Fixed-parameter tractability of the Maximum Agreement Supertree problem

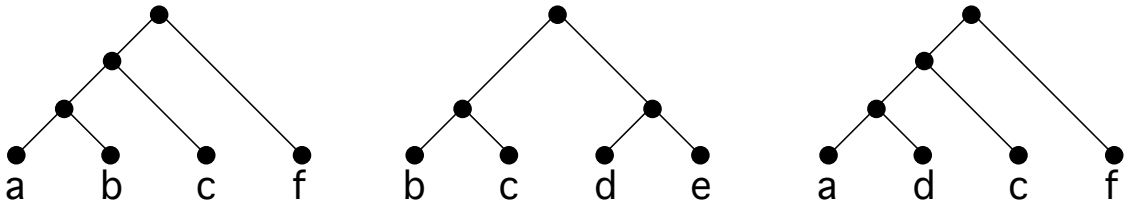
Sylvain Guillemot, Vincent Berry

Équipe Méthodes et Algorithmes pour la Bioinformatique
LIRMM - CNRS - Université Montpellier II
Montpellier, France

11th July 2007

Setting the scene

The collection $\mathcal{T}$

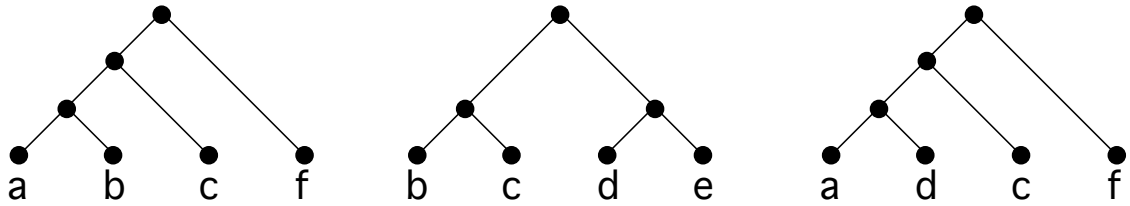


We consider a collection $\mathcal{T} = \{T_1, ..., T_k\}$ of bijectively leaf-labeled trees with partially overlapping label sets. The union of their label sets is denoted by Ł.

Example. A collection $\mathcal{T}$ with $L = \{a, b, c, d, e, f\}$.

Supertree construction

The collection $\mathcal{T}$



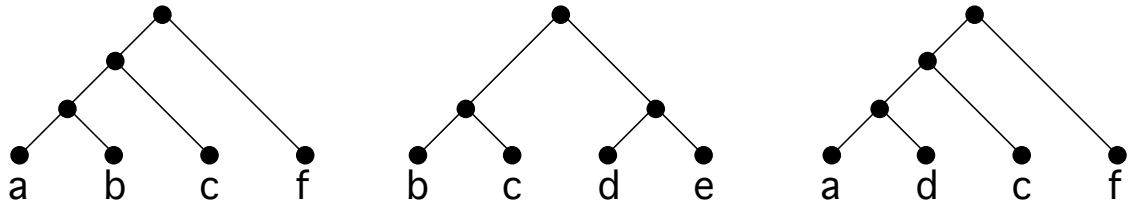
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



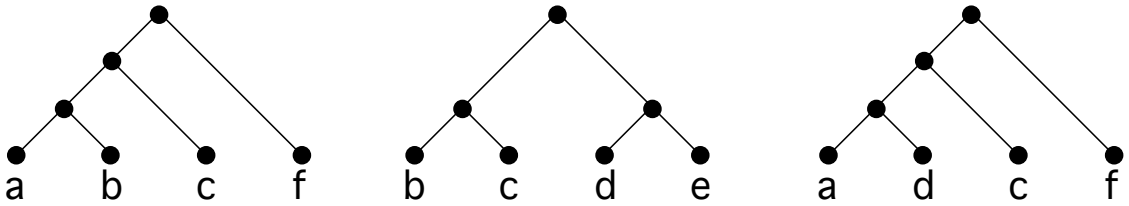
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



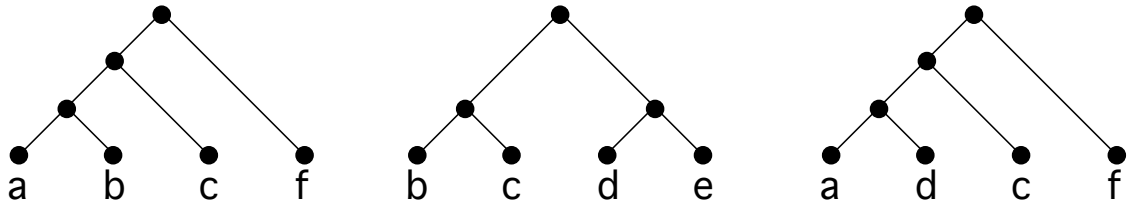
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



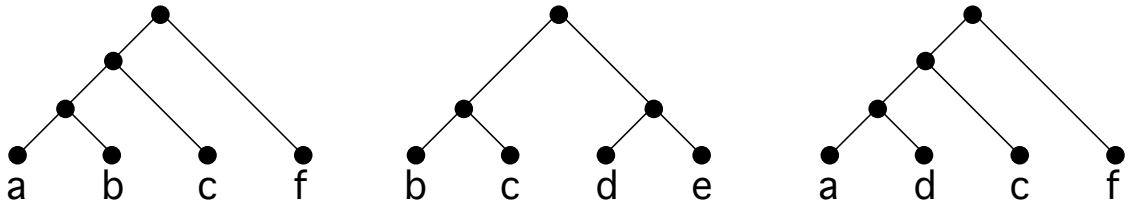
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



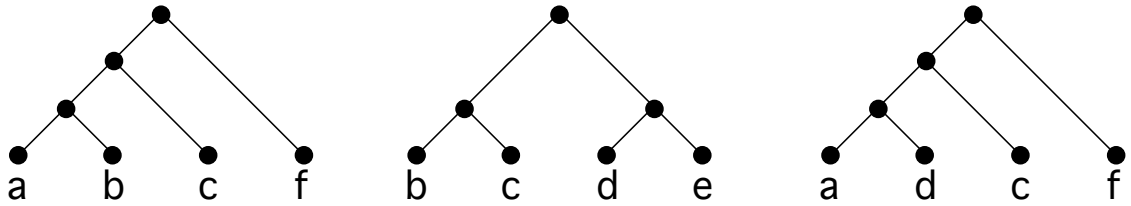
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



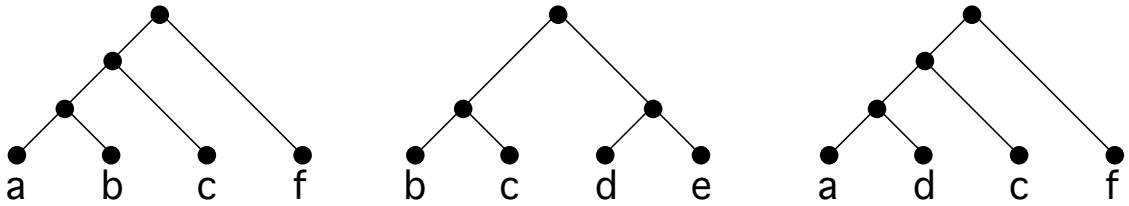
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



