Similarity Search

Token-Based Tree Distances

Nikolaus Augsten
nikolaus.augsten@plus.ac.at
Department of Computer Science
University of Salzburg

database research group
https://dbresearch.uni-salzburg.at

WS 2022/23

Version January 17, 2023

Outline

Tokens for Trees – Intuition

- q-Grams for strings:
  - split string into substrings (q-grams) of length q
  - strings with many common substrings are similar
- Tokens for trees:
  - split tree into small subunits (tokens) of the same shape
  - tokens may be individual nodes, subtrees, or subgraphs
  - trees with many common tokens are similar
- Example: the so-called pq-gram tokens are besom-shaped subtrees with $p+q$ nodes

$$\begin{array}{ccc} & v_1 & \\ & | & \\ & \cdots & \\ & | & \\ & v_p & \\ \swarrow & & \searrow \\ v_{p+1} & \cdots & v_{p+q} \end{array}$$

Token Profile and Label Tuples

- Token profile $P(T)$: set of all tokens of tree T
  - a token may be a subtree or a subgraph of the tree
  - the token profile $P(T)$ of a tree T is the set of all its tokens
- A linear encoding of a token traverses all its nodes in preorder:

$$\begin{array}{ccc} & v_1 & \\ & | & \\ & \cdots & \\ & | & \\ & v_p & \\ \swarrow & & \searrow \\ v_{p+1} & \cdots & v_{p+q} \end{array} = (v_1, \ldots, v_p, v_{p+1}, \ldots, v_{p+q})$$

- Label tuple $\lambda(t)$: tuple of the nodes labels $\lambda(\mathrm{v}_i)$ of token $t = (\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_k)$ in preorder:

$$\lambda(t) = (\lambda(\mathrm{v}_1), \lambda(\mathrm{v}_2), \ldots, \lambda(\mathrm{v}_k))$$

Token Index

Definition (Token Index)

Let $P(T)$ be a token profile of tree T. The token index, $\mathcal{I}$, of tree T is the **bag of all label tuples** of T,

$$\mathcal{I}(\mathrm{T}) = \biguplus_{g \in \mathrm{P_T}} \lambda(g)$$

- Note:
  - tokens consist of nodes and are unique within a tree
  - but: different tokens may yield identical label tuples
  - thus the token index may contain duplicates

Token-Based Distance

Definition (Token-Based Distance)

The token-based distance between two trees, T and T′, with token indexes $\mathcal{I}(\mathrm{T})$ and $\mathcal{I}(T')$, respectively, is defined as

$$\delta(\mathrm{T}, \mathrm{T}') = |\mathcal{I}(\mathrm{T}) \uplus \mathcal{I}(\mathrm{T}')| - 2|\mathcal{I}(\mathrm{T}) \Cap \mathcal{I}(\mathrm{T}')|$$

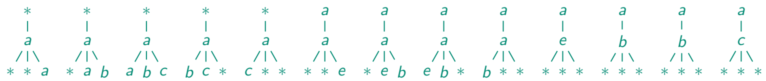
- Metric normalization to [0..1]: $\delta'_g(\mathrm{T}, \mathrm{T}') = \frac{\delta_g(\mathrm{T}, \mathrm{T}')}{|\mathcal{I}(\mathrm{T}) \uplus \mathcal{I}(\mathrm{T}')| - |\mathcal{I}(\mathrm{T}) \Cap \mathcal{I}(\mathrm{T}')|}$
- Pseudo-metric properties hold for normalization [ABG10]:
  - ✓ self-identity: $x = y \not\Leftarrow \Rightarrow \delta_g(x, y) = 0$
  - ✓ symmetry: $\delta_g(x, y) = \delta_g(y, x)$
  - ✓ triangle inequality: $\delta_g(x, z) \leq \delta_g(x, y) + \delta_g(y, z)$
- Different trees may have identical indexes.

