Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Efficient Enumeration of Non-isomorphic Processing Tree

F. Waas

Information Systems (INS)

INS-R9808 July 31, 1998

Report INS-R9808
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Efficient Enumeration of Non-isomorphic Processing Trees

Florian Waas

CWI
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
flw@cwi.nl

ABSTRACT

Enumeration of search spaces belonging to join queries, so far comprises large sets of isomorphic processing trees, i.e. trees that can be equalized by only commutative exchange of the input relations to the operators. Since these differences can be handled effectively by the costing method it is sufficient to focus on the enumeration of non-isomorphic processing trees only [2]. This restriction reduces the size of the search space by factor 2^k, k being the number of operators.

In this paper we present a technique that generates all non-isomorphic trees belonging to an arbitrarily shaped query graph. The algorithm generates sequences over the query graph's edges which are simultaneously turned into processing trees. During the incremental expansion of sequence prefixes the remaining edges are classified and only those not leading to isomorphic duplicates are appended. This incremental proceeding causes only little overhead leading to superior performance.

1991 Computing Reviews Classification System: [F.2.2] Nunnumerical Algorithms, [G.2.2] Graph algorithms, [H.2.4] Query Processing
Keywords and Phrases: non-isomorphic processing trees, enumeration algorithm, query optimization
Note: Funded by the HPCN/IMPACT project.

1. Introduction

Recently, the problem of enumeration of search spaces has seen an unexpected renaissance, not least due to some vendors' plans to deploy such methods in their products [6]. Close investigation of standard techniques revealed that for instance the enumeration of duplicates, or the discarding of intermediate results keeps these methods from meeting the theoretical complexity bounds (e.g. [7, 9]). Our own research is driven by enumeration experiments to gain deeper insight into cost modelling and the imposed cost distributions on search spaces. An area where enumeration of large spaces is an inevitable prerequisite.

In this paper we tackle the underlying, general question: *Can't we reduce the total number of plans that have to be explored by exploiting the topology of the processing trees?*

Our research was inspired by the cost model proposed by Ioannidis and Kang in [3]. A costing algorithm can anticipate the commutative exchange of the input relations to a join operator and choose the more cost efficient of the two alternatives on-the-fly. This decision is operator local and no larger context

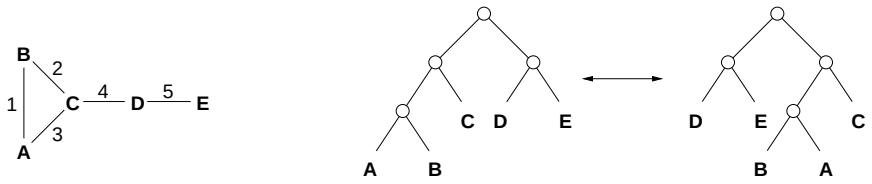

Figure 1: Query Graph with isomorphic processing trees.

than the immediate predecessors has to be taken into account. We call a pair of processing trees that can be equalized in such a way *isomorphic*, i.e. one can be turned into the other by commutatively exchange of sibling nodes. In Figure 1, two isomorphic trees together with the underlying query graph are depicted.

Conventional enumeration algorithms construct 2^k isomorphic alternatives of each single processing tree, where k is the number of joins. In other words, generating only non-isomorphic processing trees reduces the search space's size by factor 2^k.

In a different context, Galindo-Legaria et. al. developed counting, ranking, and un-ranking methods for non-isomorphic processing trees belonging to tree-shaped query graphs [1]. These techniques could be combined and after counting the trees each plan can be generated by unranking its ordinal number. However, for arbitrary query graphs, the problem appears to be much harder [2] and no such method is known for the general case, nor do the existing ones extend to it easily.

We present an approach based on sequences of the query graphs edges that overcomes these limitations. Every sequence corresponds to a processing tree as follows. For each edge of the sequence we add the respective join operator to the tree, or, if the edge's relations are already connected by a previous join, we only add the predicate to that join. However, different sequences do not necessarily correspond to different non-isomorphic trees: For the query graph given in Figure 1, the sequences $\langle 1, 5, 3, 2, 4\rangle$ and $\langle 5, 1, 3, 4, 2\rangle$ would yield isomorphic processing trees.

