may add an edge into the MST of G_1 with probability $1/2$, and the final $T_1 = \text{MST}(G_1)$ has at most $n' - 1$ edges. This is then equivalent to flipping an unbiased coin until we get $n - 1$ heads. Therefore the expectation of the number of useful coin flips is at most $2(n' - 1) \leq 2n'$. □

Theorem 2.11. *$KKT(G = (V, E))$ can return the MST in time $O(m + n)$.*

Proof. Let X_G be the expect running time on graph G, and

$$X_{m,n} := \max_{G=(V,E),|V|=n,|E|=m} \{X_G\}$$

In the KKT algorithm, Step 1, 2, 4 and 6 will be done in linear time, so we assume the time cost of Step 1, 2, 4 and 6 is at most cm. Step 3 will spend time X_{G_1}, and Step 5 will spend time X_{G_2}. Then we have

$$X_G \leq cm + X_{G_1} + X_{G_2} \leq cm + X_{m_1,n'} + X_{m_2,n'}$$

Here we assume that $X_{m,n} \leq c(2m + n)$, then

$$\begin{aligned} X_G &= cm + \mathbf{E}[c(2m_1 + n')] + \mathbf{E}[c(2m_2 + n')] \\ &\leq c(m + m' + 6n') \\ &\leq c(2m + n) \end{aligned}$$

The first inequality holds because $\mathbf{E}[m_1] \leq \frac{1}{2}m'$ and $\mathbf{E}[m_2] \leq 2n'$. The second inequality holds because $n' \leq n/8$ and $m' \leq m$. □

3 MST Verification

Now we come back to the implementation of the MST verification blackbox. Here we only consider only trees (not forests), since we can run this algorithm on each connected component separately. Here we refine Theorem 2.6 as follows.

Theorem 2.12 (MST Verification). *Given $T = (V, E)$ where $|V| = n$ and m pairs of vertices (u_i, v_i), we can find the heaviest edge on the path in T from u_i to v_i for all i in $O(m + n)$ time.*

Dixon, Rauch and Tarjan first showed that MST verification was achievable in $O(m + n)$ time. Komlos [Kom85] shows how to do it with $O(m + n)$ comparisons. King [Kin97] showed how to achieve linear time on a RAM machine, and Hagerup [Hag10] presents a simpler linear time algorithm. Here we will present the algorithm of Komlos.

