Optimum Branchings and Spanning Aborescences

Ali Tofigh

September 23, 2009

1 Introduction

A branching in a directed graph $G = \langle V, E \rangle$ is defined as a set of edges $B \subseteq E$ such that

1. B does not contain a cycle, and
2. no two edges of B enter the same vertex, i.e., if (u, v) and (u', v') are distinct edges of B, then $v \neq v'$.

A vertex v of G is a root of B if no edge in B is directed towards v. Notice that in our terminoligy, if $B = \emptyset$, then B is a branching of G in which every vertex of G is a root. Clearly, every branching has at least one root. A branching that has exactly one root, is sometimes called a spanning arborescence. A spanning arborescence is, of course, nothing other than a rooted tree. In tree terminolgy, a branching is a forest of rooted trees.

Given a weight function $w : E \to \mathbb{R}$ on the edges of G, the weight of a subset $E' \subseteq E$ is defined as the sum of the weights of the edges in E':

$$w(E') = \sum_{e \in E'} w(e).$$

An optimum branching is then defined as a branching with optimum weight among all possible branchings for G. Depending on context, optimum can mean either minimum or maximum. Without loss of generality, we will confine our discussion to maximum branchings.

Polynomial time algorithms for finding optimum branchings were independently discovered by Edmonds [5], Chu and Liu [4], and Bock [1]. These algorithms are all based on the same method. The correctness proof in [5] is based on concepts in linear programming and Karp gave a simplified, purely combinatorial proof in [7]. Tarjan described an efficient implementation of Edmonds's algorithm in [8]. The algorithm can be implemented to run in time $O(m \log n)$, where n is the number of vertices of the graph and m is the number of edges. With a slight modification, the implementation can

be made to run in time $O(n^2)$, which is preferable when dealing with dense graphs. Camerini et. al. [2] corrected an error in [8]. In a later paper [3], the same authors give an implementation of an algorithm for finding the K best spanning arborescences, i.e., maximum branchings with exactly one root. The root, however, must be specified beforehand. In [6], Gabow et. al. give an $O(n \log n + m)$ time implementation of Edmonds's algorithm for finding an optimum spanning arborescence. They note that the algorithm can be modified to either use a specified vertex as root, or to find a best root vertex by itself. The algorithm in [6] uses F-heaps as a main data structure. The authors also note that a better time complexity cannot be achieved by any implementation of Edmonds's algorithm since Edmonds's algorithm can be used to sort n numbers (sorting n numbers by comparison requires $\Omega(n \log n)$ time, and since we always have to look at every edge of the graph, we cannot achieve a better time complexity for Edmonds's algorithm than $O(n \log n + m)$.

In this document we will review Karp's proof of the correctness of Edmonds's algorithm and restate Tarjan's implementation. We will also discuss how slight modifications of Tarjan's algorithm can produce branchings with different criteria:

1. An optimum branching with arbitrary roots (this is the original version)
2. An optimum branching with a prespecified set of roots
3. An optimum branching with one unspecified root.

2 Karp's Derivation of Edmonds's Algorithm

In this section we will review Karp's derivation of Edmonds's algorithm. In the next section we will discuss Tarjan's implementation.

We will first give a thorough description of how a maximum spanning arborescence (MSA) with an arbitrary root can be obtained, after which we will describe the changes needed to obtain either a maximum spanning arborescence with a prespecified root, or a branching with arbitrary roots.

Let $G = \langle V, E \rangle$ and $w : E \to \mathbb{R}$ be given. In the most general setting, a graph may have several edges between the same two vertices; therefore, we will assume that E is a multiset. Furthermore, we will assume that the weight function w assigns weights to each occurrence of an edge independently. So, although the fact that E is a multiset will not be made explicit in our notation, we assume that any implementation of the algorithm treats each edge between the same two vertices as distinct elements. Achieving this in any standard programming language is, of course, trivial. We also assume that there are no loops in E, that is $(u, u) \notin E$ for any vertex $u \in V$.

The following is some of the notation that we will use. If $e = (u, v)$ is an edge, then $\text{tail}(e) = u$ and $\text{head}(e) = v$. A path is a sequence of distinct edges $e_1, \ldots, e_k$ such that $\text{head}(e_i) = \text{tail}(e_{i+1})$ for $i = 1, \ldots, k-1$. A path is called a cycle if, in addition, $\text{head}(e_k) = \text{tail}(e_1)$. If $F \subseteq E$, we let $u \underset{F}{\rightarrow} v$ mean that there is a path in F from u to v, i.e., there is a path $e_1, \ldots, e_k$ in F such that $\text{tail}(e_1) = u$ and $\text{head}(e_k) = v$.

