Comparison of trees
Different trees for same set of species
Robinson-Foulds distance
Any edge $e$ in a tree $T$ corresponds to a split $U|V$ of the species into two disjoint sets. An **internal edge** corresponds to a **non-trivial split**.

**Non-trivial splits in $T_1$:**
- ADF | BCEG
- DF | ABCEG
- BC | ADEFG
- EG | ABCDF

**Non-trivial splits in $T_2$:**
- ADF | BCEG
- AD | BCEFG
- BC | ADEFG
Split-dist($T_1, T_2$) := “number of splits in $T_1$ not in $T_2$”

RF-dist($T_1, T_2$) := Split-dist($T_1, T_2$) + Split-dist($T_2, T_1$)

:= “number of splits in $T_1$ not in $T_2$” + “number of splits in $T_2$ not in $T_1$”

:= “number of splits not found in both trees”

Any edge $e$ in a tree $T$ corresponds to a split $U|V$ of the species into two disjoint sets.

An internal edge corresponds to a non-trivial split.

Non-trivial splits in $T_1$:  
ADF | BCEG  
DF | ABCEG  
BC | ADEFG  
EG | ABCDF

Non-trivial splits in $T_2$:  
ADF | BCEG  
AD | BCEFG  
BC | ADEFG
Split-dist\((T_1, T_2)\) := “number of splits in \(T_1\) not in \(T_2\)”

RF-dist\((T_1, T_2)\) := Split-dist\((T_1, T_2)\) + Split-dist\((T_2, T_1)\)

\(= \) “number of splits in \(T_1\) not in \(T_2\)” + “number of splits in \(T_2\) not in \(T_1\)”

\(= \) “number of splits not found in both trees”

Any edge \(e\) in a tree \(T\) corresponds to a split \(U|V\) of the species into two disjoint sets

An internal edge corresponds to a non-trivial split

Non-trivial splits in \(T_1\):
- ADF | BCEG
- DF | ABCEG
- BC | ADEFG
- EG | ABCDF

Non-trivial splits in \(T_2\):
- ADF | BCEG
- AD | BCEFG
- BC | ADEFG

If we can find “shared splits” then we can compute RF-distance
Split-dist($T_1, T_2$) := “number of splits in $T_1$ not in $T_2$”

RF-dist($T_1, T_2$) := Split-dist($T_1, T_2$) + Split-dist($T_2, T_1$)

:= “number of splits in $T_1$ not in $T_2$” + “number of splits in $T_2$ not in $T_1$”

:= “number of splits not found in both trees”

Any edge $e$ in a tree $T$ corresponds to a split $U | V$ of the species into two disjoint sets

An internal edge corresponds to a non-trivial split

Split-dist($T_1, T_2$) := 2

Split-dist($T_2, T_1$) := 1

RF-dist($T_1, T_2$) := 3

If we can find “shared splits” then we can compute RF-distance
Split-dist\( (T_1, T_2) \) := “number of splits in \( T_1 \) not in \( T_2 \)”

RF-dist\( (T_1, T_2) \) := Split-dist\( (T_1, T_2) \) + Split-dist\( (T_2, T_1) \)

:= “number of splits in \( T_1 \) not in \( T_2 \)” + “number of splits in \( T_2 \) not in \( T_1 \)”

:= “number of splits not found in both trees”

Note: Split-dist is symmetric if the two trees are binary. Why?

Any edge \( e \) in a tree \( T \) corresponds to a split \( U \mid V \) of the species into two disjoint sets.

An internal edge corresponds to a non-trivial split.

\begin{align*}
\text{Split-dist}(T_1, T_2) & \quad := \quad 2 \\
\text{Split-dist}(T_2, T_1) & \quad := \quad 1 \\
\text{RF-dist}(T_1, T_2) & \quad := \quad 3
\end{align*}

Non-trivial splits in \( T_1 \):

ADF \mid BCEG
DF \mid ABCEG
BC \mid ADEFG
EG \mid ABCDF

Non-trivial splits in \( T_2 \):

ADF \mid BCEG
AD \mid BCEFG
BC \mid ADEFG

If we can find “shared splits” then we can compute RF-distance.
Computing the RF-distance

**Input:** Two unrooted trees, $T_1$ and $T_2$, with $n$ identical leaves

**Algorithm:**

```
shared = 0
for each split $U \mid V$ in $T_1$ and $U' \mid V'$ in $T_2$ do
    if $U \mid V == U' \mid V'$ then
        shared = shared + 1
    endif
end
return RF-Dist = “number of splits in $T_1$ and $T_2” - 2 * shared
```
Computing the RF-distance