3.1 Balance the tree

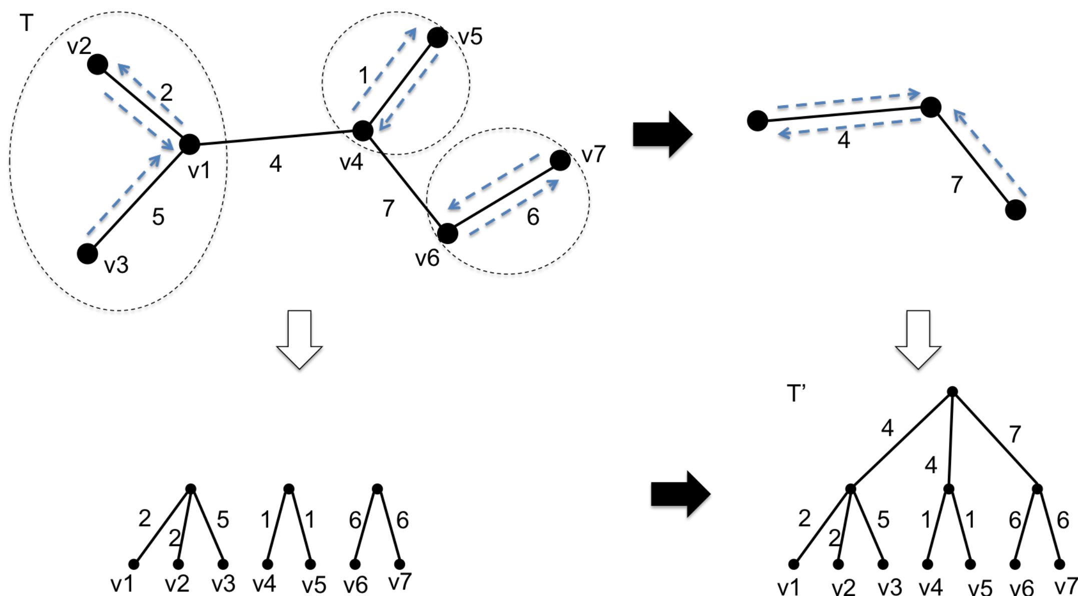

Figure 2.3: Illustration of balancing a tree

Suppose $T = (V, E)$ is a tree on n vertices, and we run Boruvka's algorithm on T. (Let $V_1 = V$ be the original vertices at the beginning of round 1, V_i be the vertices in round i, and say there are L rounds so that $|V_{L+1}| = 1$). We build a tree T' as follows: the vertices are the union of all the V_i. There is an edge from $v \in V_i$ to $w \in V_{i+1}$ if the vertex v belongs to a component in round i

and that is contracted to form $w \in V_{i+1}$; the weight of this edge (v, w) is the min-weight edge out of v in round i. (Note that all vertices in V are now leaves in T'.) Figure 2.3 shows an example of balancing a tree.

Fact 2.13. *For nodes u, v in a tree T, let $\texttt{maxwt}_T(u, v)$ be the maximum weight of an edge on the (unique) path between u, v in the tree T. For all $u, v \in V$, we have*

$$\texttt{maxwt}_T(u, v) = \texttt{maxwt}_{T'}(u, v)$$

This is a problem in Homework 1. In Figure 2.3, we have $\texttt{maxwt}_T(v_1, v_7)$ is 7 which is the weight of edge (v_4, v_6). We can check that $\texttt{maxwt}_{T'}(v_1, v_7)$ is also 7.

Fact 2.14. *T' has at most $\log_2 n$ layers since each vertex has at least 2 children and all leaves of T' are at the bottom (the same) level.*

Here we can make a balanced tree T' based on tree T with some good properties. Then we can just query the heaviest edges of vertex pairs in tree T' instead of T. Another trick is that we can assume that all queries are ancestor-descent queries if we can find the least common ancestor quickly.

Theorem 2.15 (Harel-Tarjan)**.** *Given a tree T, we can preprocess in $O(n)$ time, and answer all LCA queries in $O(1)$ time.*

This was shown by Harel and Tarjan in [HT84].

3.2 Get the answer

Now we have reduced our question to how to answer the ancestor-descent queries efficiently. For each edge $e = (u, v)$ where v is the parent of u, we will look at all queries starting in subtree T_u and ending above vertex v. Say those queries go to $w_1, w_2, \ldots, w_k$. Then the "query string" is $Q_e = (w_1, w_2, \ldots, w_k)$. Then we need to calculate the "answer string" $A_e = (a_1, a_2, \cdots, a_k)$ where a_i is the maximal weight among the edges between w_i and u.

Figure 2.4 gives us an example. Suppose $Q_{(b,a)} = (w_1, w_3, w_4)$, which means there are three queries starting from some vertices in the subtree of b and ending to w_1, w_3, w_4. Then we get the answer

$$A_{(b,a)} = (a_1, a_3, a_4) = (6, 4, 4)$$

since the maximal weight of an edge on the path from w_1 to b is the weight of edge (w_1, w_2), and the maximal weight of an edge on the path from either w_3, w_4 to b is from the weight of edge (w_3, w_4).

