Large scale phylogenetic inference based on minimum spanning arborescences

Joaquim Espada
Instituto Superior Técnico
Lisboa, Portugal
joaquim.espada@tecnico.ulisboa.pt

***Abstract*—Current methods struggle to reconstruct phylogenetic relationships and evolution patterns of more than a few thousands of bacterial genomes. Calculating minimum spanning trees from legacy MLST is a quick and efficient way of reconstructing phylogenetic relationships. Legacy MLST is based on only seven loci and there is no missing data. Missing data is a problem for the minimum spanning tree approach. Recently, it was shown that finding minimum spanning arborescences can help with this problem by considering asymmetric pairwise distances in directed graphs instead of symmetric distances in undirected graphs. The present approach depends on a fully connected graph. The running time complexity of algorithms for finding spanning arborescences of minimum weight such as Edmonds' algorithm is at least quadratic therefore the running time may still be problematic for analyzing large data sets.**

In this paper we present an Edmonds' algorithm implementation that takes $O(|E| \log |V|)$ time, where V denotes the set of vertices and E denotes the set of edges of a directed graph $G = (V, E)$. Moreover, we consider the problem of updating a minimum spanning arborescence or a directed minimum spanning tree of G, whenever an edge is deleted or an edge is added. Our implementation of the dynamic minimum spanning arborescence proved to be two times faster than calculating Edmonds' algorithm over a sequence of m operations.

***Index Terms*—Phylogenetic Inference, Minimum Spanning Trees, Minimum Spanning Arborescences, Dynamic Minimum Spanning Arborescences, Missing Data.**

State-of-the-art phylogenetic inference methods attempt to reconstruct the evolutive patterns and phylogenetic relationships of more than a few thousands of bacterial genomes. Initially, the genetic relationships of legacy STs were represented by phylograms based on hierarchical clustering methods. Phylograms may, however, be problematical for the presentation of large numbers of genotypes because each genotype is represented by a unique branch, even when multiple genotypes are identical. An alternative to phylograms are minimum spanning trees. Calculating minimum spanning trees from legacy MLST is quick and efficient because legacy MLST is based on only seven loci, and allelic calls for each of the seven loci are a prerequisite for calling an ST, i.e. no missing data. However, cgMLST may span hundreds of loci and STs routinely include low levels of missing data because some cgMLST genes are occasionally deleted, or are not identified due to various problems in the bioinformatic analysis, such as erroneous assemblies of genomes from short reads. Such missing alleles result in multiple almost identical STs which only differ due to missing data but each of which is nevertheless a unique node in a phylogram because its allelic contents differs from those of other STs. The problem is even more dramatic in wgMLST schemas: the large majority of loci belong to the accessory genome, only a fraction of which is contained in any individual genome. Missing data is a problem for the classical minimum spanning tree approach and, recently, it was shown that finding minimum spanning arborescences can help with this problem by considering directed graphs instead of undirected ones [1]. Nevertheless, given that the present approach depends on a fully connected graph and the running time complexity of Edmonds' algorithm is at least quadratic the running time may still be problematic for analyzing large data sets. In this article we provide a fast and efficient implementation of Edmonds' algorithm able to handle large data sets based on the contributions [2]–[4]. Moreover, we go further by giving an implementation for the maintenance of a dynamic minimum spanning arborescence based on the ideas from [5], [6]. The latter problem, allow us to quickly perceive the changes over a directed phylogenetic tree if an edge is removed or added.

In Section I, we introduce the problem of finding a minimum spanning arborescence. Still, in Section I, the fully dynamic maintenance of minimum spanning arborescence is formalized. In Section II, we present the details of our implementations of Edmonds' algorithm as well as the technicalities related with the maintenance of a dynamic minimum spanning arborescence. In Section III, we describe the experimental protocol used to assess all the developed work along with the experimental evaluation. Finally, in Section IV we present the limitations of this work and prospects for future contributions.

I. Problem

Let $G = (V, E)$ denote a weighted directed graph. G consists of a set of **vertices** V and a set of **edges** E. Each edge is an ordered pair (u, v) of distinct vertices u, v, called **endpoints**. Every $e \in E$ as a real value cost function $w(e)$.

A. Minimum spanning arborescence

Given a directed connected weighted graph G, a directed arborescence of G rooted at r, $r \in V$, is a subgraph T of G such that the undirected version of T contains a directed path from r to any other nodes in V. A minimum spanning arborescence rooted at r is a directed spanning tree rooted at r of minimum cost. A directed connected graph contains a directed spanning tree rooted at r if and only if all nodes in G are reachable from r.

Several authors such as Edmonds' [7], Bock [8] as well as Chu and Liu [9], independently devised the same algorithm. In this document, we are only focusing on Edmonds' description of the algorithm. The proofs of correctness can be found in [7], [10] based on linear programming and combinatorial theory.

B. Dynamic Minimum spanning arborescence

Given a directed connected weighted graph G consider the on-line problem of maintaining a minimum spanning arborescence T can be stated as follows: a minimum spanning arborescence is to be maintained for an underlying directed graph G which is modified repeatedly by inserting, removing or having the cost of an edge changed.

II. Approach

A. Edmonds' algorithm overview

Before presenting Edmonds' algorithm, it is important to notice that the original description of Edmonds' algorithm [7] calculates the optimum branching or the maximum spanning arborescence. By negating the weights of a given directed graph and relying in the original description of Edmonds' algorithm the minimum spanning arborescence is calculated. The process of negating the weights can be easily achieved in linear time, therefore does not affect the Edmonds' algorithm temporal complexity.

In this document Edmonds' algorithm is introduced with the necessary adjustments to find the minimum spanning arborescence without the need of negating the weights of a given directed graph.

Edmonds' proposes an algorithm that has two phases: a contraction and an expansion phase. The first phase begins with no edges selected and in general maintains a set of selected edges that defines the minimum arborescence under construction. As the phase proceeds, cycles of selected edges are formed. Each cycle is contracted to form a new (super-) vertex. The second phase of the algorithm consists of expanding the cycles formed during the first phase, in reverse order of their contraction, and discarding one edge from each cycle to form a spanning arborescence in the original graph.

B. Edmonds' algorithm original description

Let G_i denote the graph of iteration i and D_i, E_i, B_i, Q_i be empty buckets. The bucket D_i stores the nodes with a incident minimum weight edge selected, bucket E_i holds the chosen edges, B_i holds all the edges before E_i forms a cycle and Q_i stores the cycle.

We start the algorithm by choosing a node $v \in G_i \wedge v \notin D_i$. Insert v in bucket D_i. If there is in G_i a edge directed towards v, insert the one having the minimum weight into bucket E_i. This process is repeated until one of the two following conditions is true:

1) E_i holds a cycle.
2) Every node of G_i is in D_i and the content of E_i does not constitute a cycle.