Storing the Token Index Efficiently

- Problem: How to store node labels efficiently?
  - Long labels: large storage overhead
  - Varying label length: in a relational database, the inefficient VARCHAR type must be used instead of the efficient CHAR type
- Solution: Hashing
  - compute fingerprint hash for labels
  - store concatenation of the hashed labels
- Fingerprint hash function (e.g., Karp-Rabin [KR87]):
  - maps a string s to a hash value $h(s)$
  - $h(s)$ is of fixed length
  - $h(s)$ is unique with high probability (for two different strings $s_1 \neq s_2$, $h(s_1) \neq h(s_2)$ with high probability)

Overview: Token Index

- Token profile: (so-called pq-grams in the example, $p = 2, q = 3$)



- Hashing: map tokens to integers:

$\xrightarrow{\text{serialize}} (*, a, a, b, c) \xrightarrow{\text{(shorthand)}} *aabc \xrightarrow{\text{hash}} 03376$

label l	h(l)
*	0
a	3
b	7
c	6
e	4

Note: labels may be strings of arbitrary length!

- Token index: bag of hashed tokens

$\mathcal{I}(\mathrm{T}) = \{03003, 03037, 03376, 03760, 03600, 33004, 33047, 33470, 33700, 37000, 36000, 34000, 37000\}$

Intuition: similar trees have similar token indexes.

Binary Tree

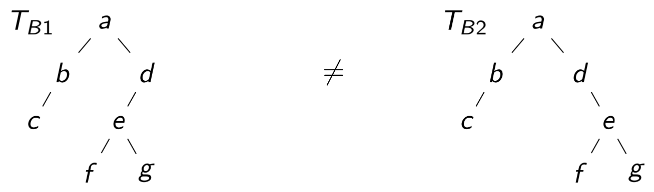
- In a binary tree
  - each node has at most two children;
  - left child and right child are distinguished:
    a node can have a right child without having a left child;
- Notation: $\mathrm{T}_B = (N, E_l, E_r)$
  - T_B denotes a binary tree
  - N are the nodes of the binary tree
  - E_l and E_r are the edges to the left and right children, respectively
- Full binary tree:
  - binary tree
  - each node has exactly zero or two children.

Example: Binary Tree

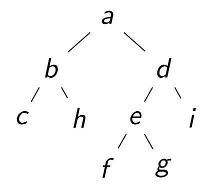
- Two different binary trees: $\mathrm{T}_B = (N, E_l, E_r)$

$\mathrm{T}_{B1} = (\{a, b, c, d, e, f, g\},\ \{(a, b), (b, c), (d, e), (e, f)\},\ \{(a, d), (e, g)\})$

$\mathrm{T}_{B2} = (\{a, b, c, d, e, f, g\},\ \{(a, b), (b, c), (e, f)\},\ \{(a, d), (d, e), (e, g)\})$



- A full binary tree:



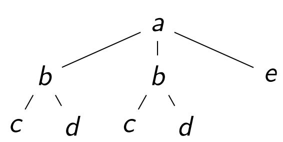
Binary Representation of a Tree

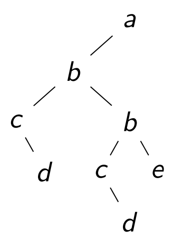
- Binary tree transformation:
  (i) link all neighboring siblings in a tree with edges
  (ii) delete all parent-child edges except the edge to the first child
- Transformation maintains
  - label information
  - structure information
- Original tree can be reconstructed from the binary tree:
  - a left edge represents a parent-child relationships in the original tree
  - a right edge represents a right-sibling relationship in the original tree

Example: Binary Tree Transformation

- Represent tree T as a binary tree:

T $\quad\rightarrow$ binary representation of T





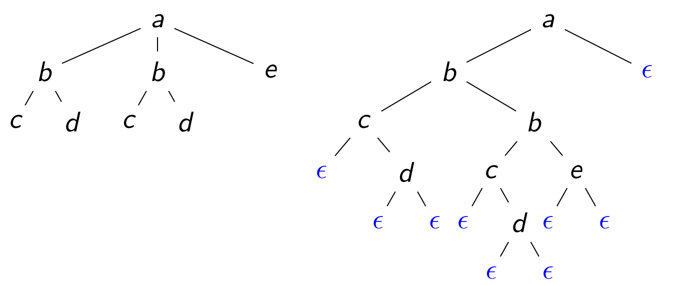
Normalized Binary Tree Representation

- We extend the binary tree with null nodes ϵ as follows:
  - a null node for each missing left child of a non-null node
  - a null node for each missing right child of a non-null node
- Note: Leaf nodes get two null-children.
- The resulting normalized binary representation
  - is a full binary tree
  - all non-null nodes have two children
  - all leaves are null nodes (and all null nodes are leaves)

