Lecture notes on "Analysis of Algorithms": **Directed Minimum Spanning Trees** (More complete but still unfinished)

Lecturer: *Uri Zwick* [*]

March 20, 2013

Abstract

We describe an efficient implementation of Edmonds' algorithm for finding minimum *directed* spanning trees in directed graphs.

1 Minimum Directed Spanning Trees

Let $G = (V, E, w)$ be a weighted directed graph, where $w : E \to \mathbb{R}$ is a cost (or weight) function defined on its edges. Let $r \in V$. A directed spanning tree (DST) of G rooted at r, is a subgraph T of G such that the undirected version of T is a tree and T contains a directed path from r to any other vertex in V. The cost $w(T)$ of a directed spanning tree T is the sum of the costs of its edges, i.e., $w(T) = \sum_{e \in T} w(e)$. A minimum directed spanning tree (MDST) rooted at r is a directed spanning tree rooted at r of minimum cost. A directed graph contains a directed spanning tree rooted at r if and only if all vertices in G are reachable from r. This condition can be easily tested in linear time.

The proof of the following lemma is trivial as is left as an exercise.

Lemma 1.1 *The following conditions are equivalent:*

(i) T is a directed spanning tree of G rooted at r.

(ii) The indegree of r in T is 0, the indegree of every other vertex of G in T is 1, and T is acyclic, *i.e., contains no directed cycles.*

(iii) The indegree of r in T is 0, the indegree of every other vertex of G in T is 1, and there are directed paths in T from r to all other vertices.

[*]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E–mail: zwick@tau.ac.il

A vertex u is a *descendant* of v in T if and only if there is a directed path from v to u in T. As an immediately corollary of Lemma 1.1 we get:

Corollary 1.2 *Let T be a directed spanning tree of T rooted at r. Let $(u,v) \in E$ be an edge not in T, such that $v \neq r$ and u is not a descendant of v in T. Let (u',v) be the (unique) edge in T entering v. Then, $T \cup \{(u,v)\} \setminus \{(u,v')\}$ is also a directed spanning tree of G rooted at r.*

2 Closely related problems

In the formulation of the MDST problem given above, the root r was specified. We can also consider the version of MDST problem in which the root is not specified, in which case we have to find the root r for which the cost of the minimum directed spanning tree rooted at r is minimized. It is not difficult, however, to devise linear time reductions between the MDST problems with a specified or an unspecified root. We leave that as an exercise.

Instead of wanting a *minimum* spanning tree, we can also ask for a *maximum* spanning tree. We can easily do that by negating all the costs.

Historically, most MDST algorithms were presented as algorithms for finding *optimum branchings.* A branching $B \subseteq E$ of $G = (V,E)$ is an acyclic subset of edges in which the indegree of each vertex is at most 1. We leave it again as an exercise to give simple linear time reductions from the MDST problem to the problem of finding a branching of minimum/maximum total weight.

3 Differences between the directed and undirected cases

In undirected graphs, every edge is part of some spanning tree. In directed graphs, not all edges are part of DSTs. Edges that are not contained in any DST are said to be *useless.* The simplest example of useless edges are edges that enter the root r. Other edges may also be useless.

Directed versions of the cut and cycle rules that were so instrumental in devising algorithms for finding minimum spanning trees in undirected graphs are no longer valid, even if we assume that all edges are useful. (Finding counterexamples is again left as an exercise.) We thus need to use a new technique to finding MDSTs.

4 Edmonds' algorithm

We now sketch the basic algorithmic approach for solving the MDST problem suggested by Edmonds [Edm67], and independently by Bock [Boc71] and Chin and Liu [CL65].

Let $G = (V,E,w)$ be a weighted directed graph, let $r \in V$, and assume that $G = (V,E)$ has a DST rooted at r. Let F be a set of $n-1$ edges obtained by selecting the cheapest edge in G entering each vertex $v \neq r$. If we are really lucky, F is acyclic. By Lemma 1.1, F is a DST, in which case it is also a MDST. Unfortunately, F may, of course, contain cycles.

Let C be a cycle, not containing r, composed of cheapest entering edges. Obviously, a MDST cannot contain all edges of C. We can show, however, that there is a MDST that contains all edges of C, except one.