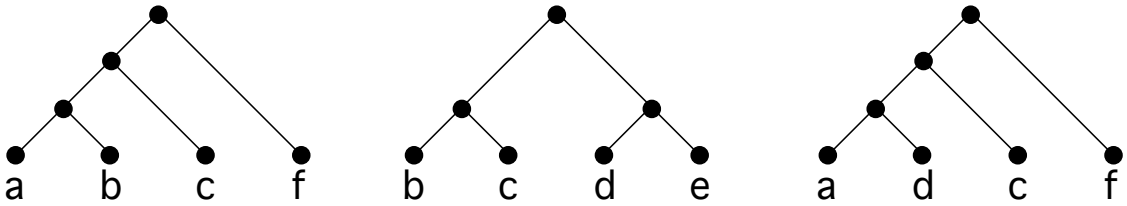
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



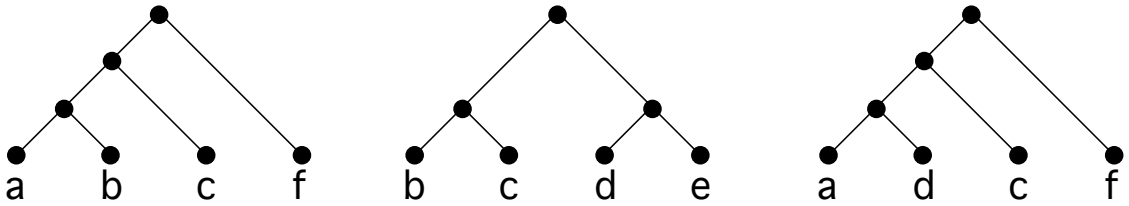
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



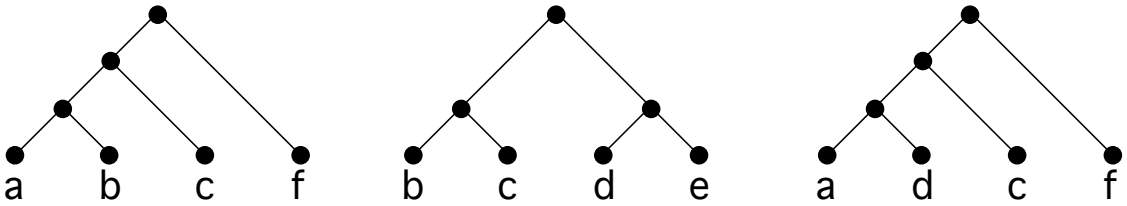
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



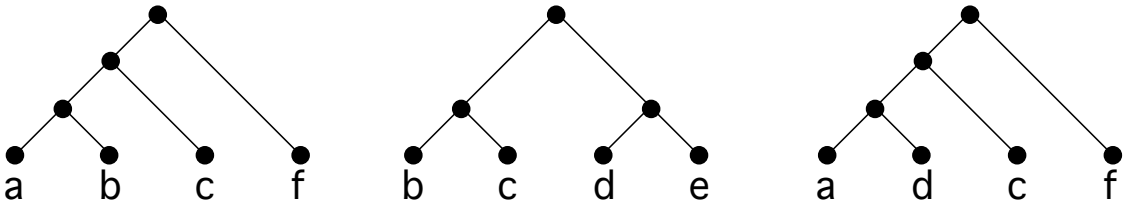
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Supertree construction

The collection $\mathcal{T}$



The tree T

?

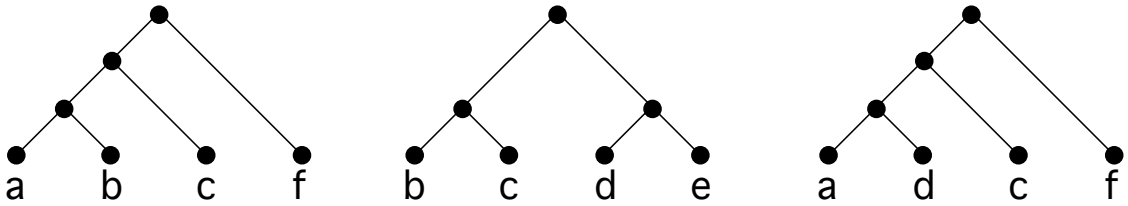
The problem:
given $\mathcal{T}$, find a single tree T with labels in L, which complies as much as possible with the input trees.

Applications:

- in phylogenetics [Bininda-Edmonds et al.]: merge several phylogenies obtained by different means into a single one;
- optimizing queries in databases [Aho et al. 81].

Agreement supertrees

The collection $\mathcal{T}$



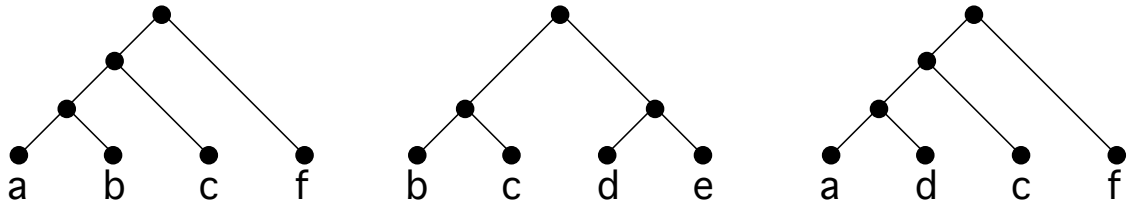
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



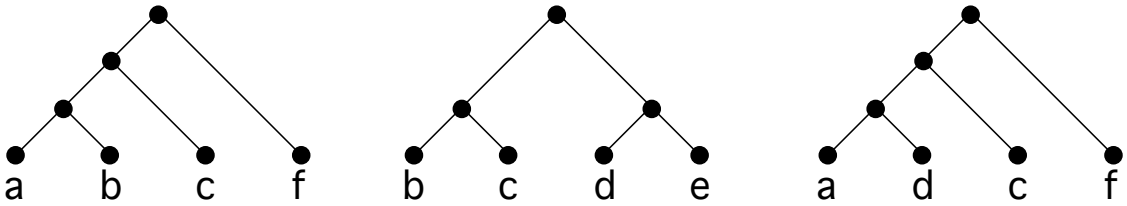
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



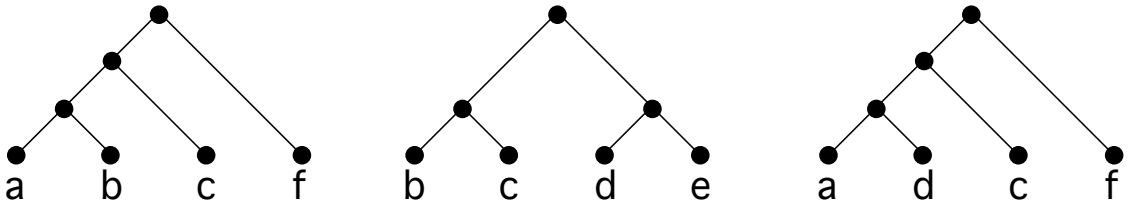
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



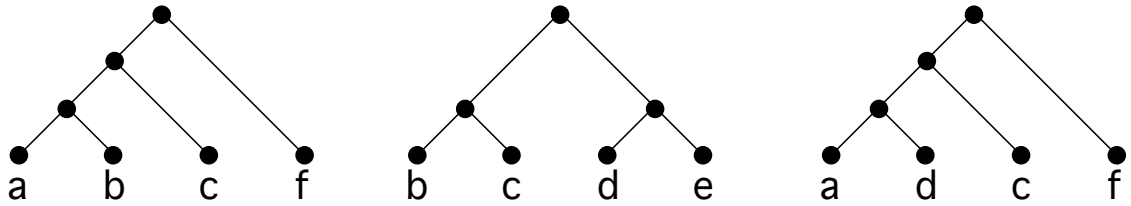
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