Example: Normalized Binary Tree

- Transforming T to the normalized binary tree $B(T)$:

T $\rightarrow$ B(T)

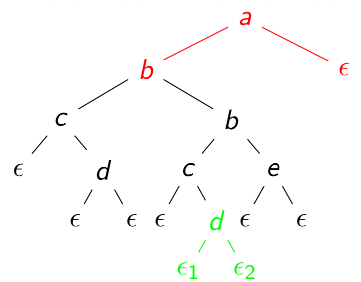


Binary Branch

- A binary branch $BiB(v)$ is
  - a subtree of the normalized binary tree $B(T)$
  - consisting of a non-null node v and its two children
- Example:

$BiB(a) = (\{a, b, \epsilon\}, \{(a, b)\}, \{(a, \epsilon)\})$

$BiB(d) = (\{d, \epsilon_1, \epsilon_2\}, \{(d, \epsilon_1)\}, \{(d, \epsilon_2)\})$ [1]



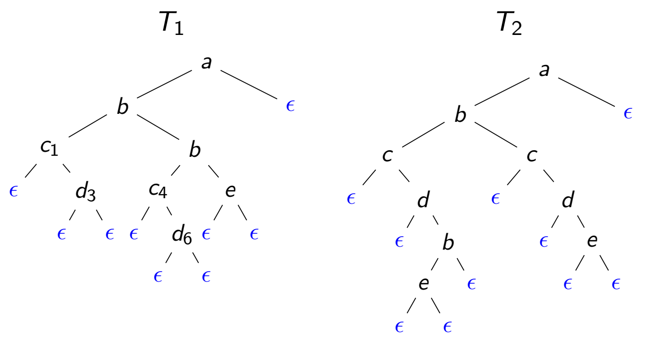
[1] Although the two null nodes have identical labels (ϵ), they are different nodes. We emphasize this by showing their IDs in subscript.

Binary Branches of Trees and Datasets

- Binary branches can be serialized as label tuples:
  - $BiB(v) = (\{v, a, b\}, \{(v, a)\}, \{(v, b)\}) \rightarrow \lambda(v) \circ \lambda(a) \circ \lambda(b)$
- Binary branch profile and index:
  - $P_{bb}(T)$ is the set of all binary branches of T
  - $\mathcal{I}_{bb}(T)$ is the multiset of all binary branch label tuples of T
- Note:
  - nodes are unique in the tree, thus binary branches are unique
  - labels are *not* unique, thus the label tuples are *not* unique
- Binary branch distance: The binary branch distance between two trees T_1 and T_2 is defined as:

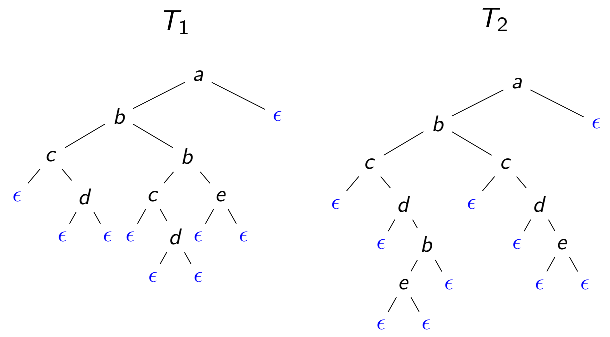
$$\delta_{bb}(T_1, T_2) = |\mathcal{I}_{bb}(T_1) \uplus \mathcal{I}_{bb}(T_2)| - 2|\mathcal{I}_{bb}(T_1) \Cap \mathcal{I}_{bb}(T_2)|$$

Example: Binary Branches and Label Tuples



- $BiB(c_1) \neq BiB(c_4)$:
  - $BiB(c_1) = (\{c_1, \epsilon_2, d_3\}, \{(c_1, \epsilon_2)\}, \{(c_1, d_3)\})$
  - $BiB(c_4) = (\{c_4, \epsilon_5, d_6\}, \{(c_4, \epsilon_5)\}, \{(c_4, d_6)\})$
- Serialization of both, $BiB(c_1)$ and $BiB(c_2)$, is identical: '$c\epsilon d$'