Lemma 4.1 *Let $G = (V, E, w)$ be a weighted directed graph and let $r \in V$. Let C be a directed cycle in G, not containing r, composed of cheapest entering edges. Then, there is a MDST of G rooted at r that contains all the edges of C except one.*

Proof: Let T be a MDST of G rooted at r. Let v_1 be a vertex on C such that the path from r to v_1 in T does not pass through any other vertices of C. (There must be at least one such vertex, i.e., one of the vertices of C that are closest to r in T.) Let $v_1, v_2, \ldots, v_k$ be the vertices of C in the order in which they appear on the cycle. If $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k) \in T$, we are done. Suppose, therefore, that $(v_1, v_2), (v_2, v_3), \ldots, (v_{i-1}, v_i) \in T$ while $(v_i, v_{i+1}) \notin T$, for some $i < k$. Let $(u, v_{i+1}) \in T$ be the edge entering v_{i+1} in T. The ancestors of v_i in T are the vertices on the path from r to v_1 in T and the vertices $v_1, v_2, \ldots, v_{i-1}$ on C. Thus, v_i is not a descendant of v_{i+1}. By Corollary 1.2, $T' = T \cup \{(v_i, v_{i+1})\} \setminus \{u, v_{i+1})\}$ is also a spanning tree rooted at r. As (v_i, v_{i+1}) is a cheapest edge entering v_{i+1}, we get that $w(v_i, v_{i+1}) \leq w(u, v_{i+1})$, and hence $w(T') \leq w(T)$. Thus, T' is also a MDST rooted at r. Continuing in this way, we can construct a MDST rooted at r that contains all the edges of C expect (v_k, v_1). $\square$

Suppose that C is a cycle of cheapest entering edges. We now know that there is an MDST that contains all the edges of C, except one. But which one? Which edge of C should we throw away? Edmonds [Edm67] suggests an elegant solution to this problem. We *contract* the cycle C, appropriately modify the cost of the edges that enter C in the contracted graph, and then recursively find an MDST $\bar{T}$ rooted at r in the contracted graph. If the edge $\bar{e} \in \bar{T}$ that enters C in $\bar{G}$ is derived from an edge $e = (u, v)$ that enters C at v, we convert $\bar{T}$ into an MDST of G by adding to it all the edges of C except the edge of C entering v. We show below that T is then an MDST of G. We next formalize this approach and prove its correctness.

Let C be a cycle of G and let $\bar{G} = G/C$ be the graph obtained from G by contracting C. Formally, if C is composed of the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)$, then the vertex set of $\bar{G}$ is $V \cup \{c\} \setminus \{v_1, v_2, \ldots, v_k\}$, i.e., the vertices $v_1, v_2, \ldots, v_k$ of the cycle are replaced by the *super-vertex* c. The edge set of $\bar{G}$ is $\bar{E} = \{\bar{e} \neq (c, c) \mid e \in E\}$, where if $e = (u, v)$, then $\bar{e} = (\bar{u}, \bar{v})$, where $\bar{u} = u$, if $u \neq C$, and $\bar{u} = c$, if $u \in C$. (Note that self-loops (c, c), if formed, are removed. Parallel edges are kept.) By a slight abuse of notation, we identify each edge of $\bar{E}$ with the edge of E that gave rise to it, and thus assume that $\bar{E} \subseteq E$.

There is a natural one-to-one correspondence between DSTs of G rooted at r that contain all edges of C, except one, and DSTs of $\bar{G}$ rooted at r. If T is a DST of G rooted at r and $T \cap C = C \setminus \{e\}$, then it is easy to see that $\bar{T} = T \setminus (C \setminus \{e\})$ is a DST of $\bar{G}$ rooted at r. Conversely, if $\bar{T}$ is a DST of $\bar{G}$ rooted at r, we can *expand* it into a DST $\exp(\bar{T})$ of G as follows. Let $e \in \bar{E}$ be the edge in $\bar{T}$ that enters c. (Recall that according to our convention, the edges of $\bar{T}$ are also edges of the original graph G.) If $e = (u, v)$, where $v \in C$, let $e' = (u', v)$ be the edge of C that enters v. We let

$$\exp(\bar{T}) = \bar{T} \cup (C \setminus \{e'\}) ,$$

i.e., $\exp(\bar{T})$ is obtained by adding all the edges of C, except e', to $\bar{T}$. It is again easy to check that $\exp(\bar{T})$ is a DST of G rooted at r and it clearly contains all edges of C, except one. It is also easy to check that if $\bar{T}$ is the DST in $\bar{G}$ corresponding to a DST T in G that contains all the edges of C except one, then $\exp(\bar{T}) = T$.