Whenever condition 1 occurs a new graph G_{i+1} is obtained from G_i by contracting to a single (super-)vertex v_k^{i+1} the cycle Q_i. In order to build G_{i+1} we must compute the reduced cost of the edges incident on Q_i and update the endpoints of the edges. Formally G_{i+1} must comply with the following constraints:

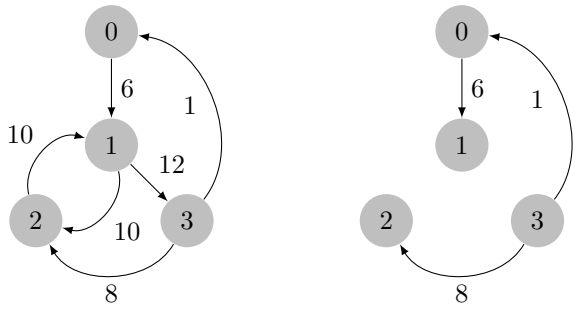
- $\forall_{(u,v)} \in G_i.E \mid u \in Q_i \wedge v \in Q_i \implies (u,v) \notin G_{i+1}$
- $\forall_{(u,v)} \in G_i.E \mid u \in Q_i \wedge v \notin Q_i \implies (u_k^{i+1}, v) \in G_{i+1}$
- $\forall_{(u,v)} \in G_i.E \mid u \notin Q_i \wedge v \in Q_i \implies (v, v_k^{i+1}) \in G_{i+1}$
- $\forall_{(u,v)} \in G_i.E \mid u \notin Q_i \wedge v \notin Q_i \implies (u,v) \in G_{i+1}$

Every edge which is directed towards a node in Q_i has the weights reduced with the following formula:

$$w_{(u,v_k^{i+1})} = w_{(u,v)} + \sigma_{Q_i} - w_{(j,v)} \tag{1}$$

where $w_{(u,v_k^{i+1})}$ denotes the weight of edge (u, v_k^{i+1}) in G_{i+1}, $w_{(u,v)}$ is the weight of $w_{(u,v)}$ in G_i, σ_{Q_i} denotes the maximum weight edge in Q_i and $w_{(j,v)}$ denotes the weight of the cycle edge incident in node v. Continue applying the algorithm and increment i by one unit. Eventually after a few applications condition 2 must occur.

When condition 2 is true the contraction phase ends and the expansion phase begins. The final content of E_i holds a minimum spanning arborescence for graph G_i. Let H denote a subgraph formed by the edges of E_i. If v_k^i is not a root of H then add all edges from Q_i except the one that shares the same destination as an edge currently in H. If v_k^i is a root of H then adding all but one of the edges from Q_i yields an arborescence. Since we are calculating the minimum spanning arborescence we remove the edge with maximum weight of Q_i. This process is repeated until all contractions are undone. The pseudo-code of the original Edmonds' algorithm description is presented in Algorithm 1 and it takes polynomial time [11]. In Figure 1 is depicted a directed graph and the respective minimum spanning arborescence.



(a) Input graph G.

(b) Minimum spanning arborescence of the input graph.

Fig. 1. Minimum Spanning Arborescence of G.

C. Efficient derivation of Edmonds' algorithm

The implementation of Edmonds' algorithm presented in the current section is built around the following contributions [2]–[4].

Algorithm 1: Edmonds' algorithm - Original description

Input: Directed weighted graph $G = (V, E)$
Output: Minimum spanning arborescence H

1) Set $i = 0$, $G_0 = G$, $B_0, E_0, Q_0, D_0 = \emptyset$
2) Choose node v such that $v \in G_i \wedge v \notin D_i$
3) Add v to D_i
4) Find minimum weight edge incident on v and add it to E_i
5) Repeat Steps 2 3 4 until Condition 1 or Condition 2 is true
6) Condition 1 - E_i contains a cycle
   Build contracted version of G_i being G_{i+1}
   Maintain a historic of G_i, B_i, E_i, Q_i
   Increment i
   Go to 2
7) Condition 2 - Expansion Phase
   Let $H \leftarrow E_i$
   While $i > 0$ do
   Decrement i
   If created (super)-vertex at iteration i is a root in H
   $H \leftarrow H \cup Q_i$
   $H \leftarrow H \setminus \max(Q_i)$
   Else
   $\forall_{(u,v)} \in Q_i \cup H \mid \forall (u', v') \in H, v' \neq v$
8) **return** H

1) Contraction phase: The algorithm builds a set of edges H defining a subgraph $G(H) = (V, H)$. It is assumed that the graph $G = (V, E)$ has a vertex set $V = \{1, 2, .., n\}$, and that, for each j, $L(j) = \{(i, j) \in E\}$ is an incidence list for node j. It is necessary to keep track of both weakly and strongly connected components of G as well as the non-examined edges entering each strongly connected component. The bookkeeping mechanism to keep track of the connected components relies in the *union-find* [12] data structure that supports the following operations:

- `MAKE-SET(`x`)`: creates a new set whose only member and representative is x. Since the sets are disjoint, we require that x is not already in some other set.
- `UNION(`x`,`y`)`: unites the sets that contain x and y, let S_x and S_y denote sets containing x and y then the result will be $S_x \cup S_y$.
- `FIND-SET(`x`)`: returns a pointer to the representative of the unique set containing x.

Let `SFIND`, `SUNION`, `SMAKE-SET` denote operation on strongly connected components. Since we represent the edges in the current contracted graph and their modified edge costs implicitly, the union-find associated with strongly connected component must also support the following operations:

- `SADD-WEIGHT(`x`,`k`)`: Add value k to all members of set x
- `SFIND-WEIGHT(`x`)`: Return the current value of set x.

Given a edge $e = (u, v, w) \in E$ the contracted version of e, $e' = (\texttt{SFIND}(u), \texttt{SFIND}(v), w + \texttt{SFIND-WEIGHT(v)})$. Let `WMAKE-SET`, `WFIND`, `WUNION` denote operations over the weakly connected components.

To keep track of the edges entering each strongly component, a collection of binomial heaps [13] is used. The binomial heap data structure implements four operations:

- `MELD(`C, D`)`: adds the elements of heap D to heap C.
  Time required: $O(\log |C| + \log |D|)$.
- `EXTRACT-MIN(`C`)`: returns the smallest element of heap C
  Time required: $O(\log |C|)$.
- `INIT(`C, L`)`: initializes the heap C with all element of list L.
  Time required: $O(|L|)$.
- `ADD(`a, C`)`: add constant a to all elements of heap C.
  Time required: $O(1)$.

Other variables and data structures used in the algorithm:

- *roots* set: gives the root components of $G(H)$ which may have entering edges of positive value.
- *rset* set: gives the root components of $G(H)$ with no entering edges of positive value.
- *inEdgeNode* array: for each i gives the unique node of F, associated with a edge of H, entering the strongly connected component i.
- *max* array: keeps track of the possible roots of the minimum spanning arborescence.
- $w(i, j)$: denotes the weight of edge (i, j). These values are the updated ones, rather than the original values given in the input directed graph.