**Input:** Two unrooted trees, $T_1$ and $T_2$, with $n$ identical leaves

**Algorithm:**

1. $\text{shared} = 0$
2. **for** each split $U \mid V$ in $T_1$ and $U' \mid V'$ in $T_2$ **do**
   1. **if** $U \mid V == U' \mid V'$ **then**
      1. $\text{shared} = \text{shared} + 1$
   2. **endif**
3. **end**

**return** $\text{RF-Dist} = \text{“number of splits in } T_1 \text{ and } T_2 \text{” - 2 * shared}$

Running time: $O(n^2 \times \text{“time it takes to decide } U \mid V == U' \mid V' \text{”})$

By using time and space $O(n^2)$ for preprocessing $U \mid V == U' \mid V'$ can be decided in time $O(1)$
Details about deciding $U | V = U' | V'$
Deciding if $U \mid V = U' \mid V'$

We consider every edge $e$ in $T$ and $e'$ in $T'$ as two directed edges...
Deciding if $U \mid V == U' \mid V'$

Assume that we have a table $T$ s.t.:

$$T(\rightarrow, \rightarrow') := |V \cap V'|$$

... then $U \cap V \neq U' \cap V'$ can be decided in time $O(t)$ ...

$$u \cap V = u' \cap V' \iff |V \cap V'| = |V|$$

(else $|V \cap V'| + 1 = u' \cap V' = u$)
Deciding if $U | V == U' | V'$

If we assume that $1, 2, 1', 2'$ are non-empty, i.e. $e$ and $e'$ are internal edges in $T$ and $T'$, we get:

$$V \cap V' = (1 \cup 2) \cap (1' \cup 2')$$

$$= (1 \cup 2) \cap 1' \cup (1 \cup 2) \cap 2'$$

$$= (1 \cap 1') \cup (2 \cap 1') \cup (1 \cap 2') \cup (2 \cap 2')$$
Deciding if $U \mid V == U' \mid V'$

Since $1 \cap 2 = 1' \cap 2' = \emptyset$, we get:

$$|V \cap V'| = |1 \cap 1'| + |1 \cap 2'| + |2 \cap 1'| + |2 \cap 2'|$$

... a simple recursive formulation; the table $T[e, e']$ of size $O(n^2)$ can be computed in time: $O(n^2)$ using dynamic programming....
Deciding if $U \mid V = U' \mid V'$

**Special case:**

\[ T[e, e'] = 1 \vee n_{e'} = T[e, e_2'] + T[e, e_2'] \]

**Base case:**

\[ e \rightarrow v \quad \quad e' \rightarrow v' \]

\[ T[e, e'] = \begin{cases} 1 & \text{if } v=v' \\ 0 & \text{otherwise} \end{cases} \]
Other ideas? Splits as bit-vectors

Any edge $e$ in a tree $T$ corresponds to a split $U|V$ of the species into two disjoint sets.

An **internal edge** corresponds to a **non-trivial split**

<table>
<thead>
<tr>
<th>Non-trivial splits in $T_1$:</th>
<th>Non-trivial splits in $T_2$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFD</td>
<td>BCEG</td>
</tr>
<tr>
<td>DF</td>
<td>ABCEG</td>
</tr>
<tr>
<td>BC</td>
<td>ADEFG</td>
</tr>
</tbody>
</table>
Other ideas? Splits as bit-vectors

Algorithm for “counting number of shared splits”

Step 1: Collect bit-vectors

Step 2: Sort bit-vectors using radix-sort

0000101
0001010
0110000
0110000
1001000
1001010
1001010

Step 3: Count “number of shared splits” as “number of doublets”

Time and space?

Non-trivial splits in $T_1$:

- AFD | BCEG
- DF | ABCEG
- BC | ADEFG
- EG | ABCDF

Non-trivial splits in $T_2$:

- ADF | BCEG
- AD | BCEFG
- BC | ADEFG

Any edge in a tree $T$ corresponds to a split $U \mid V$ of the species into two disjoint sets.
Other ideas? Splits as bit-vectors

Algorithm for “counting number of shared splits”

Step 1: Collect bit-vectors. $O(n \times \text{“size of bit vector”})$

Step 2: Sort bit-vectors using radix-sort $O(n \times \text{“size of bit vector”})$

Step 3: Count “number of shared splits” as “number of doublets”

Time and space? $O(n \times \text{“size of bit vector”})$

Non-trivial splits in $T_1$:  
AFD | BCEG 1001010  
DF | ABCEG 0001010  
BC | ADEFG 0110000  
EG | ABCDF 0000101  