If $e = (u, v) \in E$ enters C, we let $e_C = (u', v)$ be the edge of C that enters v. We next define a new cost function $\bar{w} : E \to \mathbb{R}$ on the edges of G, and hence also on the edges of $\bar{G}$ as follows:

$$\bar{w}(e) \;=\; \begin{cases} w(e) - w(e_C) & \text{if } e \text{ enters } C, \\ w(e) & \text{otherwise.} \end{cases}$$

Lemma 4.2 *Let $G = (V, E, w)$ be a weighted directed graph. Let $r \in V$. Let C be a directed cycle in G. Let T be a DST of G rooted at r such that $|T \cap C| = |C| - 1$, i.e., T contains all the edges of C, except one, and let $\bar{T}$ be the DST in $\bar{G} = G/C$ corresponding to T. Then,*

$$\bar{w}(\bar{T}) \;=\; w(T) - w(C) .$$

Proof: Suppose that $T \cap C = C \setminus \{e\}$. Then, $\bar{T} = T \setminus (C \setminus \{e\})$. Let e' be the edge of T that enters C. (Note that e and e' enter the same vertex of C.) Then,

$$\begin{aligned} \bar{w}(\bar{T}) &= (w(T) - w(C \setminus \{e\})) - w(e) \\ &= w(T) - w(C) . \end{aligned}$$

Note that $w(C \setminus \{e\})$ is subtracted as the edges of $C \setminus \{e\}$ are removed from T to form $\bar{T}$. The cost $w(e)$ is subtracted as $\bar{w}(e') = w(e') - w(e)$. For all other edges $e'' \in \bar{T}$ we have $\bar{w}(e'') = w(e'')$. $\square$

We now have

Theorem 4.3 *Let $G = (V, E, w)$ be a weighted directed graph. Let $r \in V$. Let C be a directed cycle, not containing r, composed of cheapest entering edges. Let $\bar{G} = G/C$ and $\bar{w} : E \to \mathbb{R}$ be as defined above, and let $\bar{T}$ be a MDST of $\bar{G}$ with respect to $\bar{w}$ rooted at r. Then* $\exp(\bar{T})$ *is a MDST of G rooted at r.*

Proof: We know, by Lemma 4.1, that there is a MDST T^* rooted at r that contains all edges of C, except one. Let $\overline{T^*}$ be the DST corresponding to T^* in $\bar{G}$. Let $T = \exp(\bar{T})$. By Lemma 4.2 we have $\bar{w}(\bar{T}) = w(T) - w(C)$ and $\bar{w}(\overline{T^*}) = w(T^*) - w(C)$. As $\bar{T}$ is an MDST in $\bar{G}$, we get that $\bar{w}(\bar{T}) \leq \bar{w}(\overline{T^*})$. Thus $w(T) \leq w(T^*)$ and T is also a MDST rooted at r. $\square$

This suggests the following algorithm for finding a MDST, which is essentially the algorithm devised by Edmonds [Edm67]. Choose cheapest entering edges of vertices other than the root until either a DST is formed, in which case it is a MDST, or until a cycle C is formed. If a cycle C is formed, contract it and adjust the edge costs appropriately. Find an MDST in the contracted graph and expand it into a MDST of the original graph.

5 Efficient implementation

We next present an efficient implementation of Edmonds' algorithm [Edm67] that runs in $O(m \log n)$ time. The implementation presented is mainly based on an implementation suggested by Tarjan [Tar77], incorporating also some ideas of Camerini *et al.* [CFM79, CFM80] and Gabow *et al.* [GGST86], and some new ideas. Gabow *et al.* [GGST86] went on to obtain a more efficient $O(m + n \log n)$ time implementation which we will not cover in detail in class.

We assume, for simplicity, that the input graph $G = (V, E, w)$ is strongly connected. If not, we can easily make it so by adding $O(n)$ edges of sufficiently high cost.