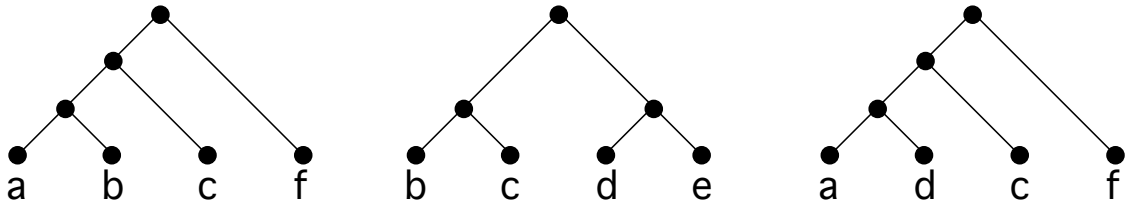
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



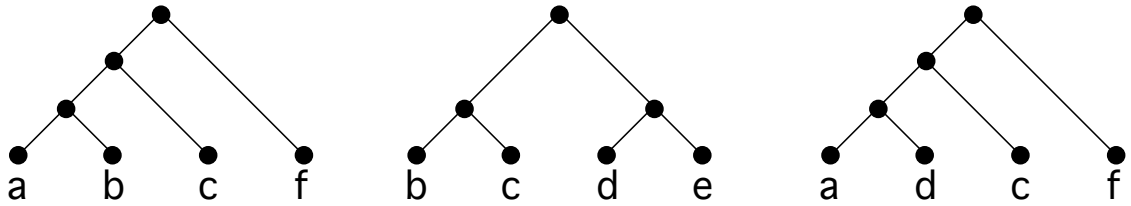
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



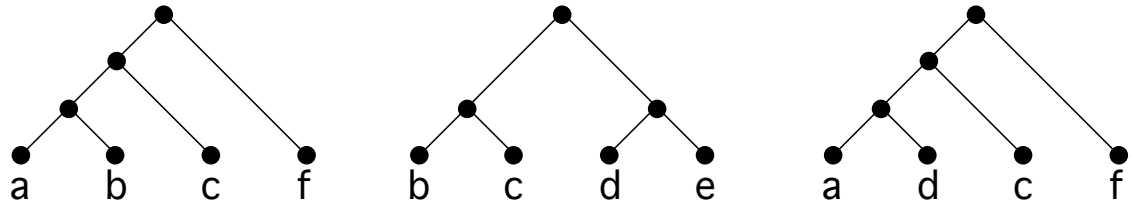
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



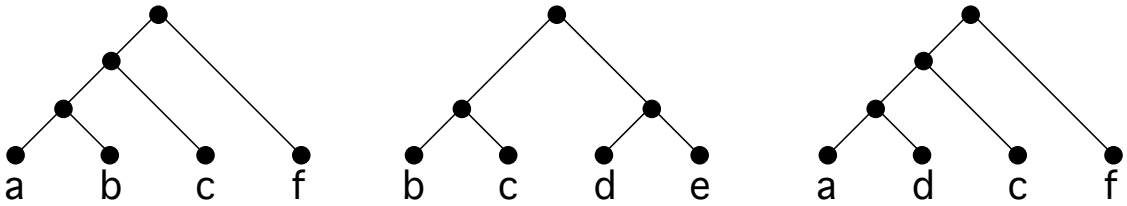
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



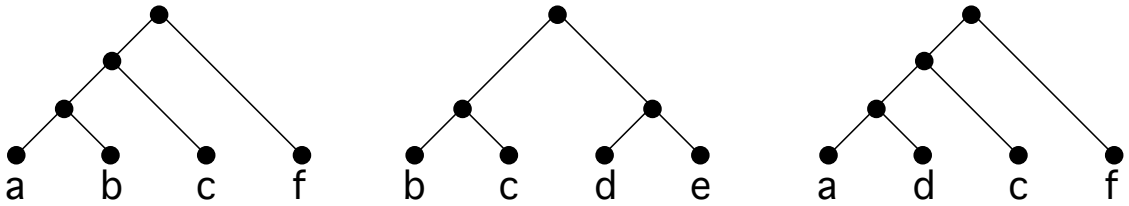
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



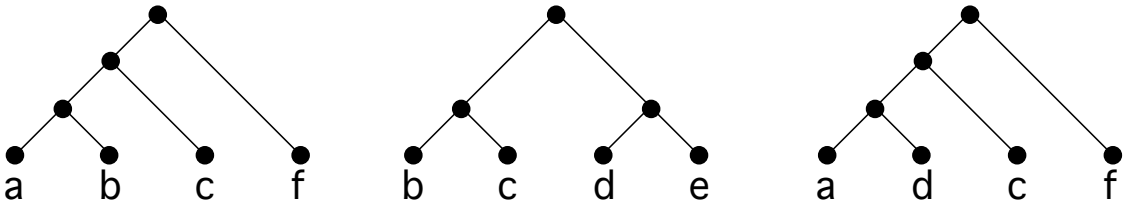
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

Example.

Agreement supertrees

The collection $\mathcal{T}$



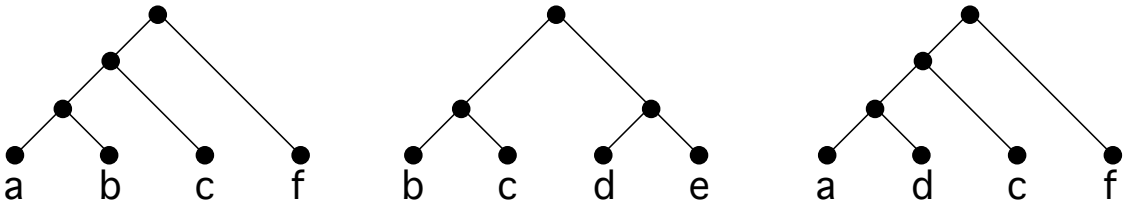
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

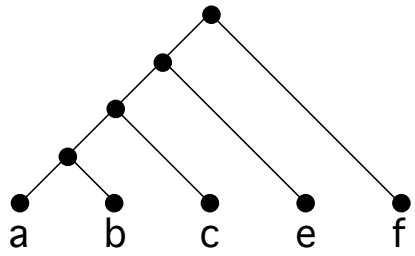
Example.

Agreement supertrees

The collection $\mathcal{T}$



The tree T



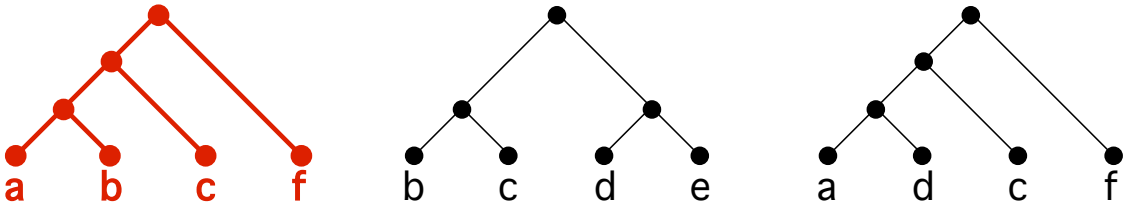
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

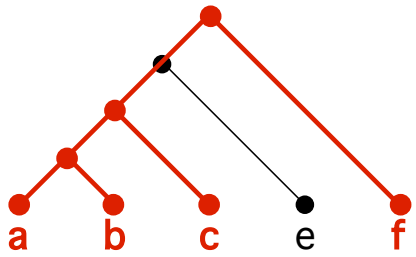
Example. T is an agreement supertree for $\mathcal{T}$.