Still, during the contraction phase, we maintain a forest F, proposed by Camerini [4], of G that at the beginning of the contraction phase is initialized as an empty forest. Every time an edge (u, v) is added to H, a new node is also added to F. The nodes of F represent edges of H. The edges of F are constructed whenever (u, v) is chosen to enter a root component S containing more than one vertex, i.e a root component S corresponds to a cycle, let $\{(x_1, y_1); (x_2, y_2); (x_k, y_k)\}$ be the sequence of edges in S. Afterwards for each i, $1 \leq i \leq k$, an edge is added to F directed from node (u, v) to node (x_i, v_i), so that (x_i, v_i) is a child of (u, v) and (u, v) the parent of (x_i, v_i). In case S contains an unique vertex a pointer is kept $\pi[v] = (u, v)$, π stores the leaf nodes of F. Take into account that during the construction of F we also need to keep track of all root nodes of F, i.e the nodes which have no parent, let this set be denoted by N.

The pseudo-code of the contraction phase is expressed in Algorithm 3.

2) Expansion phase: Once the contraction phase is completed, a minimum spanning arborescence of G can be obtained from F. Let $R = \{max[i] \mid i \in rset\}$ be the set of roots in $G(H)$. Initialize an empty set B, initialize N to be set the roots of F and apply the procedure described in Algorithm 2. The second instruction of Algorithm 2 can be easily achieved by tracing P in the child-to-parent direction, until a root node is found. When Algorithm 2 stops the minimum spanning arborescence is returned in set B, the proof of correctness can be found in [14].

3) Efficient Edmonds' algorithm complexity: In the contraction phase we mainly have two kinds of operations: priority queue operations and disjoint-set operations. The required

Algorithm 2: Expansion phase - P.M.Camerini correction

$B \leftarrow \emptyset$
while $R \neq N \neq \emptyset$ **do**
  1- If $R \neq \emptyset$, delete a root vertex v from R, else pick any root node $(u, v) \in N$ and add it to B.
  2- Identify path P in F leading from a root node to the leaf $(u, v) = \pi[v]$.
  3- Delete from F all nodes of P and all edges directed out of these nodes (this step updates the set N).
end
return B

priority queue operations are the following: $O(|V|)$ `INIT` operations, one for each vertex; $O(|E|)$ `EXTRACT-MIN`, $O(|V|)$ `MELD` operations. Hence, the total time for the priority queue operation is $O(|E| \log |V|)$. The disjoint-set required operations are: $O(|E|)$ `WFINDs`, $O(|E|)$ `SFINDs`, $O(|V|)$ `WUNIONs`, $O(|V|)$ `SUNIONs` and $O(|V|)$ `SADD-WEIGHT` operations. Let m denote the total number of disjoint-set operations as previously stated, the union-find data structure takes $O(m\alpha(|V|))$ time, where α is the inverse of the Ackerman function. The total time for other operations is $O(|E| + |V|)$. As for the expansion phase, the procedure presented in Algorithm 2 takes $0(|V|)$ since each node of F is visited exactly once. F contains does not contain more than $2|V| - 2$ nodes [2]. Thus, the total time required to find a minimum spanning arborescence is therefore dominated by the priority queue operations yielding a final temporal complexity of $O(|E| \log |V|)$.

D. Dynamic minimum spanning arborescence

From now one, assume that the input directed graph $G = (V, E)$ is strongly connected and that $w(e) > 0$, $\forall_e \in E$. If G is not strongly connected we can add a vertex v_∞ and $2|V|$ edges of ∞ weight, (v_∞, v_i) and (v_i, v_∞) for each $v_i \in V$ ensuring the connectivity of G.

Contributions [5], [6] rely in an **intermediary tree data structure** which encodes the content of set H, introduced in Section II-C. Afterwards, this intermediary tree data structure is processed accordingly to the edge operation yielding a partially contracted graph $G' = (V', E')$. Then an Edmonds' algorithm implementation is used over G'. Regarding the time complexity, both contribution rely in the concept of **output bounded complexity**. In this complexity model, the algorithms complexity of producing a novel solution is measured as a function of the number of **affected constituents** of the previous solution or of the intermediary data structure.

Before introducing more details, let $e = (u, v) \in E, t(e) = u \in V$ and $h(e) = v \in V$. Additionally, let regular nodes of the intermediary tree structure be represented by **simple nodes** while super-vertices are denoted by **c-nodes**. For the remaining of the current section, let simple nodes will be denoted by $N^s_{v_i}$, where $v_i \in V$ and c-nodes by N^c_j. Let the unsuperscripted N refer to the intermediary tree nodes regardless of their type.

1) Augmented tree data structure: Pollatos et al. [6] proposed a data structure named ATree (Augmented tree data structure) that encodes the set H, introduced in Section II-C, along with all vertices (c-vertices or super-vertices and simple ones) processed during the contraction phase of Edmonds' algorithm. Six records are maintained for each node N of the ATree:

1) $e(N)$: is the edge selected by Edmonds' algorithm for the represented vertex.
2) y_N: The cost of e at the time it was selected for the vertex represented by N.
3) $children(N)$: list holding the children of N in the ATree.
4) $\pi(N)$: is the parent node of N in the ATree.
5) $contracted - edges(N^c)$: is a list holding all edges contracted during the creation of the corresponding N^c.
6) $kind(N)$: simple or c-node.

Since G is strongly connected, all vertices will eventually be contracted to a single c-node by the end of the algorithm's execution. This c-node is represented by a root node of the ATree. The parent of each other node N is the intermediate c-node N^c to which it was contracted.

The ATree takes $O(2|V| + 1)$ space and its construction can be embedded into the Edmonds' algorithm implementation without affecting its complexity.

In Figure 2(a) is depicted a directed graph G with dashed lines that represent the cycles formed by Edmonds' algorithm. In Figure 2(b) is illustrated the ATree of G, the nodes of the ATree have the following edges associated: $e(N^s_0) = (1, 0)$, $e(N^s_1) = (0, 1)$, $e(N^c_{0,1}) = (2, 1)$, $e(N^s_2) = (3, 2)$, $e(N^s_3) = (2, 3)$ and $e(N^c_{2,3}) = (0, 3)$.

2) Operation `DELETE`: Let $e_{out} \in E$ be the edge to be removed. Two cases must be considered:

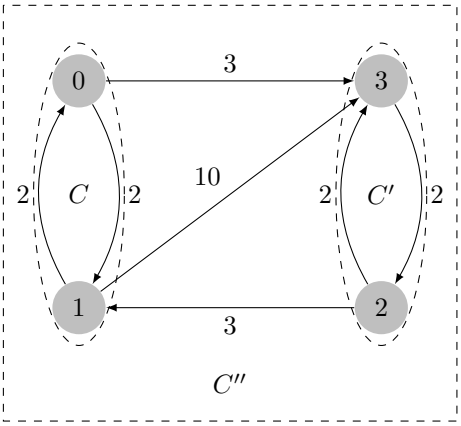
- $e_{out} \notin H$: remove e_{out} from G and from the $contracted - edges$ list to which e_{out} belongs.
- $e_{out} \in H$: decompose ATree, initialize Edmonds' algorithm with the remainders of the ATree and execute it.