Assume that there exists a MSA in G. Call an edge $e = (u, v)$ *critical* if $w(e)$ is maximal among all edges coming in to v. A set $H \subseteq E$ is called a *critical graph* if:

1. each edge of H is critical, and
2. no two edges of H are directed towards the same vertex.

A critical graph is maximal if it is not a proper subset of another critical graph. Note that in order to obtain a maximal critical graph we only need to choose one maxium-weight incoming edge for each vertex of G with indegree > 0.

Lemma 1. *If H is a maximal critical graph that contains no cycles, then H is a MSA.*

Proof. If H does not contain a cycle, then H is a branching. Any branching has at least one root, so H has at least one root. By our assumption that there exists an MSA in G, at most one vertex of G has indegree zero, and therefore, by the maximality of H, H has at most one root. Hence, H has exactly one root and is a spanning arborescence. To see that H is a maximum spanning arborescence, simply note that for each vertex $v \in V$, and for any branching B of G,

$$w(\{(u, v) \in H\}) \geq w(\{(u', v) \in B\}).$$

Summing over all vertices v in G, we have that

$$w(H) \geq w(B).$$

□

Lemma 2. *Each edge of a critical graph H is in at most one cycle of H.*

Proof. By definition, a critical graph H has at most one edge directed towards each vertex, and therefore, for each edge $e \in H$, there is a unique maximal path $e_1, \ldots, e_n$ in H such that $e_n = e$. Hence, if e is part of a cycle, then the cycle must consist of exactly the edges $e_1, \ldots, e_n$. □

Lemma 3. *Let B be a branching and let u, v, and w be distinct vertices of G. If $u \underset{B}{\rightarrow} w$ and $v \underset{B}{\rightarrow} w$, then either $u \underset{B}{\rightarrow} v \underset{B}{\rightarrow} w$ or $v \underset{B}{\rightarrow} u \underset{B}{\rightarrow} w$.*

Proof. Assume that $u \underset{B}{\rightarrow} w$ and $v \underset{B}{\rightarrow} w$. Since B is a branching, B has at most one edge directed towards each vertex, and therefore, there is a unique maximal path in B ending with w. Clearly, both u and v are in this path. Hence, either $u \underset{B}{\rightarrow} v \underset{B}{\rightarrow} w$ or $v \underset{B}{\rightarrow} u \underset{w}{\rightarrow}$. □

If B is a branching and e is an edge not in B, then e is called *eligible* with respect to B if and only if

$$B' = B \cup \{e\} - \{f \in B : \text{head}(f) = \text{head}(e)\}$$

is a branching. Intuitively, an edge (u, v) is eligible with respect to a branching B if the set obtained from B by adding (u, v) and removing any other edge directed towards v is a branching. Clearly, an edge e is eligible with respect to B if and only if adding e to B does not create a cycle.

Lemma 4. *Let B be a branching and let $C \subseteq E$ be a cycle in G. If $|C - B| \geq 2$, then at least one of the edges in $C - B$ is eligible with respect to B.*

Proof. Assume that $|C - B| \geq 2$ and that no edge of $C - B$ is eligible with respect to B. Let e and e' be edges of $C - B$ such that $\text{head}(e) \underset{C \cap B}{\rightarrow} \text{tail}(e')$, i.e., e and e' are edges of $C - B$ such that every edge in C between e and e' is in B. We will show that there is a path from $\text{head}(e')$ to $\text{head}(e)$ in B.

Let $e = (u, v)$ and $e' = (w, z)$. Since e' is not eligible with respect to B, adding e' to B creates a cycle; in other words, $z \underset{B}{\rightarrow} w$. If $v = w$, then we are done. Assume instead that $v \neq w$. Clearly, $v \underset{B}{\rightarrow} w$. By Lemma 3, either $v \underset{B}{\rightarrow} z \underset{B}{\rightarrow} w$ or $z \underset{B}{\rightarrow} v \underset{B}{\rightarrow} w$. But clearly, z is not on the path in B from v to w. Hence, we must have that $z \underset{B}{\rightarrow} v \underset{B}{\rightarrow} w$. This proves that $\text{head}(e') \underset{B}{\rightarrow} \text{head}(e)$.