Agreement supertrees

The collection $\mathcal{T}$



The tree T



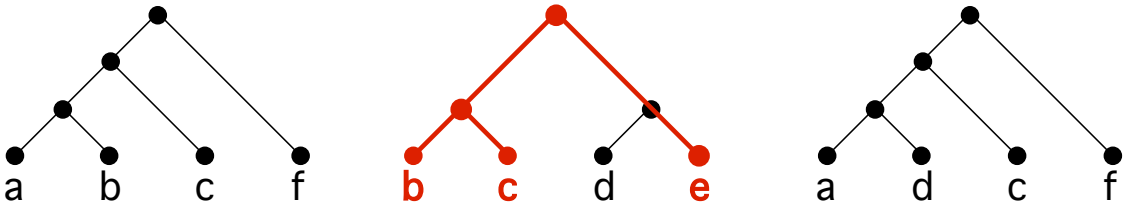
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

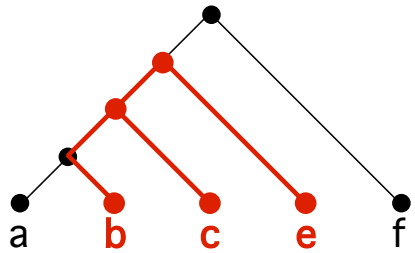
Example. T is an agreement supertree for $\mathcal{T}$.

Agreement supertrees

The collection $\mathcal{T}$



The tree T



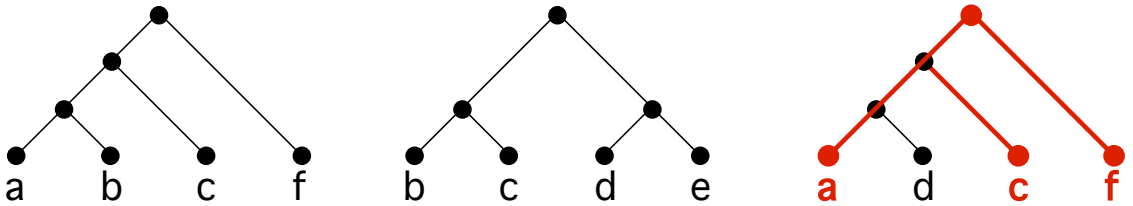
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

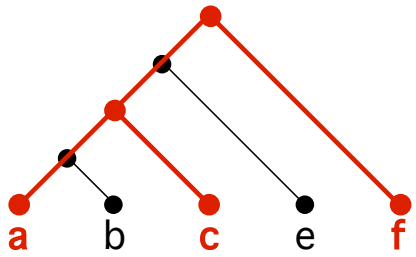
Example. T is an agreement supertree for $\mathcal{T}$.

Agreement supertrees

The collection $\mathcal{T}$



The tree T



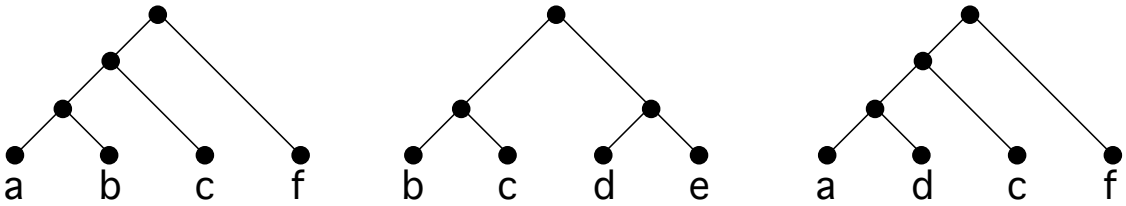
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

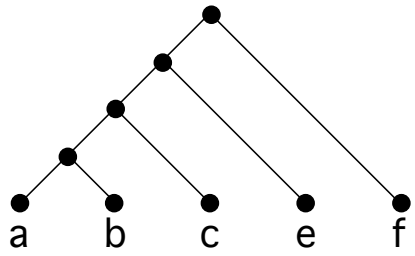
Example. T is an agreement supertree for $\mathcal{T}$.

Agreement supertrees

The collection $\mathcal{T}$



The tree T



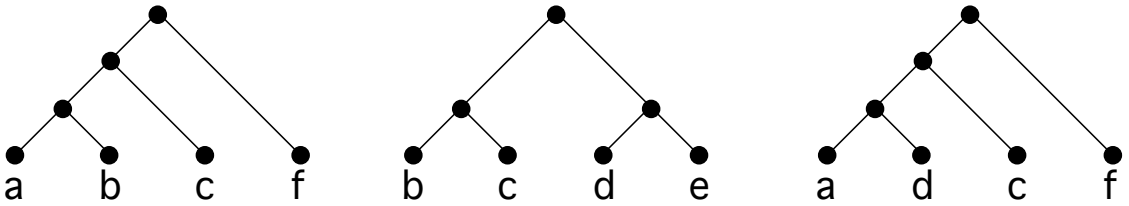
T *agrees* with T' iff they indicate the same relationships for their common labels. An *agreement supertree* for $\mathcal{T}$ is a tree T with labels in L, which agrees with every T_i.

An agreement supertree is *total* if it contains every label from L. $\mathcal{T}$ is *compatible* if it admits a total agreement supertree.

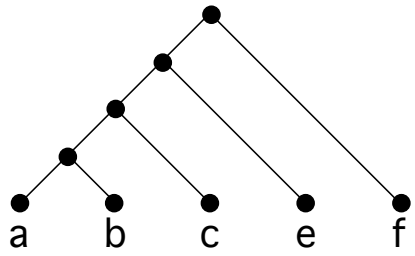
Example. T is an agreement supertree for $\mathcal{T}$.

The Maximum Agreement Supertree problem

The collection $\mathcal{T}$



The tree T



A *maximum agreement supertree* is an agreement supertree with the maximum number of labels. Its size is denoted $\#SMAST(\mathcal{T})$.

The Maximum Agreement Supertree problem (Smast, [Berry et al. 04], [Jansson et al. 04]) seeks a maximum agreement supertree for an input collection $\mathcal{T}$.

Example: here $\#SMAST(\mathcal{T}) = 5$.

Previous results

Previous results for SMAST on binary trees:

- SMAST is NP-hard;
- SMAST is solvable in subquadratic time for two trees [Berry et al. 04], and is solvable in $O(n^{3k^2})$ time for k trees [Jansson et al. 04];
- SMAST is as hard to approximate as MAXIMUM CLIQUE, and is approximable within $O(\frac{n}{\log n})$ [Jansson et al. 04] using approximation via partitioning;
- the complementary minimization problem is as hard to approximate as MINIMUM SET COVER, hence not approximable within $\Omega(\log n)$ unless P = NP [Berry et al. 04].

New results

Our new results are as follows:

- a bounded-search algorithm with running time $O((2k)^p \times kn^2)$, where p is an upper bound on the number of labels missing in a SMAST solution;
- a dynamic-programming algorithm with running time $O((8n)^k)$;
- an algorithm solving SMAST on complete collections of triples in time $O(4^p n^3)$;
- hardness results for several parameterizations of SMAST.

A bounded-search algorithm for SMAST

Theorem: SMAST can be solved in $O((2k)^p \times kn^2)$ time.

Observation: the problem can be defined equivalently as:

- is there an agreement supertree with $\geq n - p$ labels?
- is there a set $L' \subseteq L$ of size $\leq p$ s.t. $\mathcal{T} \backslash L'$ is compatible?