The **decomposition** of the ATree begins from node N such that $e(N) = e_{out}$ and proceeds by following a path P from N to root of the ATree and removing all c-nodes on P except N. Each child of a removed c-node is made the root of its own sub-tree. Edge e_{out} must be removed $\pi(N)$, where $e(N) = e_{out}$, or from E' recognizing a partially contracted directed graph. In Figure 3(a) we can see the decomposed ATree after removing edge $(2, 3)$.

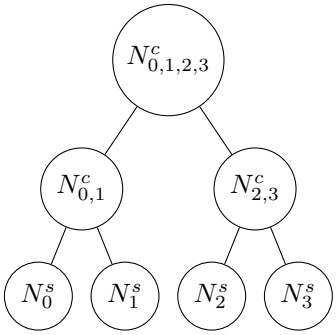
3) Recognizing a partially contracted directed graph: After decomposing the ATree, we proceed to recognize the partially contracted directed graph $G' = (V', E')$ represented by the remainders of the ATree. Let $V' = \{N_1 ... N_k\}$ be the roots of the ATrees after the decomposition. These will establish the vertex set of G'. A BFS (breadth first search) on each tree suffices to assign each original vertex v_i to some ATree root in V' in $O(|V|)$ time. Let $R = \{N^c_1 ... N^c_r\}$ be the set of c-nodes during the decomposition of the ATree. Note that the union of the $contracted - edges$ list of each $N^c_k \in R, 1 \leq k \leq r$

Algorithm 3: Efficient Edmonds' algorithm - Contraction phase

$roots \leftarrow \emptyset$
foreach *node* $v \in V$ **do**
  INIT $(v, L[v])$
  SMAKE-SET (v)
  WMAKE-SET (v)
  $roots = roots \cup \{i\}$
  $max[i] = i$
  $inEdgeNode[i] = null$
end
$F \leftarrow \emptyset$
$H \leftarrow \emptyset$
while $roots \neq \emptyset$ **do**
  $root \leftarrow$ POP$(roots)$
  $h \leftarrow queues[root]$
  if $h = \emptyset$ **then**
    CONTINUE
  end
  $(u, v) \leftarrow$ EXTRACT-MIN (h)
  while $h \neq \emptyset \wedge$ SFIND$(u) =$ SFIND(v) **do**
    $(u, v) \leftarrow$ EXTRACT-MIN (h)
  end
  if SFIND$(u) =$ SFIND(v) **then**
    CONTINUE
  end
  $H \cup \{(u, v)\}$
  Let $minEdgeNode$ be a node in Forest F associated with edge (u, v)
  if $edgeNodeCycle[root] = null$ **then**
    $\pi[root] \leftarrow minEdgeNode$
  end
  else
    foreach *node* $n \in$ CHILDREN$(minEdgeNode)$ **do**
      PARENT $(n) \leftarrow minEdgeNode$
      CHILDREN $(minEdgeNode) \cup \{n\}$
    end
  end
  if WFIND *(u)* $\neq$ WFIND *(v)* **then**
    $inEdgeNode[v] \leftarrow minEdgeNode$
    WUNION (u, v)
  end
  else
    $inEdgeNode[root] \leftarrow null$
    $cycle \leftarrow \emptyset, cycle \cup minEdgeNode$
    Let map denote a dictionary
    for $i \leftarrow$ SFIND *(u); $inEdgeNode[i] \neq null$; $i \leftarrow$* SFIND *(u (*EDGE *($inEdgeNode[i]$)))* **do**
      $cycle \cup inEdgeNode[i]$
      PUT $(map, i,$ EDGE$(inEdgeNode[i]))$
    end
    Let w_{max} denote the weight of maximum weight edge in $cycle$ and max the edge
    foreach *node* $n \in cycle$ **do**
      $reduction \leftarrow w($GET $(map, n))$ - w_{max}
      SADD-WEIGHT $(n, reduction)$
    end
    foreach *edge* $(u, v) \in cycle$ **do**
      SUNION (u, v)
    end
    $rep \leftarrow$ SFIND$(v(max))$
    $roots \cup rep$
    $max[rep] = max[v(max)]$
    foreach *node* $v \in cycle$ **do**
      if $v \neq rep$ **then**
        MERGE $(queues[rep], queues[v])$
      end
    end
  end
end



(a) Directed graph G. Dashed lines represent cycles formed by Edmonds' algorithm.



(b) ATree of G.

Fig. 2. In Figure 2(a) is represented a directed graph G, the dashed lines represent cycles formed by Edmonds' algorithm. In Figure 2(b) we present the ATree of G.

is precisely the edge set E' which can be found in $O(|V|)$. In Figure 3(b) we present the partially contracted graph G' after removing edge $(2, 3)$.

Nevertheless, given a partially contracted directed graph G', each $N \in V'$ associated with a set of incoming edges $\phi_{E'}(N)$, the proper reduced costs must be assigned. Let $v_i \in V$ be a vertex of the original directed graph represented as a leaf node N^s_i of an ATree with root $N \in V'$. Let $e = (u, v_i)$ with $e \in \phi_E(v_i) \cap e \in \phi_{E'}(N)$. Let $P = \{N_i, N^c_1, ...N^c_l, N\}$ denote the path from N_i to the root N in the ATree and by functionality of Edmonds' algorithm we can determine the reduced cost of e:

$$\hat{w}(e) = w(e) - \sum_{N \in P} y_N \tag{2}$$

where $\hat{w}(e)$ represents the current cost of e and denotes $w(e)$ the original cost.

Let r_i denote the subtracted sum for each simple node N^s_i of the ATree. In order to find the reduction quantity r_i we run a single BFS on each tree in a total of $O(|V|)$ time. Then proceed to scan the edges $e = (u, v_i) \in E'$ and assign the proper reduced cost $\hat{w}(e) = w(e) - r_i$.