Given the answer of $A_{(b,a)}$, how should we find the answer of $A_{(c,b)}$ efficiently? Say $Q_{(c,b)} = (w_1, w_4, b)$. Comparing with $Q_{(b,a)}$, we may lose some queries since they are from some other children of vertex b, and we may have some other queries ending at b which will not contain in $Q_{(b,a)}$. The easiest way is to update every query. Suppose the weight of edge (c, b) is t. Then we can calculate

$$A_{(c,b)} = (\max\{a_1, t\}, \max\{a_4, t\}, t) = (\max\{6, 5\}, \max\{4, 5\}, 5)$$

The trick is that if in $Q_e = (w_1, w_2, \ldots, w_k)$, $w_1, w_2, \ldots, w_k$ is sorted from the top to the bottom, then the answers $A_e = (a_1, a_2, \cdots, a_k)$ should be non-increasing, $a_1 \geq a_2 \geq \cdots \geq a_k$. Therefore we can do binary search to reduce the number of comparisons.

Claim 2.16. *Given the question string Q_e for $e = (u, v)$ where v is the parent of u, the answer string A_e for e , we can compute answers $A_{e'}$ for $e' = (w, u)$ where w is a child of u, within time $\lceil \log(|A_e| + 1) \rceil$.*

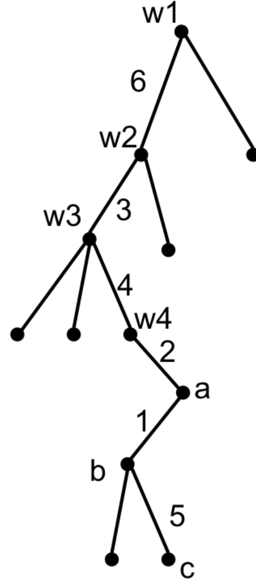

Figure 2.4: Illustration of queries and answers

Theorem 2.17. *The total number of comparisons for all queries*

$$t \leq \sum_{e} \log\left(|Q_e| + 1\right) \leq O(m+n)$$

Proof. Assume the number of edges in level i is n_i. Here the level is counted from the bottom to the top, the edges connected to leaves are at level 0.

$$\begin{aligned}
\sum_{e \in \text{level } i} \log_2(1 + |Q_e|) &= n_i \operatorname*{avg}_{e \in \text{level } i} (\log_2(1 + |Q_e|)) \\
&\leq n_i \log_2\left(1 + \operatorname*{avg}_{e \in \text{level } i} (|Q_e|)\right) \\
&= n_i \log_2\left(1 + \frac{\sum_{e \in \text{level } i} |Q_e|}{n_i}\right) \\
&\leq n_i \log_2\left(1 + \frac{m}{n_i}\right) \\
&= n_i \left(\log_2 \frac{m+n}{n} + \log_2 \frac{n}{n_i}\right)
\end{aligned}$$

The first inequality holds by Jensen's inequality and convexity of the function $\log_2(1+x)$. The second inequality holds since the number of all queries is m and each query will only appear on at most one edge on any particular level. Therefore we have

$$
\begin{aligned}
t &= \sum_e \log_2(1+|Q_e|) \\
&= \sum_i \sum_{e \in \text{level } i} \log_2(1+|Q_e|) \\
&\leq \sum_i n_i \left(\log_2 \frac{m+n}{n} + \log_2 \frac{n}{n_i} \right) \\
&= n \log_2 \frac{m+n}{n} + \sum_i n_i \log_2 \frac{n}{n_i} \\
&\leq n \log_2 \frac{m+n}{n} + cn \\
&= O(n \log \frac{m+n}{n} + n) = O(m+n) \qquad \square
\end{aligned}
$$

4 Minimum Cost Arborescence

We've now finished with our discussion of minimum spanning trees on undirected graphs. In the homework, we also see that undirected MSTs are a special case of matroids. Today, we consider another generalization, in which the graph is directed.

Consider the setting where we have a graph $G = (V, A)$, with V a set of vertices, and A a set of directed arcs (an arc is a directed edge in a graph, and will be used interchangeably). The graph is rooted at some root $r \in V$.

Definition 2.18. An r-arborescence is a collection of arcs $B \subseteq A$ such that

1. Each vertex has 1 outgoing arc, except r
2. There exists a directed path from each vertex to r

Remark 2.19. It's easy to check if an r-arborescence exists. We can simply reverse the edges, and run a depth-first search from the root, and check that all the vertices are reached.