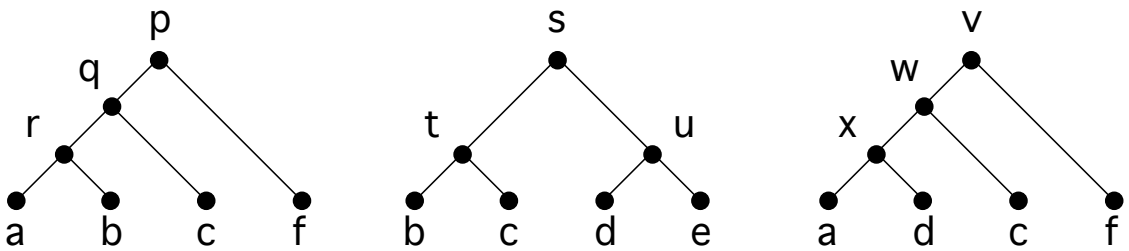
Proposition: there is an algorithm which takes as input a collection of k trees, in $O(kn^2)$ time decides if the collection is compatible, or returns a conflict of size $\leq 2k$ in case of incompatibility.

$\rightarrow$ yields an algorithm with the claimed running time, using *bounded search*.

A bounded-search algorithm for SMAST

Preliminary definitions:

The collection $\mathcal{T}$

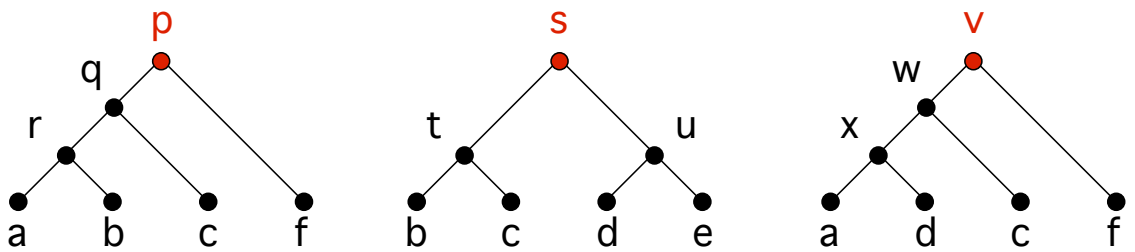


A *position in* $\mathcal{T}$ is a tuple $\pi = (\pi[1], ..., \pi[k])$, where each $\pi[i]$ is $\perp$ or a node of T_i. The *root position* is $\pi_\top = (r(T_1), ..., r(T_k))$.

A bounded-search algorithm for SMAST

Preliminary definitions:

The collection $\mathcal{T}$



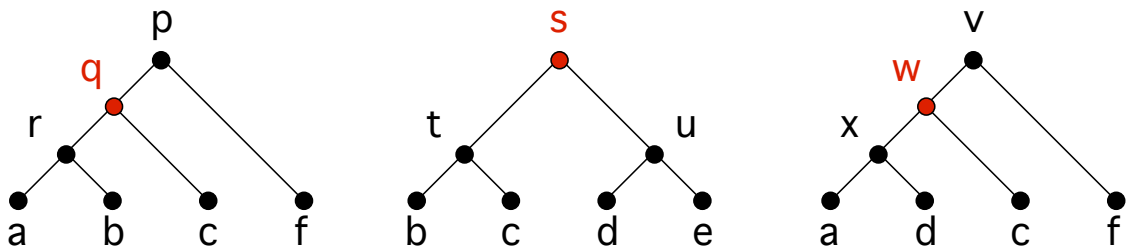
A *position in* $\mathcal{T}$ is a tuple $\pi = (\pi[1], ..., \pi[k])$, where each $\pi[i]$ is $\perp$ or a node of T_i. The *root position* is $\pi_\top = (r(T_1), ..., r(T_k))$.

Example: $\pi_\top = (p, s, v)$,

A bounded-search algorithm for SMAST

Preliminary definitions:

The collection $\mathcal{T}$



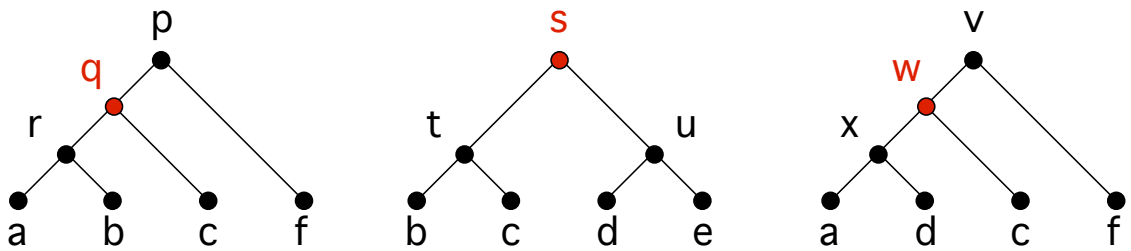
A *position in* $\mathcal{T}$ is a tuple $\pi = (\pi[1], ..., \pi[k])$, where each $\pi[i]$ is $\perp$ or a node of T_i. The *root position* is $\pi_\top = (r(T_1), ..., r(T_k))$.

Example: $\pi_\top = (p, s, v)$, $\pi = (q, s, w)$.

A bounded-search algorithm for SMAST

Preliminary definitions:

The collection $\mathcal{T}$

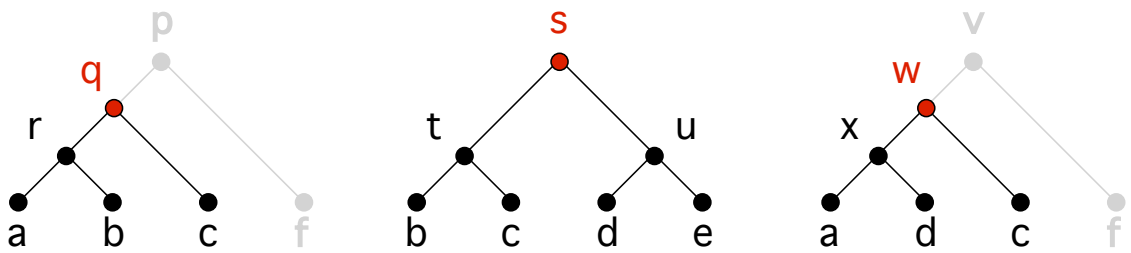


For each i, let T'_i be the subtree of T_i rooted at $\pi[i]$ (or the empty tree if $\pi[i] = \bot$). We define: $\mathcal{T}(\pi) := \{T'_1, ..., T'_k\}$.

A bounded-search algorithm for SMAST

Preliminary definitions:

The collection $\mathcal{T}$



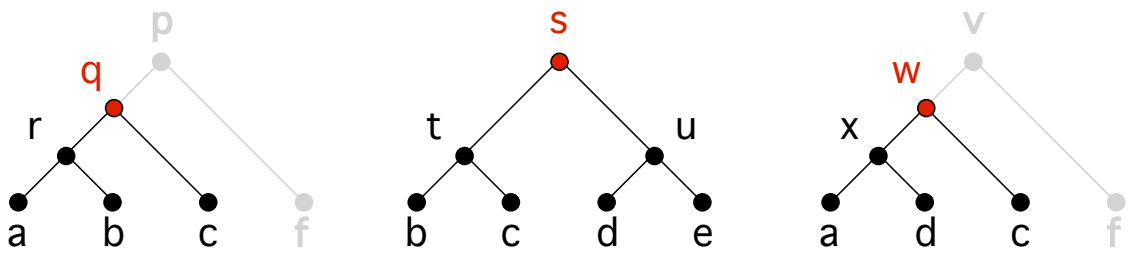
For each i, let T'_i be the subtree of T_i rooted at $\pi[i]$ (or the empty tree if $\pi[i] = \bot$). We define: $\mathcal{T}(\pi) := \{T'_1, ..., T'_k\}$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We say that π is *compatible* iff the collection $\mathcal{T}(\pi)$ is compatible.