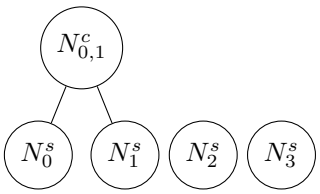
4) Operation `ADD`*:* Adding a new edge e_{in} is handled by reducing the problem to an edge deletion instance. First it is necessary to check whether e_{in} should replace some edge encoded in the ATree. This process involves c-nodes that are ancestors of $N^s_{h(e_{in})}$ and $N^s_{h(e_{in})}$. This process can be performed as follows: starting from node $N^s_{h(e_{in})}$ follow the path towards the root of the ATree. For each visited node N verify whether $w'(e(N)) > w'(e_{in})$, where $w'(N)$ denotes the reduced costs. If the previous condition does not hold then proceed to the parent of N. Otherwise, a candidate node N has been found which should have e_{in} as its selected edge, since e_{in} has lower cost. Maybe the case that the root node of the ATree is reached implying that e_{in} cannot replace any edge in H and subsequently in the ATree. In this case insert e_{in} in G and in the $contracted - edges$ list associated with the LCA (lowest common ancestor) of $N^s_{h(e_{in})}$ and $N^s_{t(e_{in})}$.

Considering that we have found a candidate node N which should replace its $e(N)$ with e_{in}, we need to determine if e_{in} is safe to be added. To do so examine whether $N^s_{t(e_{in})}$ is hanged over the sub-tree of the candidate node N, by engaging a BFS. If $N^s_{t(e_{in})}$ is found it is implied that e_{in} should not belong, so insert e_{in} in G and in the $contracted - edges$ list associated with the LCA of $N^s_{h(e_{in})}$ and $N^s_{t(e_{in})}$. Otherwise, insert e_{in} in G and engage a **virtual deletion** of $e(N)$, without actually removing $e(N)$ from the G. After this virtual edge deletion the recognition of $G' = (V', E')$ is carried out and Edmonds' algorithm is executed over $G' = (V', E' \cup e_{in})$.

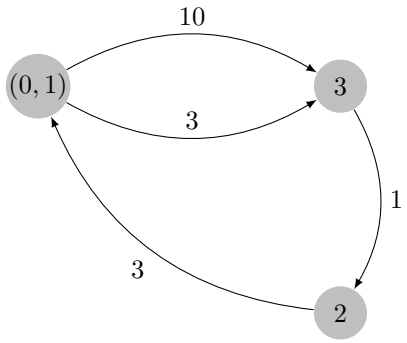
E. Dynamic minimum spanning arborescence complexity

In order to study the output complexity of this dynamic scheme, it is necessary to identify the minimal portion of the maintained output that is affected by each edge operation. The output consists of all processed nodes. A node is affected if it takes part in a different contraction in the new output after an edge operation. A contraction is defined exactly by the vertices and edges that comprise the directed cycle which caused it. The set of affected vertices is denoted by ρ and its size $|\rho|$. The extended set of affected elements (namely vertices and edge incoming to the affected vertices) is denoted as $||p||$. Accordingly to the results presented by Pollatos et al. [6] the root nodes of a decomposed ATree represent the affected vertices. Notice that $|\rho| \leq |V|$ and that during edge insertion or deletion all the supplementary operations occur in $O(|V|)$ complexity, while re-execution of Edmonds' algorithm processes only the affected vertices.

Hence by results presented by Pollatos et al. [6] and assuming the implementation of Edmonds' algorithm presented in Section II-C the fully dynamic maintenance of a minimum spanning arborescence can be achieved in $O(|V|+||\rho|| \log |\rho|)$ output complexity by edge operation.



(a) Decomposed ATree after removing edge $(2, 3)$.



(b) Partially contracted graph G' after removing edge $e_{out} = (2, 3)$.

Fig. 3. Decomposed ATree after removing edge $(2, 3)$ and corresponding partially contracted graph G'.

TABLE I
PHYLOGENETIC DATA SET.

Data set	$\|V\|$	$\|E\|$
`Yersinia.wgMLST`	3073	9443329
`Salmonella.Achtman7GeneMLST`	5464	29855296

III. RESULTS

A. Experimental protocol

1) Environment: All of the code was developed recurring to `Java 8` high level programming language, more specifically:

- `openjdk version "1.8.0_212"`
- `OpenJDK Runtime Environment (build 1.8.0_212-8u212-b01-1 deb9u1-b01)`
- `OpenJDK 64-Bit Server VM (build 25.212-b01, mixed mode)`

As for the `Java` compiler the used version was `javac 1.8.0_212`.

All the benchmarks were taken on a server running `Debian 9` with the following hardware: `Intel(R) Xeon(R) CPU E7-4830 @2.13GHz` and `256 GB` of RAM.

2) Data set: In order to evaluate the developed work we considered the following graph models:

- ER (Erdős–Rényi) model with $p = \frac{c \log |V|}{|V|}, c \geq 1$, where p denotes the probability of linking a node u with a node v and $|V|$ is the number of nodes in the network. Whenever p has the previously defined value, the network has one giant component and some isolated nodes.
- Scale-free network is a connected graph with property that the number of edges originating from a given none exhibits a power law distribution. In this particular case the probability of linking of node u being connected to a node v is proportional to the number of edges of v.
- Complete graphs.

The following phylogenetic data sets were considered:

- `Yersinia.wgMLST`
- `Salmonella.Achtman7GeneMLST`

B. Edmonds' algorithm with different priority queue representation

In Figure 4 is presented a temporal scalability test on the impact of the priority queue representation in Edmonds' algorithm, namely: a binary and binomial heap. In Figures 4(a) and 4(c) we can see that the binomial heap performed better than the binary heap for sparse and complete graphs as excepted. Recall that operation `MELD` in a binomial heap takes logarithmic time while in a binary heap takes linear time. In Figure 4(b) both priority queues had a similar behavior since scale-free network are extremely sparse. All in all, our implementation of Edmonds' algorithm has shown to have the expected temporal complexity.

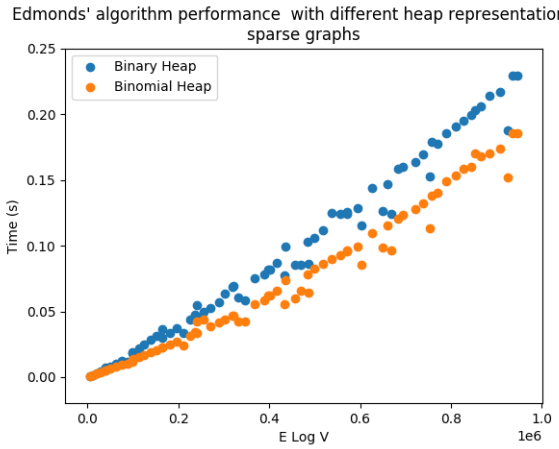
C. Efficient Edmonds' algorithm vs original description description

In Figure 5 we compare a naive inefficient polynomial time implementation of Edmonds' algorithm, discussed in Section II-B, with our efficient implementation. As expected, the efficient version performed much better than the naive implementation for complete and sparse graphs.

