Presentation
Martin Randers
Outline

• Introduction
• Algorithms
• Implementation and experiments
• Memory consumption
• Summary
Introduction
Introduction

• Evolution of species can be modelled in trees
• Trees consist of nodes and edges
  - nodes divided into *leaves* and *internal nodes*
  - edges connecting internal nodes are called *internal edges*
The Quartet Distance

- A quartet is a set of four leaves
- Can have four possible quartet topologies

Butterfly topologies

Star topology

- The quartet distance between a pair of trees is the number of quartets with different topologies
Other Algorithms

- Previously: Binary trees - algorithms created by Tsang ($O(n^2)$) & Brodal et al. ($O(n \log(n))$)
- Now: Trees of arbitrary degrees
  - Generalization of Tsang's algorithm uses time $O(|V|d^2|V'|d'^2)$
  - Generalization of Brodal et al.'s algorithm to trees of bounded degree uses time $O(d^9n\log(n))$
Algorithms
Algorithms

• Ways to compute shared leaf set sizes are presented in thesis
• Five algorithms for computing the quartet distance designed and implemented:
  - Two center based
  - Two based on edge claiming
  - One based on node claiming
Center Based Algorithms

- Observation: Each triplet of leaves has a unique internal node as *center*
Center Based Algorithms

• Observation: Each triplet of leaves has a unique internal node as *center*

```
\[ a \rightarrow c \rightarrow b \]
```

The center of \(a, b\) and \(c\)
Center Based Algorithms

• Observation: Each triplet of leaves has a unique internal node as center

• The topologies of quartets containing the triplet easily determined from this center
Center Based Algorithms

- Observation: Each triplet of leaves has a unique internal node as *center*

- The topologies of quartets containing the triplet easily determined from this center
Center Based Algorithms

• Observation: Each triplet of leaves has a unique internal node as center

The center of a, b and c

• The topologies of quartets containing the triplet easily determined from this center

The topologies
First Center Based Algorithm

- $O(n^4)$ algorithm:
  - For each triplet of leaves:
    - Find center of the triplet, in each tree
    - Determine topology of quartets, in each tree
   - Compare topology of quartets

- Uses space $O(n)$ to store topologies
Second Center Based Algorithm

- $O(n^3)$ algorithm:
  - Precompute shared leaf set sizes
  - For each pair of leaves:
    - Find center of each triplet containing the pair, in each tree
    - Use precomputed sizes to calculate number of shared quartets

- Uses space $O(n^2)$ to store shared leaf set sizes
Handling Star Quartets

- Quartets can be divided into five categories:

  \[ Q_{SS}(T, T'), Q_{SB}(T, T'), Q_{BS}(T, T'), Q_{B=B}(T, T'), Q_{B\neq B}(T, T') \]
Handling Star Quartets

• Quartets can be divided into five categories:

\[ Q_{SS}(T, T'), Q_{SB}(T, T'), Q_{BS}(T, T'), Q_{B=B}(T, T'), Q_{B\neq B}(T, T') \]

\[ qdist(T, T') \]
• Quartets can be divided into five categories:

\[ Q_{SS}(T, T'), Q_{SB}(T, T'), Q_{BS}(T, T'), Q_{B=B}(T, T'), Q_{B\neq B}(T, T') \]

\[ qdist(T, T') \]

• Observe that

\[ Q_{B=B}(T, T) = Q_{BS}(T, T') + Q_{B=B}(T, T') + Q_{B\neq B}(T, T') \]

\[ Q_{B=B}(T', T') = Q_{SB}(T, T') + Q_{B=B}(T, T') + Q_{B\neq B}(T, T') \]
Handling Star Quartets

• Quartets can be divided into five categories:

\[ Q_{SS}(T, T'), Q_{SB}(T, T'), Q_{BS}(T, T'), \]
\[ Q_{B=B}(T, T'), Q_{B\neq B}(T, T') \]

\[ q_{dist}(T, T') \]

• Observe that

\[ Q_{B=B}(T, T) = Q_{BS}(T, T') + Q_{B=B}(T, T') + Q_{B\neq B}(T, T') \]
\[ Q_{B=B}(T', T') = Q_{SB}(T, T') + Q_{B=B}(T, T') + Q_{B\neq B}(T, T') \]

• Yielding

\[ q_{dist}(T, T') = Q_{B=B}(T, T) + Q_{B=B}(T', T') - 2Q_{B=B}(T, T') - Q_{B\neq B}(T, T') \]
Claiming

- Associating quartets to specific edges or nodes in the trees
- Calculate shared/nonshared quartets by processing pairs of edges/nodes
- Important that each quartet is claimed by a fixed number of edges/nodes
Edge Claiming

- Each directed internal edge $e$ claims all butterfly quartets where two leaves are located in two different subtrees in front of $e$, and the other two are in the subtree behind $e$
Each directed internal edge $e$ claims all butterfly quartets where two leaves are located in two different subtrees in front of $e$, and the other two are in the subtree behind $e$.
Edge Claiming

- Each directed internal edge $e$ claims all butterfly quartets where two leaves are located in two different subtrees in front of $e$, and the other two are in the subtree behind $e$.