5.1 High level description

The generic algorithm of the previous section maintains a set F of cheapest entering edges. Initially F is empty. At each step, the algorithm chooses an arbitrary vertex $v \neq r$ which does not yet have an incoming edge in F, finds a cheapest edge $(u, v) \in E$ entering v, and adds (u, v) to F. If F is still acyclic, the algorithm proceeds. Otherwise, the cycle C formed in F is found and contracted and the process continues. The process ends when F is DST of the contracted graph. The contractions should then be expanded, in reverse order, i.e., the most recent contraction first. In the worst-case, the algorithm may need to perform $\Omega(n)$ nested contractions.

In the efficient implementation of this section, the set F of cheapest entering edges always forms a *path* $v_0 \leftarrow v_1 \leftarrow \cdots \leftarrow v_k$, where each v_i is either an original vertex of the graph or a super-vertex obtained by contracting a cycle of (super-)vertices. (Formally, the path is composed of the edges $(v_k, v_{k-1}), \ldots, (v_1, v_0)$.) Initially $v_0 = a$, where a is an arbitrary vertex of the graph. In each step, the algorithm finds the cheapest edge $v_k \leftarrow u$ entering v_k. If u is not one of $v_0, \ldots, v_{k-1}$, the path is extended by letting $v_{k+1} = u$. If $u = v_i$, a cycle $v_i \leftarrow \cdots \leftarrow v_k \leftarrow v_i$ is found. The cycle is contracted, forming a new super-vertex c. The (super-)vertices $v_i, \ldots, v_k$ are then replaced by c. This contraction phase continues until the whole graph is contracted into a single super-vertex. The root r does not play a special role in the contraction phase. In particular, the cheapest edge entering r is eventually chosen and r eventually participates in a contracted cycle. (Recall our assumption that G is strongly connected.)

There are two reasons for ignoring the identity of r during the contraction phase, as done above. The first is that it allows us to maintain the invariant that the selected cheapest incoming edges form a path. Otherwise, we would not be able to continue when r is added to the path. The

```
Function MDST($G = (V, E, w), r$)
    contract($G$)
    return expand($r$)
```

Figure 1: Algorithm for finding MDSTs.

second is that the same contraction phase can now be used to find an MDST from *any* root. As the expansion phase requires only $O(n)$ time, we can thus find MDSTs from k specified roots in $O(m \log n + kn)$.

The whole algorithm, as shown in Figure 1, is thus composed of a contraction phase `contract`(G), which does not depend on r, followed by an expansion phase `expand`(r) that does depend on r.

Many implementation details need to be filled. We need to explain how we implement the contractions performed by the algorithm, how we appropriately adjust the edge weights following a contraction, how we efficiently find a cheapest edge entering a given (super-)vertex, and how we determine whether an added edge closes a cycle. Finally we need to explain how the contractions are undone in the expansion phase.

The algorithm generates explicit objects that represent the super-vertices created during the contraction process, and maintains a *contraction tree* that records the nesting of the contraction performed. This facilitates, of course, the expansion stage. Endpoint of edges are not updated, however, when contractions are made, as that would be too costly. In particular, self-loops are not eliminated and parallel edges are maintained. A *union-find* data structure is used to find the endpoints, in the current version of contracted graph, of an edge (u, v) of the original graph.

For every (super-)vertex of the current graph we maintain a *priority queue* that holds all the edges entering u in the graph. To find a cheapest edge entering a given (super-)vertex, we repeatedly perform *extract-min* operations on its priority-queue until the returned edge is not a self-loop in the current contracted graph. This turns our to the bottleneck step of the algorithm in terms of complexity, giving it its $O(m \log n)$ running time. When several (super-)vertices are contracted to form a new super-vertices, we *meld* their priority queues to form the priority queue of the new super-vertex.

5.2 Detailed description of contraction phase

Pseudo-code of `contract`(G), which performs the contraction phase, is given in Figure 2. It begins by a call to `initialize`, given in Figure 3 which performs the necessary initializations.