Example: Binary Branch Distance



$\mathcal{I}_{bb}(T_1) = \{abϵ, bcb, cϵd, dϵϵ, bce, cϵd, dϵϵ, eϵϵ\}$

$\mathcal{I}_{bb}(T_2) = \{abϵ, bcc, cϵd, dϵb, beϵ, eϵϵ, cϵd, dϵe, eϵϵ\}$

$\delta_{bb}(T_1, T_2) = 17 - 2 \cdot 4 = 9$

Lower Bound Theorem

Theorem (Lower Bound)

Let T_1 and T_2 be two trees. If the tree edit distance between T_1 and T_2 is $\delta_t(T_1, T_2)$, then the binary branch distance between them satisfies

$$\delta_{bb}(T_1, T_2) \leq 5 \times \delta_t(T_1, T_2).$$

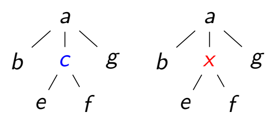
Proof (Sketch — Full Proof in [YKT05]).

- Each node v appears in at most two binary branches.
- *Rename*: Renaming a node causes at most two binary branches in each tree to mismatch. The sum is 4.
- Similar rational for *insert* and its complementary operation *delete* (at most 5 binary branches mismatch).

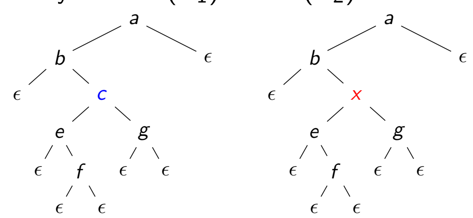
□

Proof Sketch: Illustration for Rename

- transform T_1 to T_2: $ren(c, x)$



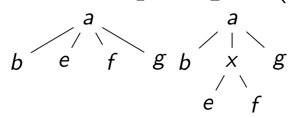
- binary trees $B(T_1)$ and $B(T_2)$



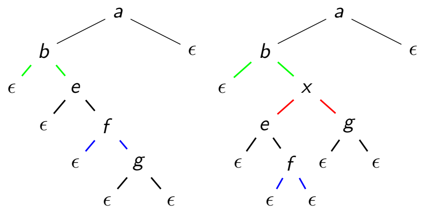
- Two binary branches ($b\epsilon c$, ceg) exist only in $B(T_1)$
- Two binary branches ($b\epsilon x$, xeg) exist only in $B(T_2)$
- $\delta_t(T_1, T_2) = 1$ (1 rename)
- $\delta_{bb}(T_1, T_2) = 4$ (4 binary branches different)

Proof Sketch: Illustration for Insert

- transform T_1 to T_2: $ins(x, a, 2, 2)$



- binary trees $B(\mathrm{T}_1)$ and $B(\mathrm{T}_2)$



- Two binary branches ($b\epsilon e$, $f\epsilon g$) exist only in $B(\mathrm{T}_1)$
- Tree binary branches ($b\epsilon x$, $f\epsilon\epsilon$, xeg) exist only in $B(\mathrm{T}_2)$
- $\delta_t(\mathrm{T}_1, \mathrm{T}_2) = 1$ (1 insertion)
- $\delta_{bb}(\mathrm{T}_1, \mathrm{T}_2) = 5$ (5 binary branches different)

Proof Sketch

- In general it can be shown that
  - Rename changes *at most* 4 binary branches
  - Insert changes *at most* 5 binary branches
  - Delete changes *at most* 5 binary branches
- Each edit operation changes at most 5 binary branches, thus

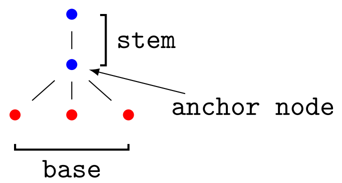
$$\delta_{bb}(\mathrm{T}_1, \mathrm{T}_2) \leq 5 \times \delta_t(\mathrm{T}_1, \mathrm{T}_2).$$