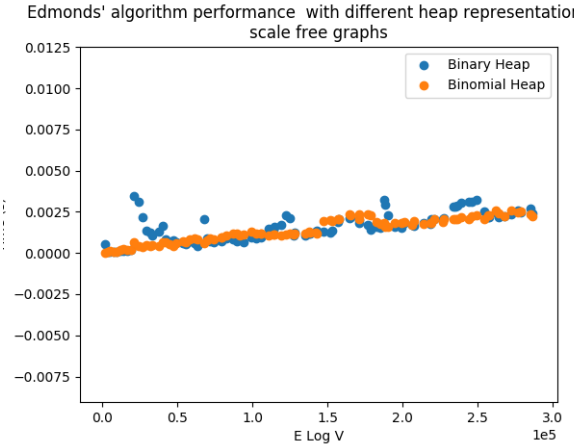
D. Dynamic minimum spanning arborescence evaluation

This section introduces a temporal scalability study over our implementation of DMSA (dynamic minimum spanning arborescence) comparing it to a static version of Edmonds' algorithm. Our data set was composed only with complete graphs since the DMSA has the constraint of only operating with strongly connected graphs and a complete graph is always strongly connected.

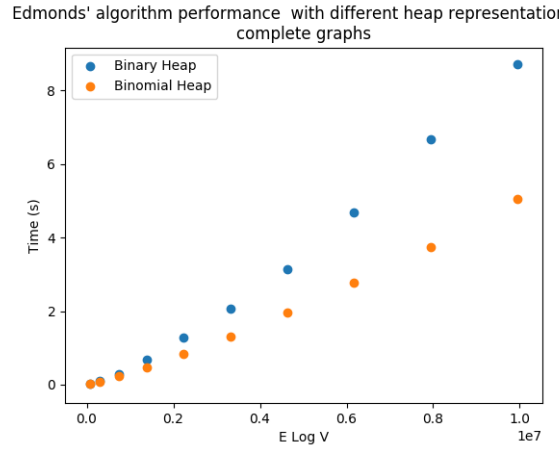
To evaluate the temporal scalability of the operation `DELETE` implementation we have considered the average time to process a sequence of $m = 10$ `DELETE` operations. That sequence was randomly selected. The temporal curves can be seen in Figure 6(a), between $[1.5 \times 10^6; 3.5 \times 10^6]$ the DMSA has shown to perform twice as fast. In Figures 7(a) and 8(b) is the presented the temporal scalability of operation `DELETE` using the following phylogenetic data sets: *Salmonella.Achtman7GeneMLST* and *Yersinia.wgMLST*



(a) Edmonds' algorithm performance with different priority queue representation on directed sparse graphs generated with ER model.

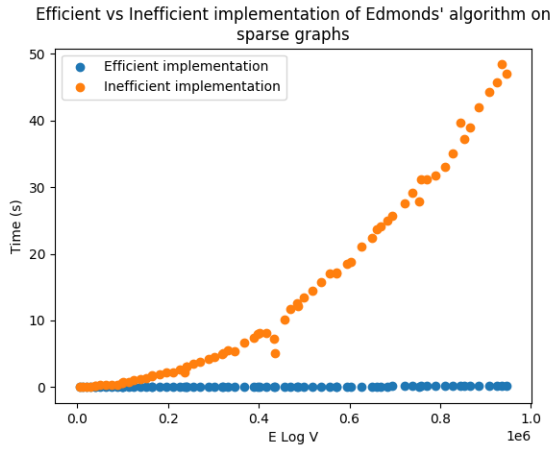


(b) Edmonds' algorithm performance with different priority queue representation on scale-free graphs.

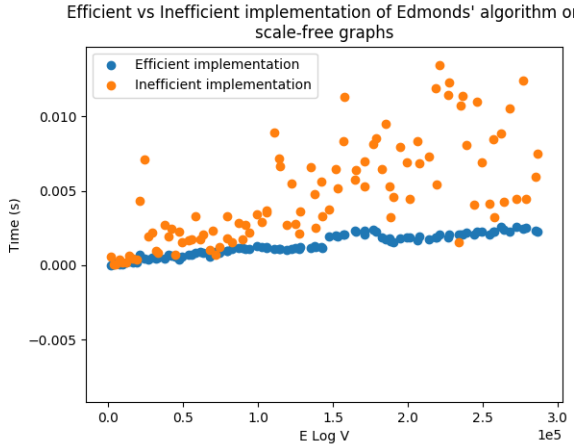


(c) Edmonds' algorithm performance with different priority queue representation on directed complete graphs.

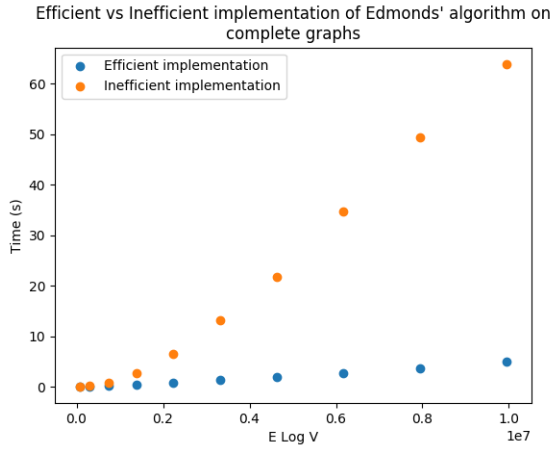
Fig. 4. Edmonds' algorithm performance with different priority queue representation.



(a) Efficient vs inefficient Edmonds' algorithm implementation on directed sparse graphs generated with ER model.

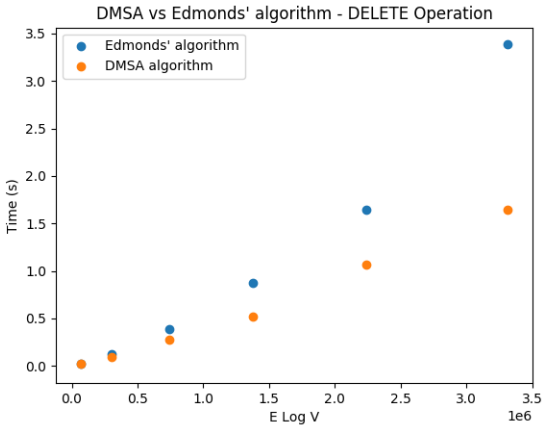


(b) Efficient vs inefficient Edmonds' algorithm implementation on scale-free graphs.

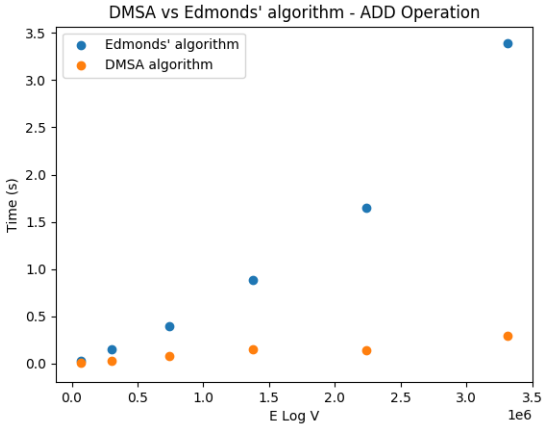


(c) Efficient vs inefficient Edmonds' algorithm implementation on complete graphs.