- Each butterfly quartet is claimed by exactly two directed edges.

```
  a--------x------c
 /          e
b--------+--------d
```

$e$ claims $ab|cx$ and $ab|dx$, but not $ab|cd$. 

- Each butterfly quartet is claimed by exactly two directed edges.
Encoding

• The quartets claimed by each edge can be encoded as a number of triplets of subtrees (*claims*).
• The number of shared and nonshared quartets in a pair of claims can be computed in time $O(1)$.
• If $e$ points to a node of degree $d$, it takes $\binom{d-1}{2}$ claims to encode the quartets claimed.
• Processing all pairs of claims takes time

$$\sum_{v \in V} \sum_{v' \in V'} \left( id_v \left( \frac{d_v - 1}{2} \right) id_{v'} \left( \frac{d_{v'} - 1}{2} \right) \right) = O(|V||V'|d^2d'^2)$$
Expansion

• First edge claiming algorithm:
  - Expand the trees to binary trees, and define *extended claims*
  - Compute shared leaf set sizes
  - Process all pairs of extended claims

• If $e$ points to a node of degree $d$, it takes $d-1$ extended claims to encode the quartets claimed

• Processing all pairs of extended claims thus takes time $\sum_{v \in V} \sum_{v' \in V'} \left( id_v d_v id_{v'} d_{v'} \right) = O(|V||V'|dd')$

• Uses space $O(n^2)$ to store shared leaf set sizes
Counting

• Instead of encoding, count number of shared/nonshared quartets claimed by both edges directly.
• Done in three steps for a pair of edges:
  i. Count too many (some “illegal”)
  ii. Deduct all the “illegal” ones (some twice)
  iii. Add the ones that were deducted twice
• If a pair of edges points to nodes of internal degrees $id$ and $id'$ respectively this can be done in time $O(idid')$
Counting (cont)

- Second edge claiming algorithm:
  - Compute shared leaf set sizes of non-leaf subtrees
  - Do the counting described above for each pair of edges
- Doing precomputations and processing all pairs of edges takes time
  \[ O(n + |V||V'| + \sum_{v \in V} \sum_{v' \in V'} (id_v^2 id_{v'}^2)) = O(n + |V||V'| idid') \]
- Uses space \( O(n + |V||V'|) \) to store shared leaf set sizes and sizes of subtrees
Node Claiming

- Motivation: A lot of redundant computations are done in the second edge claiming algorithm
- Each internal node \( v \) claims all butterfly quartets claimed by directed internal edges *pointing to* \( v \)
- Each butterfly quartet is claimed by exactly two directed edges
Shared Quartets

• Given a pair of nodes $v$ and $v'$ a number of sums can be precomputed in time $O(id_v id_{v'})$ and space $O(id_v + id_{v'})$

• These enables the number of shared quartets claimed by both nodes to be computed in time $O(id_v id_{v'})$
Nonshared Quartets

• By precomputing an additional $O\left(\min\{id_v, id_{v'}\}^2\right)$ sums in time $O(id_v id_{v'}, \min\{id_v, id_{v'}\})$ it is also possible to compute the number of nonshared quartets claimed by the nodes in time $O(id_v id_{v'}, \min\{id_v, id_{v'}\})$
The Algorithm

- Node claiming algorithm:
  - Compute shared leaf set sizes of non-leaf subtrees
  - For each pair of nodes:
    - Do the precomputing described above
    - Compute shared and nonshared quartets
- This takes time $O(n + |V||V'|\min\{id, id'\})$
- ... and uses space $O(n + |V||V'|)$
Implementation & Experiments
Implementation

- All of the presented algorithms have been implemented in Java
- Tool, with optional GUI, available for download
- Experiments has been performed using these implementations
Experiments

• The correctness of algorithms validated by comparing results
• The algorithms were run on four classes of trees to investigate actual running times:
  - worst case
  - d-ary
  - random
  - r8s based
Experiments

- The correctness of algorithms validated by comparing results
- The algorithms were run on four classes of trees to investigate actual running times:
  - worst case
  - d-ary
  - random
  - r8s based
Example of Results

**Time usage for the $O(n+|V||V'|\min\{id,id'\})$ algorithm on d-ary trees**

- $d=3$
- $d=6$
- $d=15$
- $d=30$
- $d=60$
- $d=90$
- $c_4n$
- $c_5n^2$
- $c_6n^3$
Comparing the Algorithms

Time usage for the edge claiming and node claiming algorithms on random topology trees and r8s–based trees

- Node claiming algorithm (r8s)
- Node claiming algorithm (random)
- Edge claiming algorithm (r8s)
- Edge claiming algorithm (random)
- Edge claiming algorithm using expansion (r8s)
- Edge claiming algorithm using expansion (random)
More Comparison

Time usage for the edge claiming and node claiming algorithms on worst case trees