Complexity: Binary Branch Distance

- Generating the binary branches: $O(n)$ time and space ($n = \max\{|T_1|, |T_2|\}$)
  - the binary branches are formed in a single traversal of the tree
  - for each node of a tree a single binary branch is formed
- Computing the distance: $O(n \log n)$ time and $O(n)$ space
  - sort binary branch indexes to compute intersection: $O(n \log n)$
  - alternative: average case $O(n)$ runtime complexity
    1. build hash map for index $\mathcal{I}_{bb}(T_2)$
    2. probe label tuples of $\mathcal{I}_{bb}(T_1)$ to compute size of intersection

pq-Grams

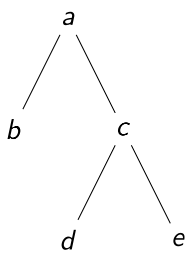
- The shape of a *pq*-gram (p=2, q=3):



- p nodes (anchor node and $p-1$ ancestors) form the stem
- q nodes (q consecutive children of the anchor node) form the base

pq-Extended Tree

- Problem: How can we split the following tree T into 2, 3-grams?

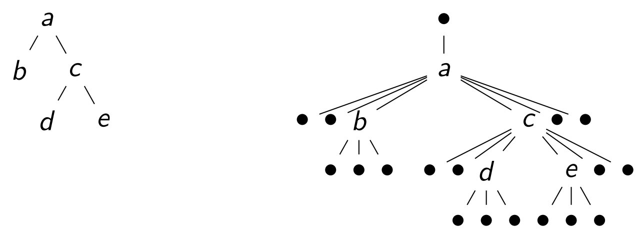


- Solution: Extend tree T with dummy nodes (•):
  - $p-1$ ancestors to the root node
  - $q-1$ children before the first and after the last child of each non-leaf
  - q children for each leaf
- The result is the *pq*-extened tree T^{pq}.

Example: Extended Tree

- An example tree T and its extended tree T^{pq} ($p=2$, $q=3$):

`T` `2, 3-extended tree` $T^{2,3}$



Definition: *pq*-Gram [ABG05]

Definition (*pq*-Gram)

Let T be a tree, $T^{p,q}$ the respective extended tree, $p > 0$, $q > 0$. A subtree of $T^{p,q}$ is a *pq-gram* g of T iff

(a) g has q leaf nodes and p non-leaf nodes,

(b) all leaf nodes of g are children of a single node $a \in N(g)$ with fanout q, called the anchor node,

(c) the leaf nodes of g are consecutive siblings in $T^{p,q}$.

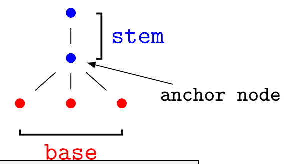
- Stem: anchor node and its ancestors in the *pq*-gram.
- Base: children of the anchor node in the *pq*-gram.

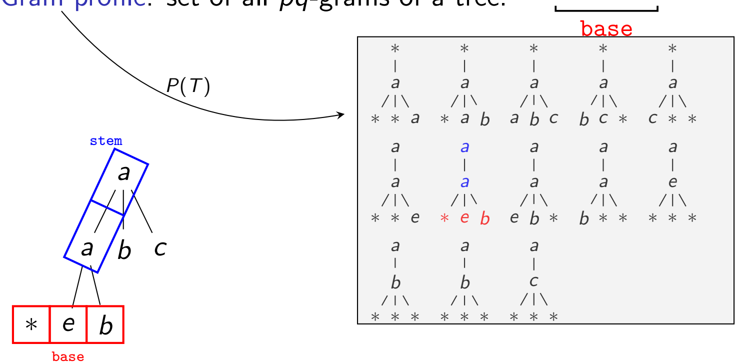
Definition (*pq*-Gram Profile)

The *pq*-gram profile, P_T, of a tree T is the set of all its *pq*-grams.

Example: Systematically Split Tree

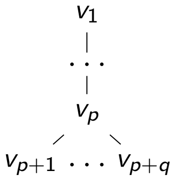
- *pq*-Gram: small subtree with stem and base
  Example: $p = 2$, $q = 3$
- Systematically split tree into *pq*-grams
- *pq*-Gram profile: set of all *pq*-grams of a tree.