Now, let $e_1, \ldots, e_k$ be the edges of $C - B$ in the order of their appearance in the cycle, i.e., $\text{head}(e_i) \underset{C \cap B}{\rightarrow} \text{tail}(e_{i+1})$ for $i = 1, \ldots, k-1$, and $\text{head}(e_k) \underset{C \cap B}{\rightarrow} \text{tail}(e_1)$. Applying our previous result to all consecutive pairs of edges in the sequence $e_1, \ldots, e_k$, we see that

$$\text{head}(e_{i+1}) \underset{B}{\rightarrow} \text{head}(e_i)$$

for $i = 2, \ldots, k$ and

$$\text{head}(e_1) \underset{B}{\rightarrow} \text{head}(e_k).$$

This demonstrates a cycle in B which contradicts the assumption of the lemma. Hence, at least one of the edges in $C - B$ is eligible with respect to B. □

Theorem 1. *Let H be a maximal critical graph of G. Then there exists a MSA B of G such that for each cycle C in H, $|C - B| = 1$.*

Proof. Let C be a cycle of H and let B be an MSA of G such that $|C-B| \geq 2$. By Lemma 4, there is an edge e in $C - B$ that is eligible with respect to B. Let B' be the branching

$$B' = B \cup \{e\} - \{f \in B : \text{head}(f) = \text{head}(e)\}$$

Clearly, the weight of B' is no less than that of B (remember that e is a critical edge). It is also easy to see that B' must be a spanning arborescence: we have simply moved the subtree rooted at head(e) (head(e) cannot be the root of B since e would not be eligible in that case). Hence, B' is a maximum spanning arborescence containing more edges of C compared to B. By repeated application of Lemma 4, we can obtain a maximum spanning arborescence such that $|C - B| = 1$. □

For a critical graph H of G, we say that a branching B of G is H-constrained if and only if $|C - B| = 1$ for each cycle C of H.

Let H be a maximal critical graph of G and let $C_1, \ldots, C_k$ be the (disjoint) cycles of H. Let V_i be the vertices of cycle C_i. By the previous theorem, there is an H-constrained MSA of G. We will now construct a graph $\hat{G}$ that has fewer edges and vertices than G, such that we can easily obtain an MSA of G given an MSA of $\hat{G}$.

Let $\hat{G} = \langle \hat{V}, \hat{E} \rangle$. We want to contract each cycle of H into one vertex and change the weights of edges coming into this supervertex. To this end we will remove from V all vertices in $V_1, \ldots, V_k$, and add as vertices the sets $V_1, \ldots, V_k$ themselves:

$$\hat{V} = (V - \bigcup_{i=1}^{k} V_i) \cup \{V_1, \ldots, V_k\}.$$

Corresponding to each vertex in V, there is a unique vertex in $\hat{V}$ defined by the function $f : V \to \hat{V}$

$$f(v) = \begin{cases} V_i & \text{if } v \in V_i \text{ for some } i = 1, \ldots, k, \\ v & \text{otherwise.} \end{cases}$$

We now define the edges of $\hat{G}$:

$$\hat{E} = \big\{\big(f(u), f(v)\big) : (u, v) \in E\big\} - \{(u, u) : u \in \hat{V}\}.$$

For each edge $\hat{e} \in \hat{E}$ we can assign a unique corresponding edge $e \in E$ such that $\hat{e} = \big(f(\text{tail}(e)), f(\text{head}(e))\big)$. Let $g : \hat{E} \to E$ be any such assignment.

Let e_i^0 be an edge of C_i with minimum weight among the edges of C_i. We define the weights on the edges of $\hat{G}$ as follows:

$$\hat{w}(\hat{e}) = \begin{cases} w(g(\hat{e})) - w(c_i(g(\hat{e}))) + w(e_i^0) & \text{if head}(g(\hat{e})) = V_i,\ i = 1, \ldots, k \\ w(g(\hat{e})) & \text{otherwise,} \end{cases}$$

where $c_i(e)$ is the unique edge of C_i that is directed towards the same vertex as e. This ends our construction of a new simpler instance of MSA. We will now prove that finding an MSA in $\hat{G}$ is equivalent to finding an MSA in G.