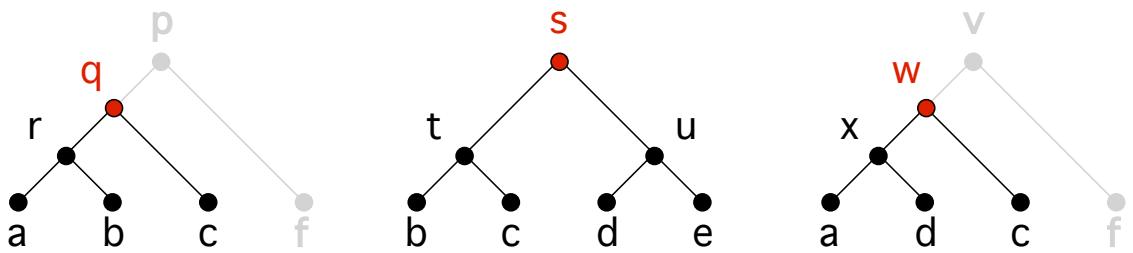
Idea:

- a recursive algorithm which takes a position π, decides if π is compatible, or returns a conflict;
- running it from $\pi_\top$ solves the compatibility problem for $\mathcal{T}$.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

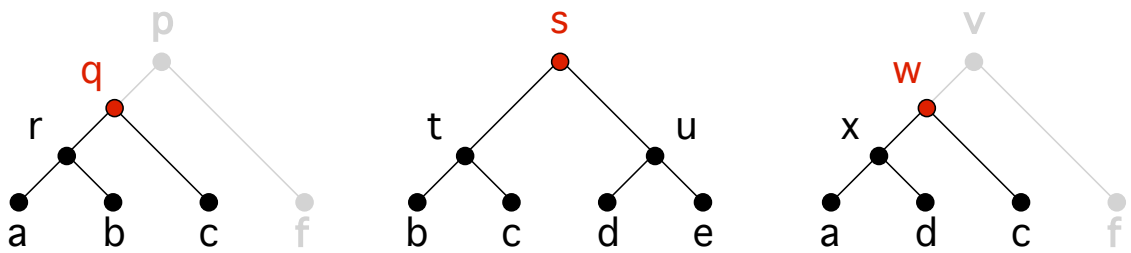
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

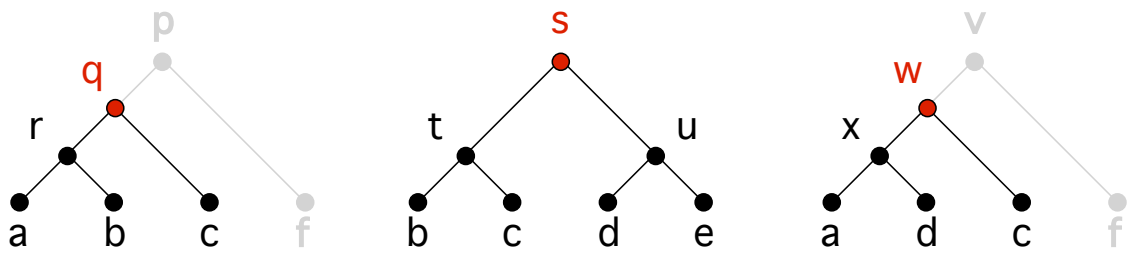
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

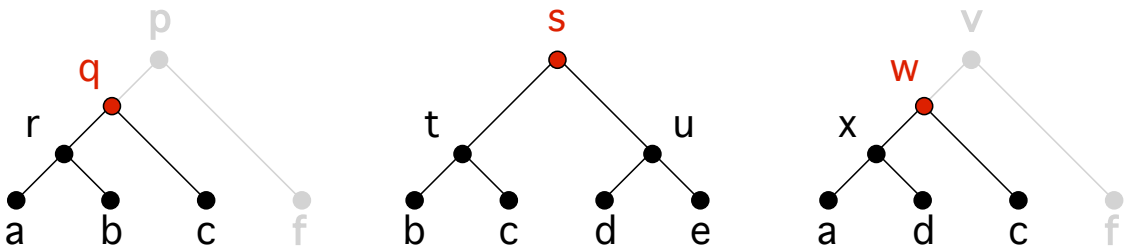
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

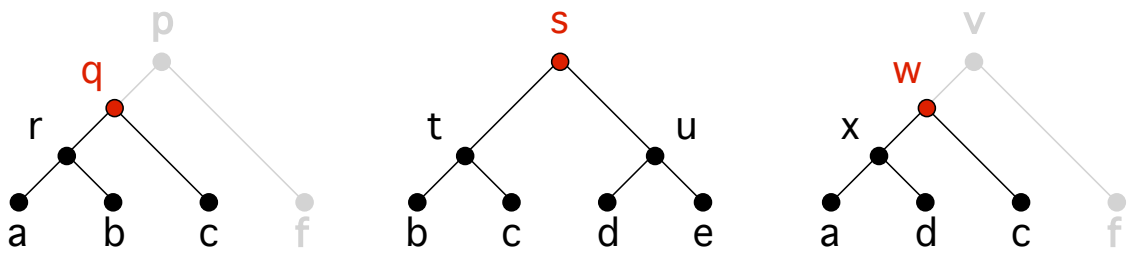
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

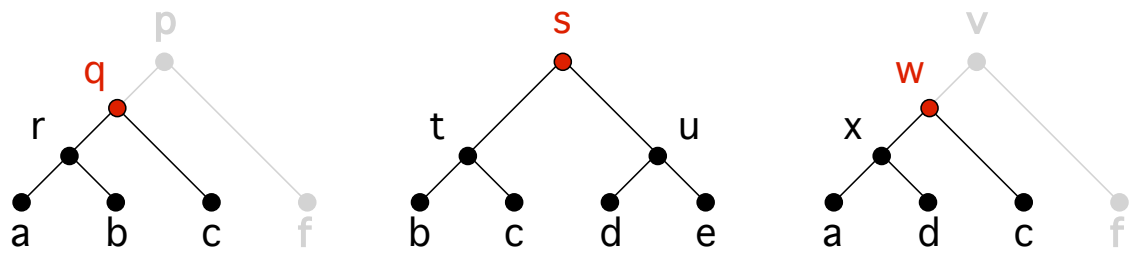
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

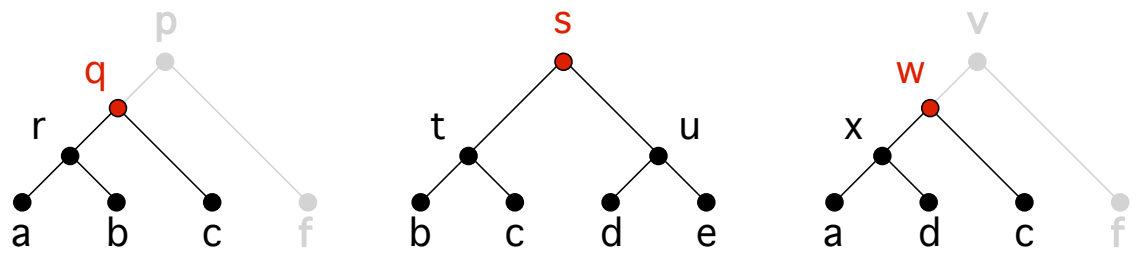
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