Label Tuples

- Linear encoding of a *pq*-gram g with anchor node v_p:
  (traverse *pq*-gram in preorder)



$$= (v_1, \dots, v_p, v_{p+1}, \dots, v_{p+q})$$

- Label tuple: tuple of the *pq*-gram's node labels

$$\lambda(g) = (\lambda(v_1), \dots, \lambda(v_{p+q}))$$

for the *pq*-gram $g = (v_1, \dots, v_{p+q})$.

pq-Gram Index

Definition (*pq*-Gram Index)

Let T be a tree with profile P_T, $p > 0$, $q > 0$. The *pq*-gram index, $\mathcal{I}$, of tree T is the **bag of all label tuples** of T,

$$\mathcal{I}(T) = \biguplus_{g \in P_T} \lambda(g)$$

- Note:
  - *pq*-grams are unique within a tree
  - but: different *pq*-grams may yield identical label tuples
  - thus the *pq*-gram index may contain duplicates

Size of the *pq*-Gram Index

Theorem (Size of the *pq*-Gram Index)

Let T *be a tree of size $n = l + i$ with l leaves and i non-leaves. The size of the pq-gram index of* T *is linear in the tree size:*

$$|\mathcal{I}^{pq}(T)| = 2l + qi - 1 = O(n)$$

Proof.

1. We count all *pq*-grams whose leftmost leaf is a dummy node: Each leaf is the anchor node of exactly one *pq*-gram whose leftmost leaf is a dummy node, giving l *pq*-grams. Each non-leaf is the anchor of $q - 1$ *pq*-grams whose leftmost leaf is a dummy, giving $i(q-1)$ *pq*-grams.
2. We count all *pq*-grams whose leftmost leaf is not a dummy node: Each node of the tree except the root is the leftmost leaf of exactly one *pq*-gram, giving $l + i - 1$ *pq*-grams.

Overall number of *pq*-grams: $l + i(q-1) + (l + i - 1) = 2l + qi - 1$. □

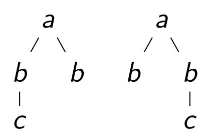
The *pq*-Gram Distance

Definition (*pq*-Gram Distance)

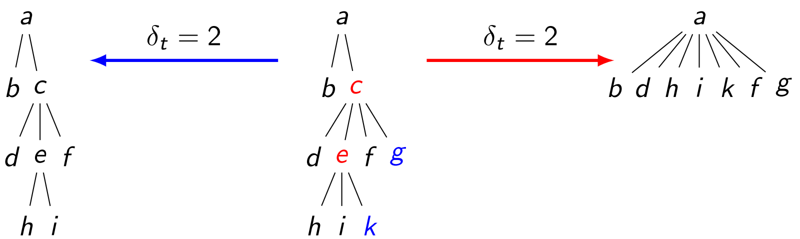
The *pq*-gram distance between two trees, T and T′, is defined as

$$\delta_g(T, T') = |\mathcal{I}(T) \uplus \mathcal{I}(T')| - 2|\mathcal{I}(T) \Cap \mathcal{I}(T')|$$

- Metric normalization to [0..1]: $\delta'_g(T, T') = \frac{\delta_g(T,T')}{|\mathcal{I}(T) \uplus \mathcal{I}(T')| - |\mathcal{I}(T) \Cap \mathcal{I}(T')|}$
- Pseudo-metric properties hold for normalization [ABG10]:
  - ✓ self-identity: $x = y \cancel{\Leftarrow}\Rightarrow \delta_g(x, y) = 0$
  - ✓ symmetry: $\delta_g(x, y) = \delta_g(y, x)$
  - ✓ triangle inequality: $\delta_g(x, z) \le \delta_g(x, y) + \delta_g(y, z)$
- Different trees may have identical indexes:



Motivation: Unit Cost Model Not Always Intuitive



- Unit cost edit distance:
  - no difference between leaves and non-leaves
  - may lead to non-intuitive results
- Conclusion: Non-leaves should have more weight than leaves.