Theorem 2. *Let $\hat{B}$ be an MSA of $\hat{G}$. Let $D = \{g(\hat{e}) : \hat{e} \in \hat{B}\}$. If there is an edge (u, v) in D such that $v \in V_i$, then let f_i be the unique edge of C_i that is directed towards v. Otherwise, let $f_i = e_i^0$. The set B defined as*

$$B = D \cup \bigcup_{i=1}^{k} (C_i - f_i),$$

is an H-constrained MSA of G.

Proof. Clearly, B is an H-constrained spanning arborescence of G. We need to show that B is also maximum. Assume not. Let X be an H-constrained MSA of G. Note that Theorem 2 guarantees the existance of X. Since X is maxium but not B, we have that $w(X) > w(B)$. Let

$$\hat{X} = \{\hat{e} \in \hat{E} : g(\hat{e}) \in X\}.$$

We will show that $\hat{w}(\hat{X}) > \hat{w}(\hat{B})$ contradicting the optimality of $\hat{B}$.

Now $\hat{X}$ does not have any cycles, since that would imply cycles in X. Also, since X is H-constrained, at most one edge of X is directed towards any cycle of H. Hence, no two distinct edges of $\hat{X}$ are directed towards the same vertex in $\hat{V}$. Thus, $\hat{X}$ is a branching. If $\hat{X}$ has more than one root, then so does X. Hence, $\hat{X}$ is a spanning arborescence of $\hat{G}$.

By the definition $\hat{w}$, it is fairly simple to verify that

$$\hat{w}(\hat{B}) = w(B) - \sum_{i=1}^{k} w(C_i) + \sum_{i=1}^{k} w(e_i^0).$$

Note that the above equation applies even if the root of $\hat{B}$ is a cycle of H. Similarly, we have that

$$\hat{w}(\hat{X}) = w(X) - \sum_{i=1}^{k} w(C_i) + \sum_{i=1}^{k} w(e_i^0).$$

Since $w(x) > w(B)$, we get

$$\hat{w}(\hat{X}) - \hat{w}(\hat{B}) = w(X) - w(B) > 0.$$

Hence, $\hat{w}(\hat{X}) > \hat{w}(\hat{B})$ which is a contradition. □

2.1 Variations

If instead of an arbitrary root, we wish to find an MSA of G rooted at a specific vertex r, then we can proceed as follows. Simply remove from G any edges directed towards r. Assuming that there is an MSA in G rooted at r, then clearly, the algorithm described above will find it. More generally, assume that we wish to find a maximum branching with a set of specified root R. Assuming that there is such a branching, then we can remove the edges directed towards any vertex in R and apply Edmonds's algorithm. The only results that need to be modified are Lemma 1, Theorem 1, and Theorem 2. The modifications are trivial.

To obtain a maximum branching with arbitrary roots, we only need to change the definition of a critical edge: we call an edge $e = (u, v)$ critical if $w(e)$ is maximal among all edges coming in to v *and* $w(e) > 0$. Again, with this modified definition, the corresponding modified results of the previous section can easily be proved.

In terms of implementations of algorithms, if a set of roots is specified, then simply do not include in H edges coming into a root. If a branching with arbitrary roots is desired rather than an MSA with an arbitrary root, simply do not include in H any edges whose weight is zero or negative.

3 Tarjan's Implementation

This section contains pseudocode for Tarjan's implementation of Edmonds's algorithm. The variant of the code given here finds an MSA with an arbitrary root assuming that there is at least one spanning arborescence in the input graph. This code uses Tarjan's modification for dense graphs to achieve a running time of $O(n^2)$, and incorporates the corrections given by Camerini et. al. in [2]. See the pseudocode at the end of this section.

The algorithm works by attempting to build a maximal critical graph by arbitrarily choosing one of the maximum weight incoming edges of each vertex. However, as soon a cycle is formed, the cycle is contracted and treated as a single vertex. To keep track of the vertices and supervertices, as well as being able to detect cycles efficiently, two disjoint set datastructures are used.

Disjoint sets are datastructures on which three operations are defined: makeset, find, union. The operation `makeset(i)` creates a set containing i as the sole member. In this case i will also be the representative of the set. The operation `find(i)` will return the representative of the set containing i. Finally, `union(i, j)` will destroy the sets represented by i and j and create a new set containing the elements of both sets. The representative of the new set will be some member of the new set. A sequence of m operations on such a datastructure takes time $O(m\alpha(m))$, where $\alpha(m)$ is the inverse of the extremely quickly-growing ackermann function $A(m, m)$. The value

of $\alpha(m)$ is less than 5 for all remotely practical purposes and can thus be viewed as a constant.