Next, we assign each arc $a \in A$ some non-negative weight w_a (otherwise, we can add a large weight M to each arc to make them nonnegative). We now want to find the minimum weight r-arborescence. We further assume that r has no outgoing edges, since they will not be part of the min-weight arborescence.

Towards finding an algorithm, 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 2.5. 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 2.20. For a vertex $v \in V$, let $\partial^+ v$ denote the set of arcs leaving v.

Definition 2.21. For a vertex $v \in V$, let $M_v = \min_{a \in \partial^+ v} w_a$.

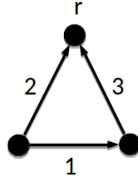

Figure 2.5: An example where the greedy algorithm fails

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

Proof. This is easy to see from the above. □

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 2.6).

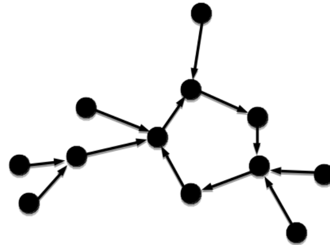

Figure 2.6: An example of a possible component after running the first step of the algorithm

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 2.23. *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 its edges, T' has the same weight as T''. For example, in Figure 2.7, 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

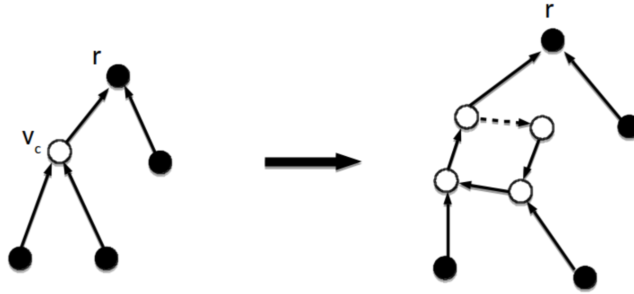

Figure 2.7: Expanding a node to a cycle

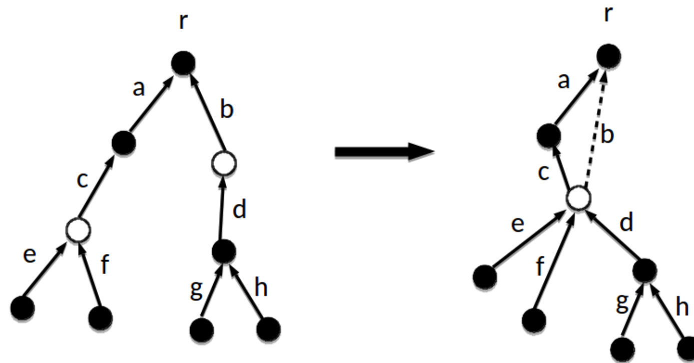

Figure 2.8: contracting nodes in a cycle

every node to the root. Therefore, we can remove some edges to create an arborescence of G''. For example, in Figure 2.8, 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 $\mathrm{OPT}(G'') \leq \mathrm{OPT}(G')$. □

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 $O(mn)$ since each contraction step takes $O(m)$ time, and each contraction reduces the number of vertices by at least one, so there are at most n rounds.

This is not the best known 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.

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, 4

[CL65] Yoeng-jin Chu and Tseng-hong Liu. On the shortest arborescence of a directed graph. *Sci. Sinica*, 14:1396–1400, 1965. 1, 4

[Edm67] Jack Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards Sect. B*, 71B:233–240, 1967. 1, 4

[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). 4

[Hag10] Torben Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In *Graph-theoretic concepts in computer science*, volume 5911 of *Lecture Notes in Comput. Sci.*, pages 178–189. Springer, Berlin, 2010. 3

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. 3.1

[Kar72] R. M. Karp. A simple derivation of Edmonds' algorithm for optimum branching. *Networks*, 1:265–272, 1971/72. 4

[Kin97] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997. 3

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. Assoc. Comput. Mach.*, 42(2):321–328, 1995. 1

[Kom85] János Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985. 3

[Tar77] R. E. Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977. 4