For every (super-)vertex u we maintain the following fields:

- $in[u]$ - the selected incoming edge of u. During the contraction phase this is the cheapest edge entering u. If no incoming edge was selected yet, then $in[u] = null$.

```
Function Contract($G = (V, E, w)$)

initialize()
$a \leftarrow$ arbitrary vertex
while $P[a] \neq \emptyset$ do
    $(u, v) \leftarrow$ extract-min($P[a]$)
    $b \leftarrow$ find($u$)
    if $a \neq b$ then
        $in[a] \leftarrow (u, v)$
        $prev[a] \leftarrow b$
        if $in[u] = null$ then // Path extended
            $a \leftarrow b$
        else // New cycle formed
            $c \leftarrow$ vertex()
            while $parent[a] = null$ do
                $parent[a] \leftarrow c$
                $const[a] \leftarrow -w[in[a]]$
                insert($children[c], a$)
                $P[c] \leftarrow$ meld($P[c], P[a]$)
                $a \leftarrow prev[a]$
```

Figure 2: The contraction phase of the MDST algorithm.

```
Function initialize()

foreach $u \in V$ do
    init-vertex($u$)
foreach $(u, v) \in E$ do
    insert($P[v], (u, v)$)
```

```
Function init-vertex($u$)

$in[u] \leftarrow null$
$const[u] \leftarrow 0$
$prev[u] \leftarrow null$
$parent[u] \leftarrow null$
$children[u] \leftarrow null$
$P[u] \leftarrow$ priority-queue()
```

Figure 3: Initializing the contraction phase.

- $const[u]$ - a constant that should be added to the weight of all edges entering u to obtain the adjusted cost of the edge in the contracted graph. Initially $cost[u] = 0$.

- $prev[u]$ - the (super-)vertex preceding u in the path constructed by the algorithm. If u is not yet in the path, or was just added to it, then $prev[u] = null$.

- $parent[u]$ - The super-vertex into which u was contracted. If u was not contracted yet, then *parent*$[u]$ = *null*.

- *children*$[u]$ - A list of the (super-)vertices contacted to form u. If u is a vertex of the original graph then *children*$[u]$ is empty.

- $P[u]$ - a priority queue that holds all the incoming edges of u.

`initialize`() starts by calling `init-vertex`(u), for every vertex u of the graph. `init-vertex`(u) initializes the fields of u mentioned above. Next, `initialize`() inserts each edge (u, v) of the input graph into the appropriate priority queue, i.e., $P[v]$.

The contraction phase begins by selecting an arbitrary vertex a to form the first vertex of the initial path. During the running of the algorithm, a would represent the first (super-)vertex on the directed path constructed so far. (In the notation used in the high-level description above we have $a = v_k$.) As long as a still has incoming edges, i.e., $P[a] \neq \emptyset$, we try to extract the cheapest edge entering a. As explained above, we do that by performing repeated `extract-min`($P[a]$) operations until the extracted edge is not a self-loop.

Let (u, v) be the edge returned by `extract-min`($P[a]$). As a may be a super-vertex, we do not necessarily have $v = a$. Recall that (u, v) is an edge in the original graph. Both its endpoints may have already participated in contractions. There are now three cases, as depicted in Figure 4:

1. The edge (u, v) is a self-loop, in which case it should be ignored. (See upper drawing.)

2. The vertex u is not on the path, in which case the path is extended. (See middle drawing.) (Note that if u does not belong to any of the (super-)vertices on the path, then u was not contracted yet.)

3. Vertex u is part of a super-vertex b contained in the path. (Possibly $b = u$.) In this case a cycle is formed and it needs to be contracted. (See lower drawing.)

How do we know which case applies? We first need to find the super-vertex b that currently contains u. We would later use a union-find data structure to find b, but for the time being consider the naïve implementation of `find` given in Figure 5. To find the top-most super-vertex that contains u we simply follow *parent* pointers from u until we reach a super-parent whose parent point is *null*. This vertex is then returned. We can thus find b using $b \leftarrow$ `find`(u).

If $a = b$, then (u, v) is a self-loop and the current iteration of the while loop is then. The next iteration of the while loops extracts the next edge from $P[a]$, as required.

If $a \neq b$, the algorithm lets $in[a] \leftarrow (u, v)$, to record that (u, v) is the cheapest edge entering a, and $prev[a] \leftarrow b$ to record that b would be the (super-)vertex preceding a in the path, or in the cycle formed. (As explained, if the path is extended, then $b = u$.)