A datastructure S is used to keep track of all current supervertices (these were referred to as strongly connected components in [8]). When a cycle has been created, the sets corresponding to the vertices of the cycle are unioned into one component in S. Also, a disjoint set datastructure called W is used to keep track of the, so called, weakly connected components. Whenever an edge is added to the critical graph, either a cycle has been created, or two distinct connected components of the critical graph have been joined together to one connected component. Each vertex of G belongs to one weakly connected component in W. Hence, the addition of an edge (u, v) into the critical graph creates a cycle if and only if u and v belong to the same weakly connected component.

We use a rooted forrest of edges of G to keep track of the edges currently in the critical graph. Each time an edge is conceptually chosen as an edge in the critical graph, it is also added to F. If the edge points into a set of S that consists of more than one vertex of G, i.e., the set in S corresponds to a cycle, then the edges in the cycle become the children of the newly added edge in F. In this way, the roots of F are the edges of the current critical graph of the graph whose vertices are the sets in S.

When extending the critical graph we need to find a maximum weight incoming edge of our chosen vertex v. To this end, we keep a list $I[v]$ that holds all incoming edges of v. The list is sorted on the tails of the edges and contains at most one edge from any vertex, i.e., if $(u, v) \in I[v]$ and $(u', v) \in I[v]$ are distinct edges, then $u \neq u'$. This allows us to find a maximum weight incoming edge of v in time $O(|V|)$ and also allows for efficient join operation of two lists. Whenever two or more sets in S are unioned into one set, then the corresponding lists of incoming edges must also be joined into one list.

Other variables and datastructures in the algorithm are:

roots contains the strongly connected components that are roots in the current critical graph, i.e., no incoming edge has been chosen for these vertices/supervertices.

min keeps track of the possible roots of the MSA. If a supervertex represented by v has been chosen as root at the end of the algorithm, then min$[v]$ is the vertex of G that will be the root of the MSA.

root is the vertex that has been identified as the final root in the critical graph. Note that this may be a supervertex, in which case min$[root]$ is the vertex of G that is chosen as the root of the MSA.

enter keeps track of the incoming edges of the strongly connected components in the critical graph. If v is the representative of the strongly

connected component S_v, then `enter[v]` is the edge of the critical graph that points into S_v (or $\emptyset$ if no such edge exists).

λ is an array of pointers, pointing to leaves of F. More specifically, if $v \in V$, then $\lambda[v]$ is the leaf of F whose head is v. If no such leaf exists, then $\lambda[v]$ is $\emptyset$.

cycle contains the edges comprising the cycles of the strongly connected components of the critical graph.

As for the time complexity, we can reason informally as follows. During the execution of the first part of the algorithm, i.e., while we are adding edges to the critical graph, we will create $O(n)$ strongly connected components. Therefore, the first while-loop will be executed at most $O(n)$ times. Finding a maximum weight incoming edge on line 16 takes $O(n)$. Finding a cycle on line 25 takes $O(n)$ since the critical graph never contains more than $O(n)$ edges. We can subract a number from all edges in a list $I[v]$ on line 31 by storing the subtracted value with $I[v]$. We can then use this to subtract the actual weights from the edges when merging lists. So this takes $O(1)$. There are at most $O(n)$ union operations in total on line 34 during the entire execution of the algorithm (again, since at moste $O(n)$ strongly connected components are created). By the same reasoning, no more than $O(n)$ list merges are performed during the entire algorithm on line 39, where each merge takes $O(n)$ time. Hence, the first part of the algorithm takes $O(n)$ time. It is easy to verify that the second part of the algorithm, where the expansion takes place, takes time $O(n)$. Hence, the entire algorithm takes time $O(n^2)$.

For sparse graphs a time complexity of $O(m \log n)$ is preferable. This can be achieved by changing the datastructure used to keep track of the incoming edges of each vertex, i.e., lists, to priority queues. We will need an implementation of priority queues that allows merging of distinc queues in time $O(logn)$ where n is the total number of elements in the two queues.