Fig. 5. Performance evaluation of between an efficient implementation and a polynomial implementation of Edmonds' algorithm.



(a) DMSA vs static Edmonds' over m DELETE operations on complete graphs.



(b) DMSA vs static Edmonds' over m ADD operations on complete graphs.

Fig. 6. DMSA vs static Edmonds' over m operations on complete graphs.

considering $[10\%, 20\%, 30\%, ..., 100\%]$ of number of nodes of the data set. As the size of the input graph grows the DMSA starts performing better being twice as fast as the static version of Edmond's algorithm.

In order to access the performance of the operation ADD implementation we have considered the averaged time to perform a sequence of $m = 10$ ADD operations. Those sequences were randomly generated. The weights also were randomly selected between interval $[0; \max(w(E))]$, where $max(w(E))$ denotes the weight of the heaviest edge of an input graph. The performance study is presented in Figure 6(b) and as can perceived the static version of Edmonds' algorithm is clearly outmatched by the DMSA as the size of the input graph grows. In Figures 7(b) and 8(b) is depicted the temporal scalability of operation ADD taking into account phylogenetic data sets. As the size of the input graph grows the DMSA starts performing better being at least twice as fast as the static version of Edmond's algorithm.

TABLE II
MEMORY CONSUMPTION COMPARISON BETWEEN STATIC EDMONDS' ALGORITHM AND DMSA ON *Yersinia.wgMLST*.

Data set %	Static Edmonds' algorithm (MegaByte)	DMSA (MegaByte)	Memory ratio
10	125.32	415.93	3.32
20	156.31	513.5	3.29
30	214.65	680.62	3.17
40	295.80	908.01	3.07
50	411.26	1201.33	2.92
60	534.45	1585.99	2.97
70	714.39	2056.44	2.88
80	898.64	2479.41	2.76
90	1098.49	3026.20	2.75
100	1306.95	3640.20	2.79

E. Memory evaluation

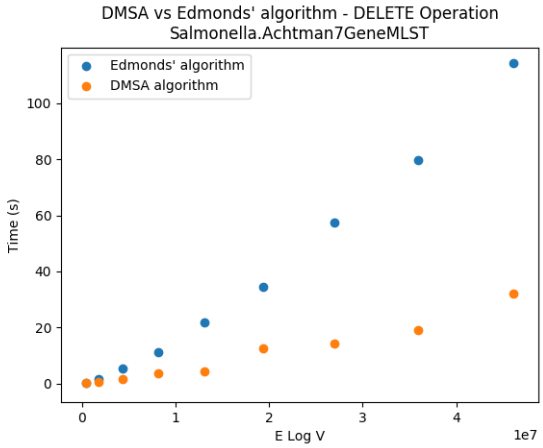
In this section we analyze the memory requirements of our implementation of Edmonds' algorithm on the phylogenetic data sets. Still, in the current section, the memory requirements of the DMSA are discussed.

In order to measure the memory consumption we used the following functions: `getHeapMemoryUsage`, `getNonHeapMemoryUsage` from the `Java 8` built-in class `MemoryMXBean` and performed the sum of the output of each function. Before executing those functions we instructed the garbage collector to be executed. The presented results are averaged of 5 runs.

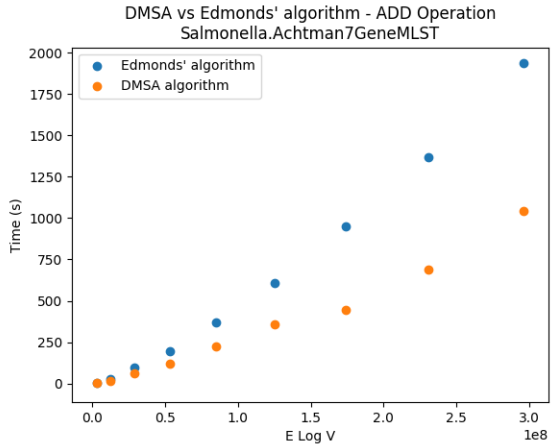
In Figure 9(a) is depicted the memory consumption of our efficient implementation of Edmonds' algorithm. In each figure, is possible to assess that the memory consumption grows linearly as expected.

In Figure 4 is presented a temporal scalability test on the impact of the priority queue representation in Edmonds' algorithm, namely: a binary and binomial heap. In Figures 4(a) and 4(c) we can see that the binomial heap performed better than the binary heap for sparse and complete graphs as excepted. Recall that operation MELD in a binomial heap takes logarithmic time while in a binary heap takes linear time. In Figure 4(b) both priority queues had a similar behavior since scale-free network are extremely sparse. All in all, our implementation of Edmonds' algorithm has shown to have the expected temporal complexity.

In Tables II and III we present a memory consumption comparison between our DMSA implementation and the static version of Edmonds' algorithm over m operations. In the first column, of each table, we have the % considered of the data set, the second column presents the consumed memory of the static Edmonds' algorithm, in the third column we have the required memory of our implementation of the DMSA and finally the fourth column presents the memory ratio between the DMSA and the static Edmonds' algorithm. With the presented results, we can determine that the memory consumption our DMSA grows linearly, which is acceptable since we are supporting a dynamic structure. Moreover, our implementation of the DMSA consumes in average three times as much memory as the static Edmonds' algorithm.

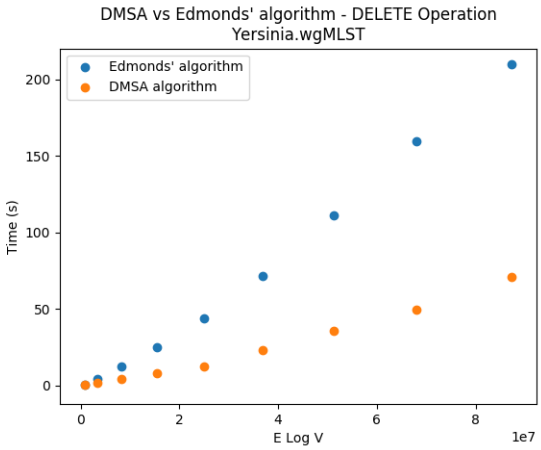


(a) DMSA vs static Edmonds' over m `DELETE` operations on *Salmonella.Achtman7GeneMLST*.

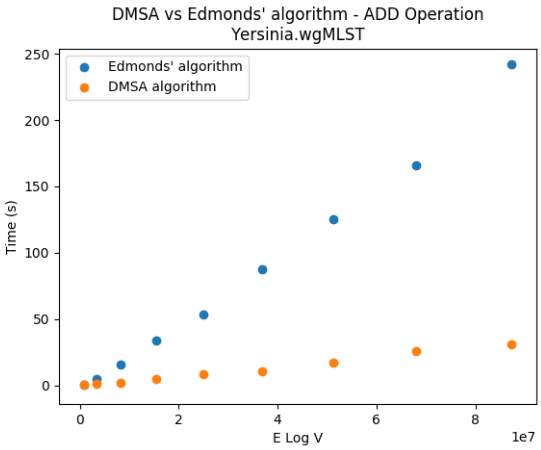


(b) DMSA vs static Edmonds' over m `ADD` operations on *Salmonella.Achtman7GeneMLST*.