To distinguish between the second and third cases, we look at $in[u]$. If $in[u]$ = *null*, then u is not on the path, so the path has been extended. We let $a \leftarrow b$, as b is the new first vertex on the path,

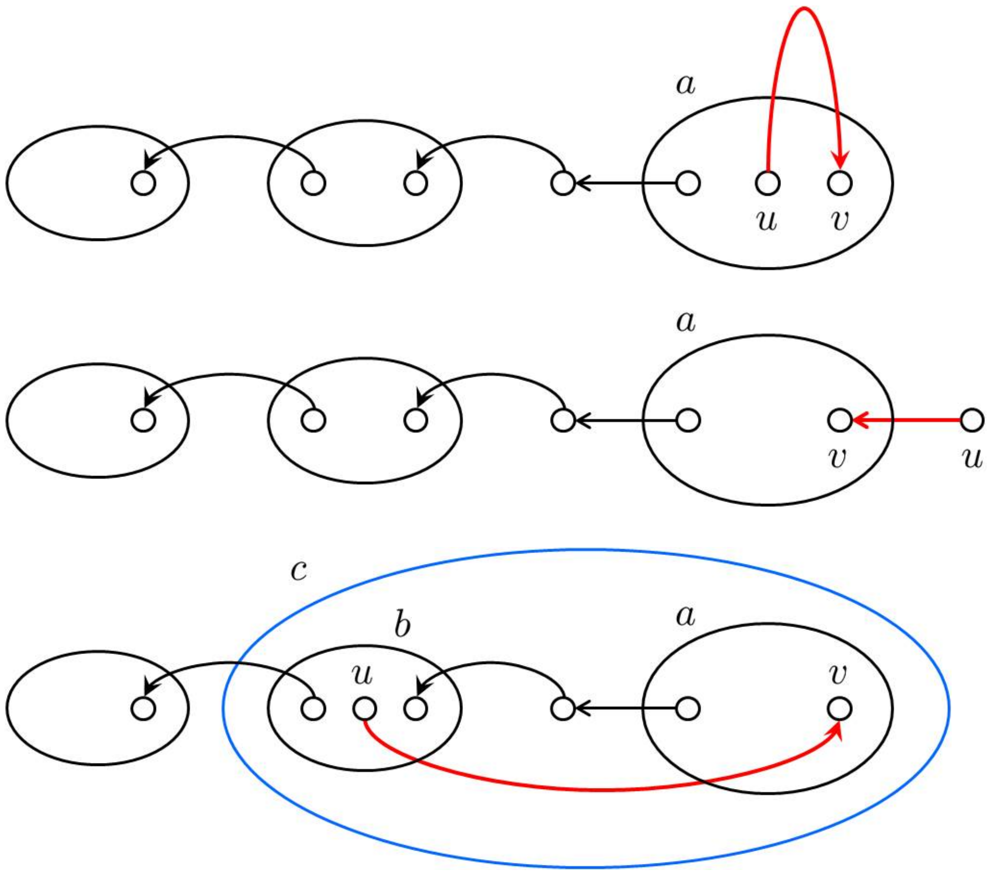


Figure 4: The three cases encountered while trying to expand the path.

and proceed with the next iteration of the while loop. (Note that the condition $in[u] = null$ here could have been replaced by $in[b] = null$.)

If $in[u] \neq null$, then a new cycle is formed. We create a new super-vertex by the call $c \leftarrow$ `vertex`(). This call allocates c and initializes it by calling `init-vertex`(c). (The call to `init-vertex`(c) is implicit and not shown in the pseudo-code.) For each (super-)vertex a in the cycle formed we set $parent[a] \leftarrow c$, to record that c becomes the parent of a, and we insert a into $children[c]$. We also set $const[a] \leftarrow -w[in[a]]$ and meld $P[a]$ into $P[c]$. We iterate over the (super-)vertices of the cycle using the command $a \leftarrow prev[a]$. Finally, we detect that we have gone full circle by checking whether we got to a vertex with $parent[a] \neq null$.

A sample run of the contraction phase of the algorithm is given in Figure 6. The tree in the figure is the contraction tree defined by the *parent* pointers of the (super-)vertices.