Non-trivial splits in $T_2$:  
ADF | BCEG 1001010  
AD | BCEFG 1001000  
BC | ADEFG 0110000  
EG | ABCDF 0000101
Other ideas? Splits as bit-vectors

Algorithm for “counting number of shared splits”

<table>
<thead>
<tr>
<th>Any edge in a tree T corresponds to a split U</th>
<th>V of the species into two disjoint sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-trivial splits in T 1:</td>
<td></td>
</tr>
<tr>
<td>AFD</td>
<td>BCEG 1001010</td>
</tr>
<tr>
<td>DF</td>
<td>ABCEG 0001010</td>
</tr>
<tr>
<td>BC</td>
<td>ADEFG 0110000</td>
</tr>
<tr>
<td>EG</td>
<td>ABCDF 0000101</td>
</tr>
<tr>
<td>Non-trivial splits in T 2:</td>
<td></td>
</tr>
<tr>
<td>AFD</td>
<td>BCEG 1001010</td>
</tr>
<tr>
<td>AD</td>
<td>BCEFG 1001000</td>
</tr>
<tr>
<td>BC</td>
<td>ADEFG 0110000</td>
</tr>
<tr>
<td>EG</td>
<td>ABCDF 0000101</td>
</tr>
</tbody>
</table>

Algorithm for “counting number of shared splits”

Step 1: Collect bit-vectors. $O(n \times \text{“size of bit vector”})$

Step 2: Sort bit-vectors using radix-sort $O(n \times \text{“size of bit vector”})$

Step 3: Count “number of shared splits” as “number of doublets”

Time and space? $O(n \times \text{“size of bit vector”})$

The bit vector of $n$ bit can be store in $n / \log n \ “\text{machine words}”$
Computing the RF-distance

- Using table of “intersections sizes” in time $O(n^2)$

- Using bit vectors in time $O(n \times \text{“size of bit vector”})$

- Day's algorithm from 1985 computes the RF-distance in time $O(n)$ !!
Day's algorithm (my explanation)

**Step 1:** Root the two input trees at the same leaf (here leaf no. 1). Time $O(n)$. 

Input trees:

After step 1:
Day's algorithm (my explanation)

Step 2: Make a **Depth-First numbering** of the leaves in $T_1$. Time $O(n)$. 
Step 3: Rename the leaves in $T_2$ cf. the DF-numbering of leaves in $T_1$. Time $O(n)$. 
Day's algorithm (my explanation)

Step 4: 1) Annotate internal nodes in $T_1$ with their DF-intervals (the DF-number of the leaves in a subtree is an interval per construction. 2) For each internal node in $T_2$ find (min. DF-leaf, max. DF-leaf, size of subtree), if “max – min + 1 = size” then the subtree is an interval of DF-leaves. Annotate the internal node in $T_2$ with this interval. Time $O(n)$. 
Day's algorithm (my explanation)

**Step 5:** Note that every DF-interval in $T_1$ which is also in $T_2$ is a shared split. Sort the intervals and identify doublets. *Time $O(n)$ by simple radix-sort.*

**Step 4:** 1) Annotate internal nodes in $T_1$ with their DF-intervals (the DF-number of the leaves in a subtree is an interval per construction). 2) For each internal node in $T_2$ find (min. DF-leaf, max. DF-leaf, size of subtree), if “max – min + 1 = size” then the subtree is an interval of DF-leaves. Annotate the internal node in $T_2$ with this interval. *Time $O(n)$.*
Quartet distance
Quartets and quartet distance

**Quartet**: Four named species in an unrooted tree

**Quartet topology**: The topology of the quartet induced by the tree

**Quartet distance**: The number of quartets that don't have the same topology in the two trees
Quartets and quartet distance
Quartets and quartet distance

A
B
C
D

A
B
C
D

A
B
C
D

A
B
D
E

A
B
C

A
B
D
E

ABCD

x
Quartets and quartet distance
Quartets and quartet distance
Quartets and quartet distance
Quartets and quartet distance

ABCD
ABCE
ABDE
ACDE
BCDE
Quartets and quartet distance

Quartet distance $= \binom{5}{4} - 3 = 5 - 3 = 2$
Previous work


C. R. Doucette. An efficient algorithm to compute quartet dissimilarity measures. O(\(n^3\)) Bachelor of Science (Honours) Dissertation. Memorial University of Newfoundland, 1985


Details about the $O(n^2)$ algorithm
Computing the quartet distance

We associate quartet ab|cd in $T$ and $T'$ with the oriented edge $e$ s.t.:
Computing the quartet distance

Algorithm: For every pair of oriented edges \((e, e')\) in \(T\) and \(T'\), we count the number of quartets associated to \(e\) in \(T\), which is also associated to \(e'\) in \(T'\).

\[
\text{Count}(e, e') = |A_{NN} l \cdot |B_{NN} l \cdot \left(\binom{1 \cup c \cup c'}{2}\right) + |A_{NN} l \cdot |B_{NN} l \cdot \left(\binom{1 \cup c \cup c'}{2}\right)
\]
Computing the quartet distance

Observe: every shared quartet ab1cd, is counted twice....