It is easy to modify the algorithm to accomodate the different variations. To obtain an MSA rooted at a specified vertex ρ, just initialize **root** to ρ and remove ρ from **roots**. This has the same effect as if ρ had no incoming edges at all. To obtain a maximum branching with a specified set of roots, let **root** be the set of specified roots instead and remove these from **roots** in the beginning of the algorithm. The if-statement on line 43 then becomes a while loop instead. In both these cases, the array **min** can be dispensed with. Finally, to obtain a maximum branching with arbitrary roots, only add edges to F if the current cost is positive. Line 16 needs to be changed in that case. Also, **root** becomes a set and must be updated during the algorithm.

Input: Directed Graph $G = \langle E, V \rangle$, weight function $w : E \to \Re$
Output: An MSA of G

$S \leftarrow$ new disjoint set; $W \leftarrow$ new disjoint set
$F \leftarrow$ new empty forest
For each $v \in V$, let $I[v]$ be the list of all incoming edges of v
foreach $v \in V$ **do**
    Sort the edges in $I[v]$ by the tails using radix sort
    S.`makeset(`v`)`; W.`makeset(`v`)`
    $\mathsf{min}[v] \leftarrow v$
end
$\mathsf{roots} \leftarrow V$
while $\mathsf{roots} \neq \emptyset$ **do**
    $r \leftarrow \mathsf{roots.pop()}$
    if $I[r] = \emptyset$ **then**
        $\mathsf{root} \leftarrow r$
        break
    end
    $(u, v) \leftarrow \underset{e \in I[r]}{\operatorname{argmax}}\, w(e)$
    Insert (u, v) as a node in F with any edges in `cycle`$[r]$ as children
    if `cycle`$[r] = \emptyset$ **then**
        $\lambda[v] \leftarrow$ pointer that points to the node (u, v) of F
    end
    if W.`find(`u`)` $\neq W$.`find(`u`)` **then**
        W.`union(`W.`find(`u`)`, W.`find(`v`))`
        $\mathsf{enter}[r] \leftarrow (u, v)$
    else
        $C \leftarrow$ the edges of the newly created cycle between components in S
        $e \leftarrow \underset{e' \in C}{\operatorname{argmin}}\, w(e')$
        $X \leftarrow \emptyset$
        for $(u', v') \in C$ **do**
            $k \leftarrow S$.`find(`v'`)`
            $X \leftarrow X \cup \{k\}$
            Subract $w(u', v') - w(e)$ from the cost of all elements in $I[k]$
        end
        $m \leftarrow \min[S.\texttt{find}(\text{head}(e))]$
        Union all components in X into one component of S
        Let r' be the representative of the new component
        $\min[r'] \leftarrow m$
        $\mathsf{roots} \leftarrow \mathsf{roots} \cup \{r'\}$
        `cycle`$[r'] \leftarrow C$
        Merge the lists $I[k]$ for $k \in X$ into one list $I[r']$
          omitting any edges from $S_{r'}$ to $S_{r'}$
    end
end
if $\lambda[\min[\mathsf{root}]] \neq \emptyset$ **then**
    Identify the path P in F from a root node to $\lambda[\min[\mathsf{root}]]$
    Delete from F all nodes of P and all arcs directed out of these nodes
end
$B \leftarrow \emptyset$
$N \leftarrow$ set of root vertices of F
while $N \neq \emptyset$ **do**
    pick any root node $(u, v) \in N$ and add it to B
    Identify the path P in F from (u, v) to $\lambda[v]$
    Delete from F all nodes of P and all arcs directed out of these nodes
    Update the set N with any new root nodes of F
end

References

[1] F. Bock. An algorithm to construct a minimum directed spanning tree in a directed network. *Developments in Operations Res. 1, Proc. 3rd annual Israel Conf. Operations Res.*, 1969.

[2] P. M. Camerini, L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks*, 9(4):309–312, 1979.

[3] P.M. Camerini, L. Fratta, and F. Maffioli. The K best spanning arborescences of a network. *Networks*, 10(2):91–110, 1980.

[4] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Sci. Sinica*, 14:1396–1400, 1965.

[5] J. Edmonds. Optimum Branchings, Mathematics and the Decision Sciences, Part 1. *Amer. Math. Soc. Lectures Appl. Math*, 11:335–345, 1968.

[6] 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.

[7] RM Karp. A simple derivation of Edmonds' algorithm for optimum branching. *Networks*, 1(265-272):5, 1971.

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