Fanout Weighted Tree Edit Distance

Definition (Fanout Weighted Tree Edit Distance)

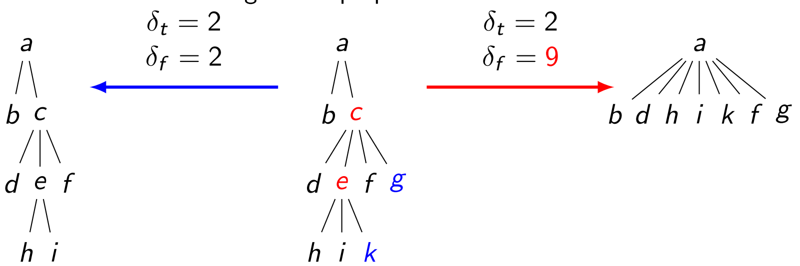
Let T and T′ be two trees, $w \in N(T)$ a node with fanout f, $w' \in N(T')$ a node with fanout f', $c > 0$ a constant. The fanout weighted tree edit distance, $\delta_f = (T, T')$, between T and T′ is defined as the tree edit distance with the following costs for the edit operations:

- Delete: $\alpha(w \to \epsilon) = f + c$
- Insert: $\alpha(\epsilon \to w') = f' + c$
- Rename: $\alpha(w \to w') = (f + f')/2 + c$

- Cost of changing a non-leaf node: proportional to its fanout.
- Cost of changing a leaf node: constant c.

Example: Fanout-Weighted Tree Edit Distance

- Fanout-Weighted Tree Edit Distance:
  - leaf changes have small cost ($c = 1$ in the example)
  - non-leaf changes cost proportional to the node fanout



pq-Gram Distance Lower Bound

Theorem

Let $p = 1$ and $c \geq \max(2q - 1, 2)$ be the cost of changing a leaf node. The pq-gram distance provides a lower bound for the fanout weighted tree edit distance, i.e., for any two trees, T *and* T′*,*

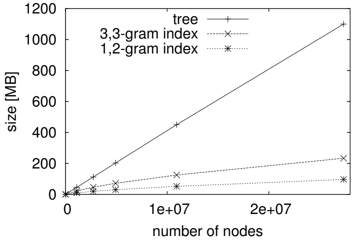
$$\frac{\delta_g(T, T')}{2} \leq \delta_f(T, T').$$

Proof.

See [ABG10] (ACM Transactions on Database Systems). □

Size of the *pq*-Gram Index

- *pq*-Gram index size: linear in the tree size
- Experiment:
  - compute *pq*-gram index for trees with different number of nodes
  - compare tree and index size



[Trees created with `xmlgen`.]

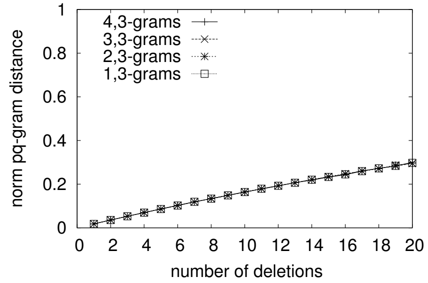
Why is the *pq*-gram index smaller than the tree?

- hash values are smaller than labels
- duplicate *pq*-grams of a tree are stored only once

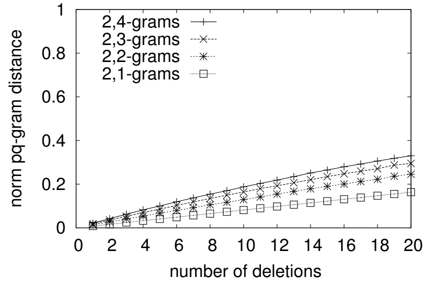
Sensitivity to Structure Change — Leaf

- Cost of leaf change → depends only on q
- Experiment:
  - delete leaf nodes
  - measure normalized *pq*-gram distance

vary p



vary q

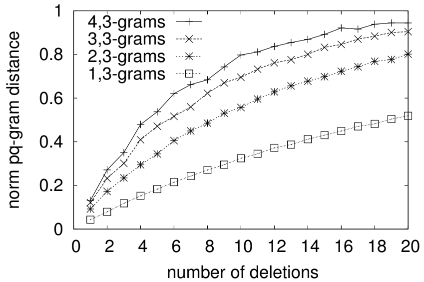


(Artificial tree with 144 nodes, 102 leaves, fanout 2–6 and depth 6. Average over 100 runs.)