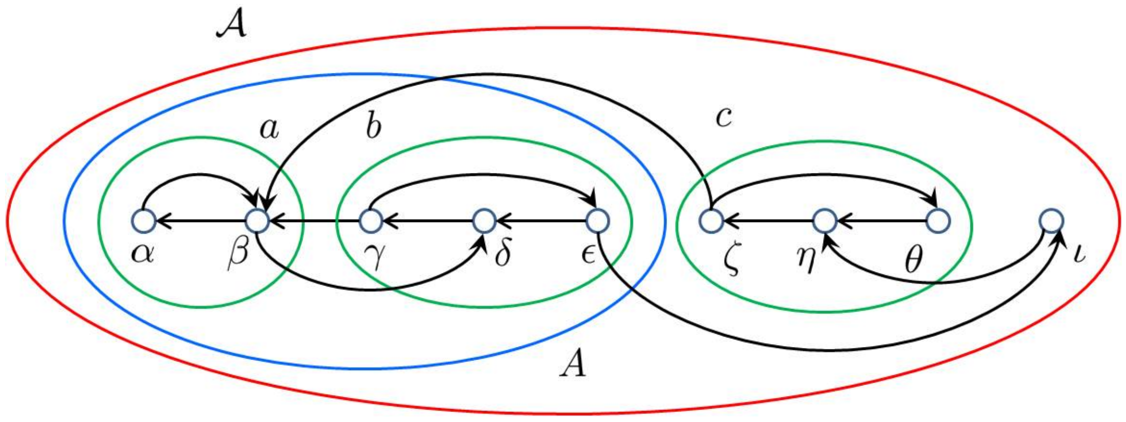
5.3 The expansion phase

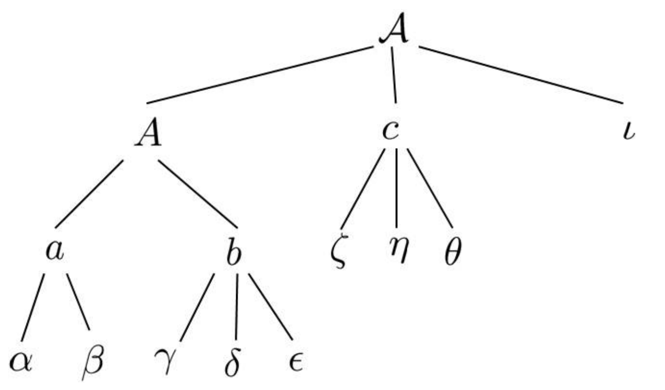
Pseudo-code of `expand`, which performs the expansion phase, is given on the left of Figure 7. `expand` uses a simple function called `dismantle` given on the right of Figure 7.

```
Function find(u)
    while $parent[u] \neq null$ do
        $u \leftarrow parent[u]$
    return $u$
```

```
Function weight(u, v)
    $w \leftarrow w[u, v]$
    while $parent[v] \neq null$ do
        $w \leftarrow w + const[v]$
        $v \leftarrow parent[v]$
    return $u$
```

Figure 5: Naïve implementation of `find` and `weight`.





$in[\alpha] = (\beta, \alpha)$	$in[a] = (\gamma, \beta)$
$in[\beta] = (\alpha, \beta)$	$in[b] = (\beta, \delta)$
$in[\gamma] = (\delta, \gamma)$	$in[c] = (\iota, \eta)$
$in[\delta] = (\epsilon, \delta)$	
$in[\epsilon] = (\gamma, \epsilon)$	$in[A] = (\zeta, \beta)$
$in[\zeta] = (\eta.\zeta)$	
$in[\eta] = (\theta, \eta)$	
$in[\theta] = (\iota, \theta)$	
$in[\iota] = (\epsilon, \iota)$	

Figure 6: A sample run of the contraction phase.

The first task of `expand` is to undo the contractions involving the root r. The super-vertices that absorbed r can be easily found by following *parent* pointers from r until getting to the root of the contraction tree. Suppose that u is super-vertex on this path. To undo the contraction of u we simply need to reset to the *parent* pointers of all the children of u to *null*, letting these children know they are now root super-vertices in the contraction forest. If we would like to undo the contractions of r in the reverse chronological order, we need to perform these operations top down. However, it is easy to see that the order of these operations does not matter, so we can simply do them bottom up while tracing the path from r to the root of the contraction tree. We refer to this as *dismantling*

```
Function expand(r)
  $R \leftarrow \emptyset$
  dismantle($r$)
  while $R \neq \emptyset$ do
      $c \leftarrow$ extract($R$)
      $(u, v) \leftarrow in[c]$
      $in[v] \leftarrow (u, v)$
      dismantle($v$)
  return $\{in[u] \mid u \in V \setminus \{r\}\}$
```

```
Function dismantle(u)
  while $parent[u] \neq null$ do
      $u \leftarrow parent[u]$
      foreach $v \in children[u]$ do
          $parent[v] \leftarrow null$
          if $children[v] \neq null$ then
              insert($R, v$)
```