Fig. 7. DMSA vs static Edmonds' over m operations on *Salmonella.Achtman7GeneMLST*.



(a) DMSA vs static Edmonds' over m `DELETE` operations on *Yersinia.wgMLST*.



(b) DMSA vs static Edmonds' over m `ADD` operations on *Yersinia.wgMLST*.

Fig. 8. DMSA vs static Edmonds' over m operations on *Yersinia.wgMLST*.

TABLE III
MEMORY CONSUMPTION COMPARISON BETWEEN STATIC EDMONDS' ALGORITHM AND DMSA ON *Salmonella.Achtman7GeneMLST*.

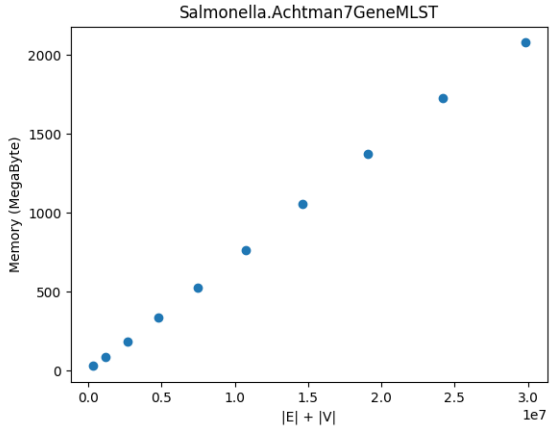
Data set %	Static Edmonds' algorithm (MegaByte)	DMSA (MegaByte)	Memory ratio
10	62.60	210.45	3.36
20	154.52	520.83	3.37
30	319.87	1041.22	3.26
40	575.05	1772.00	5.54
50	886.04	2667.96	3.01
60	1274.59	3883.46	3.04
70	1754.89	5242.21	2.99
80	2285.91	6841.03	2.99
90	2872.80	8574.12	2.99
100	3435.51	10482.84	3.05

IV. CONCLUSIONS

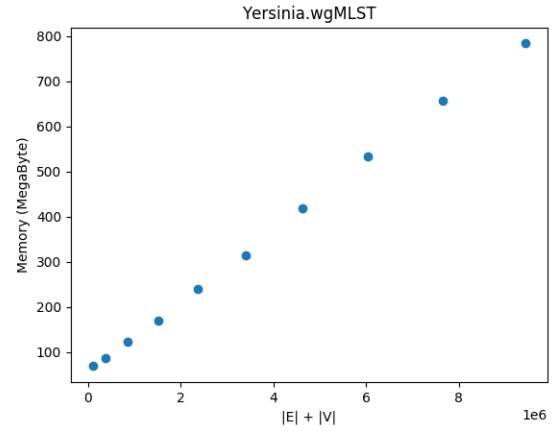
Our implementation of Edmonds' algorithm can be improved relying in some ideas provided by Gabow et al. in [3]. Some lists are necessary to keep track of the candidate edges for selection and to allow easy deletion of multiple edges. For each vertex v_i an exit list is maintained: the first edge in an exit list is called an active edge and the other edges are called passive, also every vertex v_i has a passive set containing every edge incident in vertex v_i. Every time a cycle $C = \{v_0, v_1, ..v_k\}$ is contracted, every edge from the passive list $v_i, 0 \leq i \leq k$ is deleted. Regarding the exit list remove the currently active edge or the edge that has the larger cost breaking a tie arbitrarily. Considering that the passive sets and exit list are represented by doubly linked lists the total time of manipulating the lists is linear. The priority queues are represented by Fibonacci heaps which are updated after manip-

ulating the disjoint set data structure to compute the reduced costs. After computing the reduced costs, the Fibonacci heaps associated with $v_0, v_1, ..v_k$ are melded together. Operation `MELD` between two Fibonacci heaps takes $O(1)$ amortized time while binomial heaps take $O(\log n)$ worst-case, where n denotes the sum of the number of elements in both heaps. All in all Gabow et al. [3] aim to reduce the number of insertions and deletions in the heaps. Taking into account the previous description, it is possible to lower the temporal complexity from $O(|E| \log |V|)$ to $O(|V| \log |E| + |V|)$.

Regarding the dynamic minimum spanning arborescence mainly two considerable shortcomings were found: the first drawback is that the algorithm only takes as input a strongly connected graphs, the second weak point is that the time needed to recalculate the minimum spanning arborescence is highly dependent on the affected level of the ATree. Lower the level larger the number of affected constituents. Another prospect to achieve an efficient dynamic minimum spanning arborescence algorithm could be relying in the data structure known as *link-cut trees* [15] which essentially maintains a collection of node-disjoint forests of self-adjusting binary heaps known as *splay trees* under a sequence of two kinds of operations `LINK` and `CUT`. Both operation take $O(\log n)$ worst-case.



(a) Memory consumption of the efficient implementation of Edmonds' algorithm on *Salmonella.Achtman7GeneMLST*.



(b) Memory consumption of the efficient implementation of Edmonds' algorithm on *Yersinia.wgMLST*.

Fig. 9. Memory requirements of the efficient implementation of Edmonds' algorithm on the phylogenetic data set.

References

[1] Zhemin Zhou, Nabil-Fareed Alikhan, Martin J. Sergeant, Nina Luhmann, Cátia Vaz, Alexandre P. Francisco, João André Carriço, and Mark Achtman. Grapetree: Visualization of core genomic relationships among 100,000 bacterial pathogens. *bioRxiv*, 2017.

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

[3] Harold N. Gabow, Zvi Galil, Thomas Spencer, and Robert E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, jun 1986.

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

[5] Orestis Telelis and Vassilis Zissimopoulos. Fully dynamic maintenance of optimum directed spanning forests. 06 2019.

[6] Gerasimos G. Pollatos, Orestis A. Telelis, and Vassilis Zissimopoulos. Updating directed minimum cost spanning trees. In Carme Àlvarez and María Serna, editors, *Experimental Algorithms*, pages 291–302, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[7] J. Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards*, (71B:233-240), 1967.

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

[9] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14, 1965.

[10] R. M. Karp. A simple derivation of edmonds' algorithm for um branchings. *Networks*, 1(3):265–272, 1971.

[11] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1(1):127–136, Dec 1971.

[12] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[13] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, April 1978.

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

[15] Ming-Yang Kao. *Encyclopedia of Algorithms*. Springer Publishing Company, Incorporated, 2nd edition, 2016.