1. \[ \text{Diagram 1} \]

2. \[ \text{Diagram 2} \]

so...
Computing the quartet distance

\[ \text{Quartet Dist:} = \binom{n}{4} - \frac{1}{2} \sum_{e,e'} \text{Count}(e,e') \]

Running time?

\( O(n^2) \)
Comparing partially resolved trees
Comparing partially resolved trees

The ideas of Tsang's $O(n^2)$ time algorithm for fully resolved tree can be generalized. An immediate generalization yields and $O(n^2d^4)$ time algorithm. Doucette's work from 1985 can be used for a $O(n^3)$ time algorithm ...
Idea: Iterate over all pairs of edges (or nodes) in the two trees, and count for each pair how many *associated quartets* are shared. The problem is to define “associated” such each quartet is counted as most once ...

### Different solutions

<table>
<thead>
<tr>
<th>Type</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center based</td>
<td>$O(n^3)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Edge based</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Edge based</td>
<td>$O(n+</td>
<td>V</td>
</tr>
<tr>
<td>Node based</td>
<td>$O(n+</td>
<td>V</td>
</tr>
</tbody>
</table>

- $n$ is the number of species/leaves
- $|V|, |V'|$ are number of internal nodes in $T$ and $T'$
- $id, id'$ are internal degree of $T$ and $T'$

"worst case"-tree: $|V|=id=O(n)$
A sub-cubic time algorithm for computing the quartet distance between two general trees.

Thomas Mailund, Jesper Nielsen and Christian N.S. Pedersen

In this paper we develop an $O(n^{2+})$, where $\gamma = (1/2)$ and $O(n^\gamma)$ is the time it takes to multiply two $n \times n$ matrices. Using the Coppersmith-Winograd algorithm, where $\gamma = 2.376$, this yields a running time of $O(n^{2.688})$. The running time is thus independent of the degrees of the inner nodes of the input trees, and this is the first sub-cubic time algorithm with this property.

<table>
<thead>
<tr>
<th>Type</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center based</td>
<td>$O(n^3)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Edge based</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Edge based</td>
<td>$O(n+</td>
<td>V</td>
</tr>
<tr>
<td>Node based</td>
<td>$O(n+</td>
<td>V</td>
</tr>
</tbody>
</table>

- $n$ is the number of species/leaves
- $|V|$, $|V'|$ are number of internal nodes in $T$ and $T'$
- $\text{id}$, $\text{id}'$ are internal degree of $T$ and $T'$

"worst case"-tree

$|V|=\text{id}=O(n)$
A sub-cubic time algorithm for computing the quartet distance between two general trees (2011).

Thomas Mailund, Jesper Nielsen and Christian N.S. Pedersen

In this paper we develop an $O(n^{2+\epsilon})$, where $\epsilon = (1/2)$ and $O(n^2)$ is the time it takes to multiply two $n \times n$ matrices. Using the Coppersmith-Winograd algorithm, where $\epsilon = 2.376$, this yields a running time of $O(n^{2.688})$. The running time is then independent of the degrees of the inner nodes of the input trees.

Efficient Algorithms for Computing the Triplet and Quartet Distance Between Trees of Arbitrary Degree (2012).

Gerth Brodal, Rolf Fagerberg, Thomas Mailund, Christian N.S. Pedersen and Andreas Sand

In this paper we develop an $O(n \log n)$ time algorithm for computing the triplet distance between trees of arbitrary degree, and a $O(dn \log n)$ time algorithm for computing the quartet distance between trees of degree $d$.

- $n$ is the number of species/leaves
- $|V|, |V'|$ are number of internal nodes in $T$ and $T'$
- $\text{id}, \text{id}'$ are internal degree of $T$ and $T'$
Other kinds of distances?
Subtree-Prune-Regraft

\[ \text{SPR-Dist}(T_1, T_2) = \text{Minimum number of SPR-op's that transform } T_1 \text{ into } T_2 \]

NP-complete for unrooted binary trees allowing unrestricted SPR-operations ...
Nearest-Neighbor interchange

\[
\text{NNI-Dist}(T_1, T_2) = \text{Minimum number of NNI-op's that transform } T_1 \text{ into } T_2
\]

NP-complete for unrooted binary trees ...