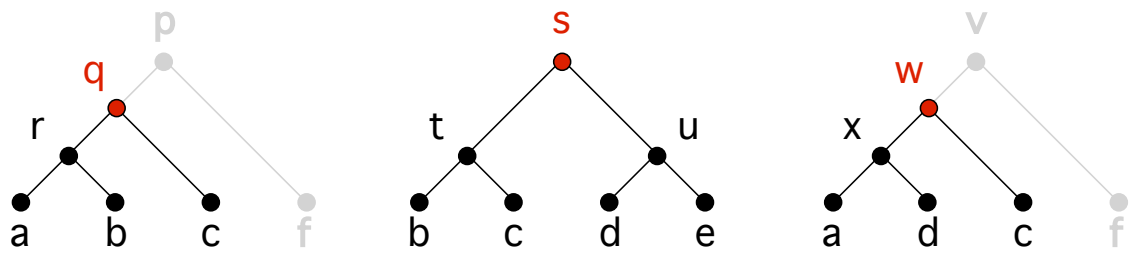
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

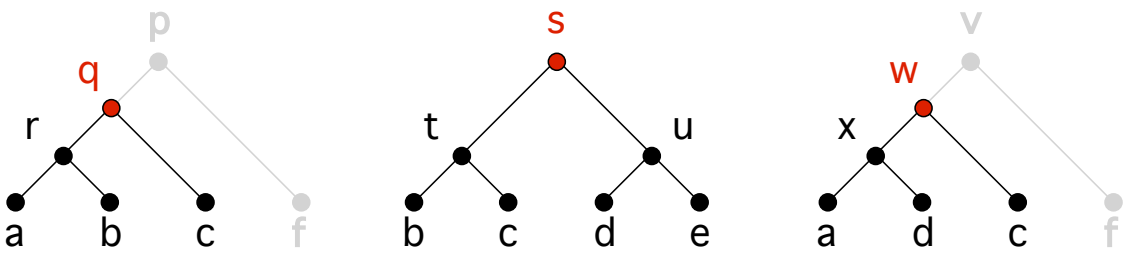
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

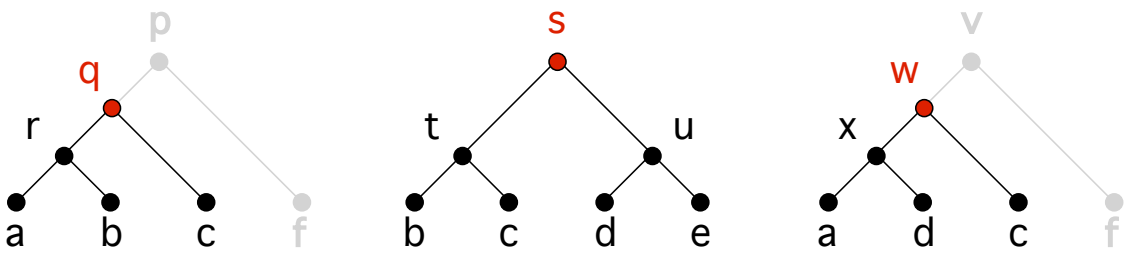
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

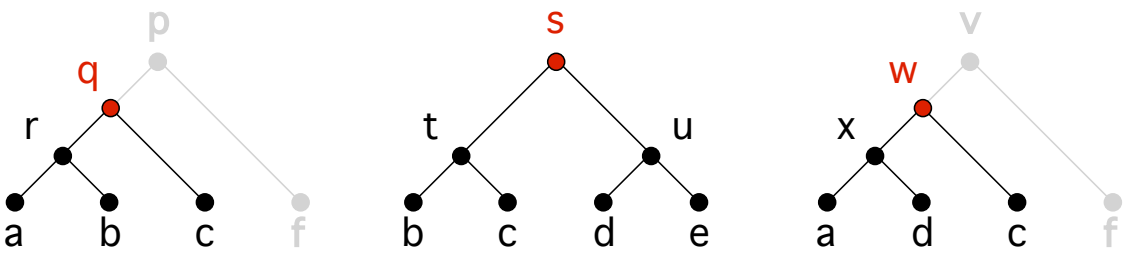
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

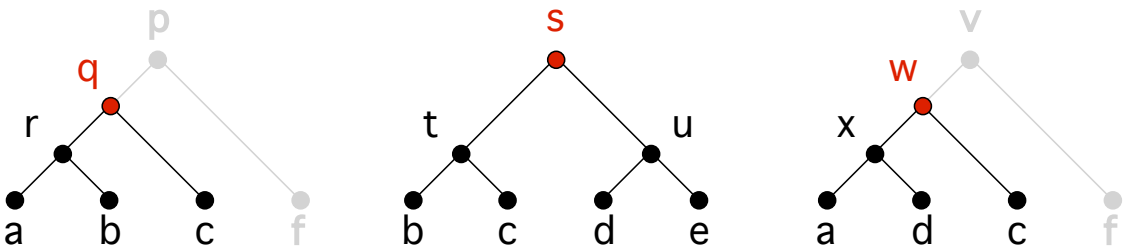
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

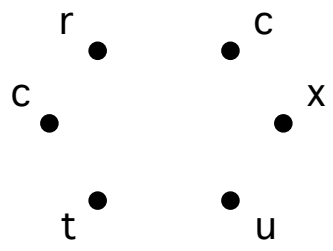
A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



The graph $G(\mathcal{T}, \pi)$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

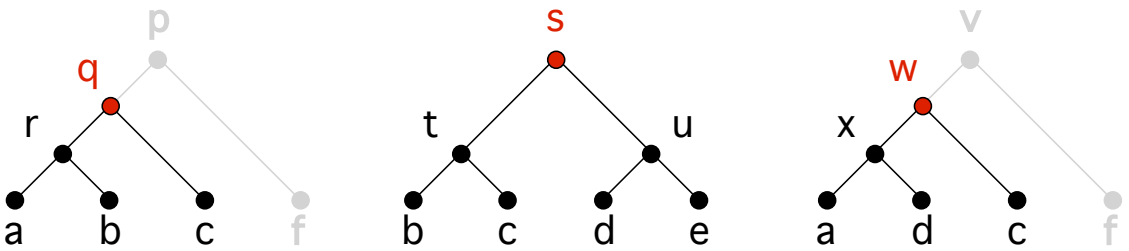
- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

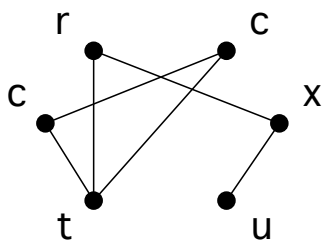
A bounded-search algorithm for SMAST

Principle of the algorithm:

The collection $\mathcal{T}$



The graph $G(\mathcal{T}, \pi)$



We define the graph $G(\mathcal{T}, \pi)$. Consider the indices $i \in [k]$ s.t. $\pi[i]$ has two children v_i, v_i'.

- the graph has vertex set $\{v_i, v_i' : i\}$;
- it contains an edge $\{x, y\}$ whenever $L(x) \cap L(y) \neq \emptyset$.

Example.

A bounded-search algorithm for SMAST

Principle of the algorithm:

There are two cases, depending on the connectnedness of $G = G(\mathcal{T}, \pi)$.

First case: G is not connected. Consider a partition of its vertex set in two disconnected sets $V_!, V_2$. Then from V_1, V_2 we can define two *successor positions* π_1, π_2 s.t. π is compatible $\Leftrightarrow$ π_1, π_2 are compatible.

Second case: G is connected. Let T be a spanning tree of G. For each edge $\{u, v\}$ choose a label in $L(u) \cap L(v)$. The resulting set is a conflict of size $\leq 2k$.