- Node claiming algorithm
- Edge claiming algorithm
- Edge claiming algorithm using expansion

Time in milliseconds

Number of leaves
Analysis of Results

• The asymptotic performance was as expected
• The second edge claiming and the node claiming are clearly faster than the others.
• The node claiming algorithm was chosen for the tool
Reducing Memory Consumption
Reducing Memory Consumption

- The work has been focused at reducing running time, not memory consumption.
- Reduction in memory from $O(n^2)$ to $O(n + |V||V'|)$ was just a byproduct of optimizing running time.
- Can further reduction be done, maybe at the cost of adding running time?
Important Facts

• When processing a pair of nodes it is necessary and sufficient to know the shared leaf set sizes of all pairs of non-leaf subtrees of the nodes, and the sizes of individual subtrees

• Implies that a trivial lower bound on memory consumption is $O(n+id id')$

• The challenge:
Important Facts

• When processing a pair of nodes it is necessary and sufficient to know the shared leaf set sizes of all pairs of non-leaf subtrees of the nodes, and the sizes of individual subtrees

• Implies that a trivial lower bound on memory consumption is $O(n+\text{id id'})$

• The challenge: *The shared leaf set sizes of subtrees are calculated using other shared leaf set sizes, i.e. they are not independent*
Basic Idea

• The idea is inspired by the O(n log(n)) algorithm:
Basic Idea

• The idea is inspired by the $O(n \log(n))$ algorithm:
  - Do a number of colorings of the leaves in one tree
Basic Idea

• The idea is inspired by the $O(n \log(n))$ algorithm:
  - Do a number of colorings of the leaves in one tree
  - Build some structure based on the other tree and update it when colors change
• Let $v$ be an internal node in a tree $T$. $T$ is said to be colored according to $v$ if the leaves of each non-leaf subtree of $v$ are colored with the colors $1, \ldots, id_v$, one for each subtree, and all leaves directly connected to $v$ are colored with the color 0.
Let $v$ be an internal node in a tree $T$. $T$ is said to be colored according to $v$ if the leaves of each non-leaf subtree of $v$ are colored with the colors $1, \ldots, id_v$, one for each subtree, and all leaves directly connected to $v$ are colored with the color 0.
• Let \( v \) be an internal node in a tree \( T \). \( T \) is said to be colored according to \( v \) if the leaves of each non-leaf subtree of \( v \) are colored with the colors \( 1, \ldots, id_v \), one for each subtree, and all leaves directly connected to \( v \) are colored with the color 0.
• Root the other tree in an arbitrary internal node, and add an array to each internal node with $id_v$ entries
• Color the leaves in the rooted tree
• Initialize the entries in the arrays with the number of leaves of each color directly connected to the node
• Update the arrays depth first with the sum of the entries in the arrays of the children
Example
Example

root

add arrays
Example

root

add arrays

color leaves

0,1,2,3
Example
Example

- **root**
- **add arrays**
- **color leaves**
- **init arrays**
- **update arrays**

1. Initialize arrays:
   - $101$
   - $020$

2. Update arrays:
   - $222$
   - $020$

0, 1, 2, 3
Analysis

- Rooting can be done once and for all in constant time
- For each node \( v \) in \( T \):
  - Coloring can be done in time \( O(n) \)
  - Adding arrays takes time \( O(|V'|id_v) \)
  - Updating arrays takes time \( O \left( \sum_{v' \in V'} (id_{v'} id_v) \right) = O \left( |V'|id_v \right) \)
  - Space consumption is \( O(|V'|id_v) \)
Analysis (cont)

- Computation can be done as before, except the nodes must be handled in an ordered way.
- The time to handle all pairs of nodes is
  \[ O(|V||V'| \min \{id, id'\}) \]
- Total complexities are thus:
  \[ O(|V|n + |V||V'| \min \{id, id'\}) \] time, and
  \[ O(|V'|id) \] space
- Note that the roles of the trees can be switched.
Summary
Overview

• Presented here:
  - The problem
  - Existing algorithms
  - A way to avoid star quartets
  - The five algorithms designed and implemented
Overview

• Presented here:
  - The problem
  - Existing algorithms
  - A way to avoid star quartets
  - The five algorithms designed and implemented

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center based without shared leaf set sizes</td>
<td>$O(n^4)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Center based using shared leaf set sizes</td>
<td>$O(n^3)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Edge claiming using expansion</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Edge claiming without expansion</td>
<td>$O(n +</td>
<td>V</td>
</tr>
<tr>
<td>Node claiming</td>
<td>$O(n +</td>
<td>V</td>
</tr>
</tbody>
</table>
Overview

- Presented here:
  - The problem
  - Existing algorithms
  - A way to avoid star quartets
  - The five algorithms designed and implemented
  - Some of the experiments performed
  - A way to reduce the memory consumption of the fastest algorithm (at performance cost)
Additional Work in Thesis

- Ways to compute shared leaf set sizes
- Generalization of existing measures
- Handling non-standard input trees
- Visualization
- Presentation of tool
- Comparison with split distance
The End