Figure 7: The expansion phase of the MDST algorithm.

the path from r to the root of the contraction tree.

`dismantle`(u) dismantles the path from a vertex u to the root of its tree in the contraction forest, as explained above. While doing so, it adds each super-vertex that becomes a root of the component forest into a list R which is initially empty. Undoing the contractions of r can therefore be done by calling `dismantle`(r).

For example, if in the example of Figure 6 we would like to find the MDST rooted at δ, undoing all contractions involving δ, by calling `dismantle`(δ), yields the graph shown on the top of Figure 8. After the call we have $R = \{a, c\}$.

We next have to undo the remaining contractions. The contraction forest may now be composed of several trees. the order in which we work on these trees is not important. Recall that R contains the roots of these trees. In each iteration we extract a super-vertex c from R. Let $in[c] = (u, v)$. Note that v is necessarily a descendant of c in the contraction forest. The edge (u, v) is the edge entering v in the DST obtained by undoing the contractions, we thus let $in[v] \leftarrow (u, v)$. We undo all contractions involving v by calling `dismantle`(v).

Continuiung our running example, the resulting DST rooted at δ is shown at the bottom of Figure 8.

References

[Boc71] F. Bock. An algorithm to construct a minimum directed spanning tree in a directed network. *in: Developments in Operations Research, Gordon and Breach, New York*, pages 29–44, 1971.

[CFM79] P.M. Camerini, L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks*, 9:309–312, 1979.

[CFM80] P.M. Camerini, L. Fratta, and F. Maffioli. The k best spanning arborescences of a network. *Networks*, 10(2):91–109, 1980.

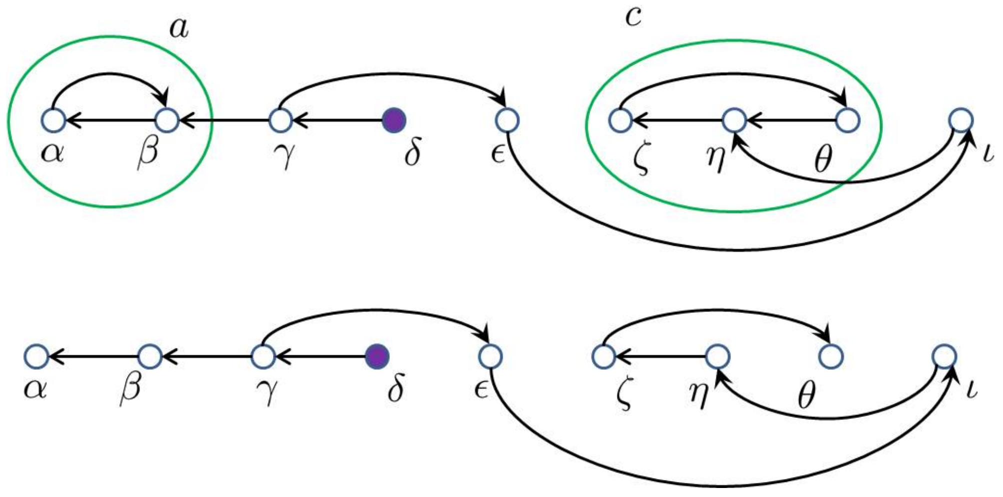


Figure 8: A sample run of the expansion phase.

[CL65] Y.J. Chu and T.H. Liu. On the shortest arborescence of a directed graph. *Sci. Sinica*, 14:1396–1400, 1965.

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

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[GGST86] H.N. Gabow, Z. Galil, T.H. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

[Kar71] R.M. Karp. A simple derivation of Edmonds' algorithm for optimum branchings. *Networks*, 1:265–272, 1971.

[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

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

[Tar79] R.E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.