A bounded-search algorithm for SMAST

The algorithm:

Procedure IsCompatible(π)

let π' be obtained from π by replacing each leaf component by $\bot$; **if** $\pi' = (\bot, ..., \bot)$ **then**
    return true;
else
    perform a connectivity test on $G := G(\mathcal{T}, \pi')$ (*);
    if *G is not connected* **then**
        we obtain a partition of V in disconnected sets V_1, V_2;
        let π_1, π_2 be the successor positions of π' induced by V_1, V_2;
        call IsCompatible(π_1), IsCompatible(π_2);
    else
        we obtain a spanning tree $T = (V, F)$ of G;
        for each $e = \{u, v\} \in F$, choose $l_e \in L(u) \cap L(v)$;
        return false, together with the conflict $\{l_e : e \in F\}$;

Step (*) can be done in $O(kn)$ time by working on the intersection model of G. Starting from $\pi_\top$, the total running time is therefore $O(kn^2)$.

A dynamic programming algorithm for SMAST

Thm: SMAST can be solved in $O((8n)^k)$ time and $O((2n)^k)$ space.

Idea: we define a value $\#SMAST(\pi)$ for each position π in $\mathcal{T}$, s.t.

- the values $\#SMAST(\pi)$ satisfy recurrence relations allowing their computation by dynamic programming;
- $\#SMAST(\mathcal{T})$ is obtained as $\#SMAST(\pi_\top)$.

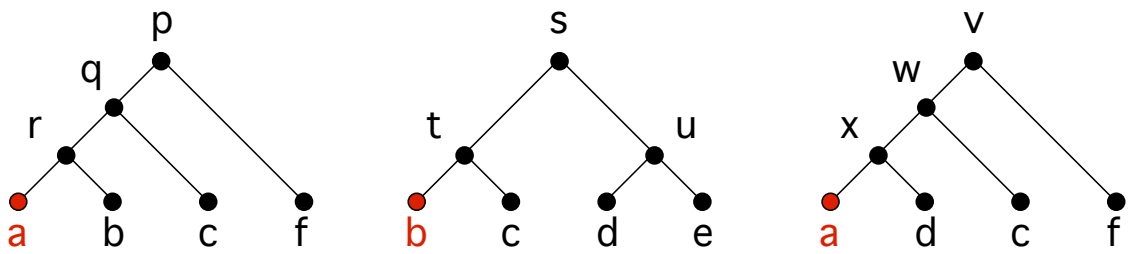
Complexity:

- space: there are $O((2n)^k)$ positions, hence the result;
- time: there are $O((2n)^k)$ positions, and each position in processed in $O(4^k)$ time, hence the time complexity is $O((8n)^k)$.

A dynamic programming algorithm for SMAST

Base case: for each $i \in [k]$, $\pi[i]$ is either $\perp$ or a leaf of T_i.

The collection $\mathcal{T}$



Let $L(\pi)$ be the set of labels associated to the $\pi[i] \neq \perp$.

A label $l \in L(\pi)$ is *maximally present* iff for each $i \in [k]$, if l appears in T_i then $\pi[i] = l$.

Example: $L(\pi) = \{a, b\}$. a is maximally present, but b is not.

A dynamic programming algorithm for SMAST

Base case: for each $i \in [k]$, $\pi[i]$ is either $\perp$ or a leaf of T_i.

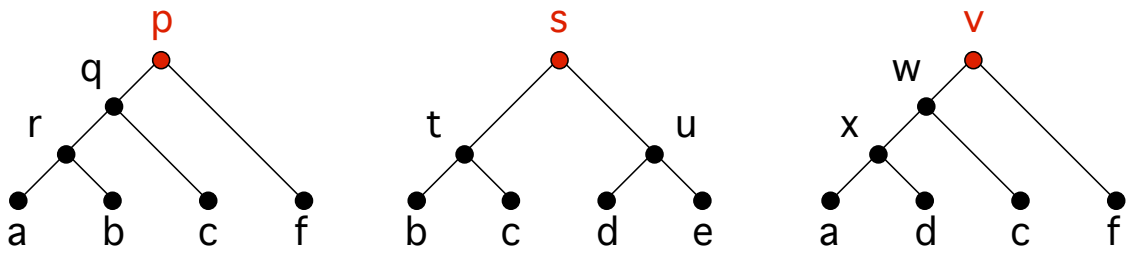
Then $\#SMAST(\pi)$ is the number of maximally present elements of $L(\pi)$.

This computation is performed in $O(k)$ time.

A dynamic programming algorithm for SMAST

General case: there exists $i \in [k]$ s.t. $\pi[i]$ is an internal node of T_i.

The collection $\mathcal{T}$



Let π, π' be positions in $\mathcal{T}$. π' is a *successor* of π iff $\exists i \in [k]$ s.t.

(i) $\pi'[i]$ is a child of $\pi[i]$ in T_i;

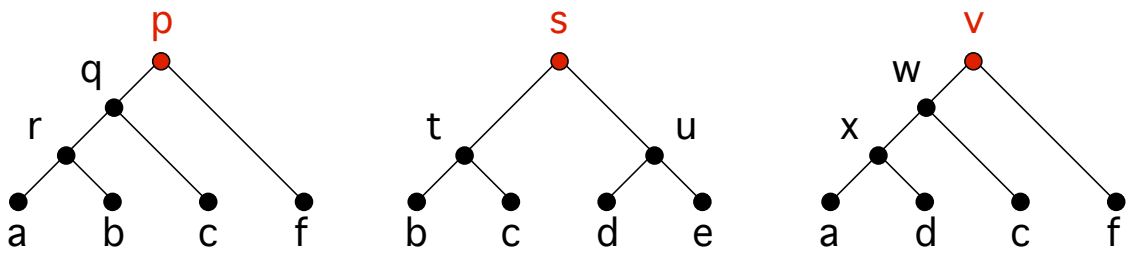
(ii) $\pi'[j] = \pi[j]$ for each $j \neq i$.

Example: $\pi = (p, s, v) \rightarrow \pi' = (p, s, w)$ is a successor of π.

A dynamic programming algorithm for SMAST

General case: there exists $i \in [k]$ s.t. $\pi[i]$ is an internal node of T_i.

The collection $\mathcal{T}$



Let π, π', π'' be positions in $\mathcal{T}$. (π', π'') is a *decomposition* of π iff (a) π', π'' are distinct from π, and (b) $\forall i \in [k]$

(i) if $\pi[i] = \bot$, then $\{\pi'[i], \pi''[i]\} = \{\bot\}$;

(ii) if $\pi[i]$ is a leaf l, then $\{\pi'[i], \pi''[i]\} = \{\bot, l\}$;

(iii) if $\pi[i]$ is an internal node u with two children v, v', then $\{\pi'[i], \pi''[i]\}$ is $\{v, v'\}$ or $\{\bot, u\}$.

Example: $\pi = (p, s, v) \rightarrow \pi' = (p, t, f), \pi'' = (\bot, u, w)$ is a decomposition of π.

A dynamic programming algorithm for SMAST

General case: there exists $i \in [k]$ s.t. $\pi[i]$ is an internal node of T_i.

Then $\#SMAST(\pi)$ is computed as follows:

$$\begin{aligned}\#SMAST(\pi) = \max(&\\ &\max\{\#SMAST(\pi') : \pi' \text{ successor of } \pi\}\\ &\max\{\#SMAST(\pi') + \#SMAST(\pi'') :\\ &\quad (\pi', \pi'') \text{ decomposition of } \pi\})\end{aligned}$$

Observations:

- the values $\#SMAST(\pi')$ involved in the computation satisfy $\pi' <_{\mathcal{T}} \pi$;
- there are $O(k)$ successors and $O(4^k)$ decompositions, hence $\#SMAST(\pi)$ is computed in $O(4^k)$ time.

Concluding remarks

Experiments?

Approximation ratio of the minimization problem?

- as a function of n: lower bound $= \Omega(\log n)$, upper bound $= n$;
- as a function of k: lower bound $= \Omega(\log k)$, upper bound $= 2k$.

End of the talk

Thank you.