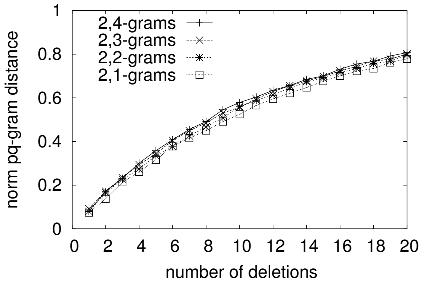
Sensitivity to Structure Change — Non-Leaf

- Cost for non-leaf change → controlled by p
- Experiment:
  - delete non-leaf nodes
  - measure normalized *pq*-gram distance

vary p



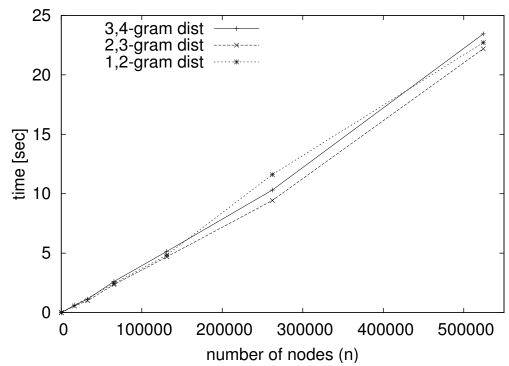
vary q



(Artificial tree with 144 nodes, 102 leaves, fanout 2–6 and depth 6. Average over 100 runs.)

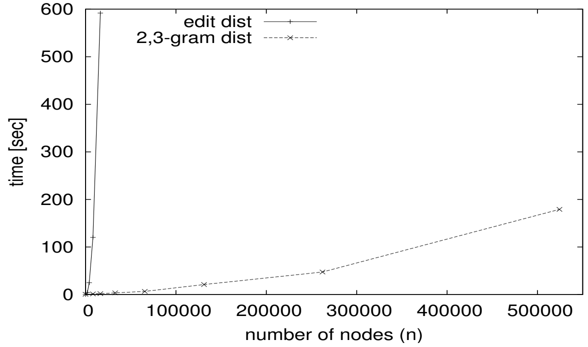
Influence of p and q on Scalability

- Scalability (almost) independent of p and q.
- Experiment: For pair of trees
  - compute *pq*-gram distance for varying p and q
  - vary tree size: up 10^6 nodes
  - measure wall clock time



Scalability to Large Trees

- *pq*-gram distance → scalable to large trees
- compare with edit distance
- Experiment: For pair of trees
  - compute tree edit distance and *pq*-gram distance
  - vary tree size: up 5×10^5 nodes
  - measure wall clock time



pq-Grams vs. other Edit Distance Approximations

Effectiveness: *pq*-grams outperform all other approximations

Experiment: two sets of address trees (299 and 302 trees)

- compute distances between all tree pairs
- find matches (symmetric nearest neighbor)

Distance	Correct	Recall	Precision	f-Measure	Runtime
fanout edit dist	259	86.6%	98.5%	0.922	19 min
unit edit dist	247	82.6%	96.5%	0.890	14 min
node intersection	197	65.9%	93.8%	0.774	4.3s
p,q-grams	236	78.9%	98.7%	0.877	8.1s
tree-embedding	206	68.9%	96.3%	0.803	7.1s
binary branch	193	64.5%	93.2%	0.763	7.4s
bottom-up	148	49.6%	92.5%	0.645	67.0s

Summary

- Binary Branch Distance
  - lower bound of the unit cost tree edit distance
  - trees are split into binary branches (small subgraphs)
  - similar trees have many common binary branches
  - complexity $O(n \log n)$ time and (n) space
- *pq*-Gram Distance
  - lower bound for the fanout weighted tree edit distance
  - trees are split into *pq*-grams (small subtrees)
  - similar trees have many common *pq*-grams
  - complexity $O(n \log n)$ time and $O(n)$ space

Nikolaus Augsten, Michael Böhlen, and Johann Gamper.
Approximate matching of hierarchical data using *pq*-grams.
In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 301–312, Trondheim, Norway, September 2005. ACM Press.

Nikolaus Augsten, Michael Böhlen, and Johann Gamper.
The *pq*-gram distance between ordered labeled trees.
ACM Transactions on Database Systems (TODS), 35(1):1–36, 2010.

Richard M. Karp and Michael O. Rabin.
Efficient randomized pattern-matching algorithms.
IBM Journal of Research and Development, 31(2):249–260, March 1987.

Rui Yang, Panos Kalnis, and Anthony K. H. Tung.
Similarity evaluation on tree-structured data.
In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 754–765, Baltimore, Maryland, USA, June 2005. ACM Press.