Therefore, we develop a ranking of sequences and classify redundancy both among and within sequences. This allows us to separate sets of edges that would cause isomorphic variants otherwise. Finally, we arrive at an algorithm that enumerates all sequences corresponding to non-isomorphic trees.

Not only does this technique extend the scope of enumeration methods it also provides the necessary efficiency: Its inherent incremental build-up of sequences together with an also incremental merge of processing trees needs only little reorganization effort due to the extensive re-use of intermediate results yet without considerable main-memory requirements.

2. Sequences

For a set $E = \{e_1, \ldots, e_n\}$ a *sequence* L *over* E is denoted by $L = \langle e_{\pi^{-1}(1)}, e_{\pi^{-1}(2)}, \ldots, e_{\pi^{-1}(n)}\rangle = \langle e_{\pi^{-1}(i)}\rangle$ where π is a permutation function on $\{1, \ldots, n\}$, i.e. element e_i is found at position $\pi(i)$ in the sequence. The set of

all sequences over E is denoted by E^* and contains $n!$ elements. If E is the empty set E^* contains only the *empty sequence* $\langle\rangle$.

For convenience, we introduce the following operations in analogy to sets. $|L|$ denotes the length of a sequence; $L \setminus E'$ is the sequence without the elements of E' but retaining the order of the residual elements. As an example, consider $\langle e_1, e_2, e_3\rangle \setminus \{e_2\}$ which is $\langle e_1, e_3\rangle$. Finally, we use :: to describe the concatenation of a sequence with either a single element or another sequence.

In order to facilitate the identification of sequences and to order E^* we introduce an enumeration function $\eta : E \to \{1, \ldots, |E|\}$ that assigns subsequent natural numbers to the elements. We define a ranking as follows:

Definition 2.1
For a sequence $L = \langle e_1, \ldots, e_n\rangle$ of length n, the *rank* of L to a base b with $b \geq n$ is

$$r_b = \sum_{i=1}^{n} \eta(e_i) \cdot b^{n-i} \tag{2.1}$$

The rank of the empty sequence is 0. ◇

For instance, the rank of $L = \langle e_1^{(3)}, e_2^{(1)}, e_3^{(4)}, e_4^{(2)}\rangle$ to the base $b = 10$—the figures in brackets show the values of $\eta(e_i)$—computes to $r_{10}(L) = 3 \cdot 10^3 + 1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0 = 3142$.

Since we demand $b \geq n$, every rank function is bijective and rank functions to different bases define the same order on E^*. For the remainder of this paper we assume $b \geq |E|$ and omit the notation of b. Finally, with $\widetilde{E}$ we denote the *η-sorted sequence* where elements with higher η value occur later in the sequence than elements with a lower one, i.e. $\eta(e_i) < \eta(e_j) \Rightarrow \pi(i) < \pi(j)$. For completeness, we define the η-sorted sequence of the empty set to be the empty sequence.

3. Query Graphs and Processing Trees

A query graph $G(V, E)$ is a connected graph that consists of nodes V, the set of base relations involved in the query, and edges E, that indicate which relations are linked by predicates. To describe a subgraph of G determined by a certain set of edges we shall write $G(E')$ as an abbreviation for $G(V', E')$ with $V' = \{v \in V | (v, w) \in E', w \in V\}$. Analogously, we use $G(V')$ to describe a subgraph containing the nodes of V' and all edges of E between nodes of V', i.e. $G(V', E')$ with $E' = \{(v, w) \in E | v, w \in V'\}$.

Definition 3.1
Let $G(V, E)$ be a query graph. A *processing tree over G* is a binary tree with $|V|$ leaves. There exists a mapping $\phi : W \to 2^V$ between the nodes W of the tree and the powerset of V with:

a) $\phi(w_{root}) = V$

b) $\phi(w)$ is a non-empty, connected subgraph of G

c) for the sons w_1, w_2 of every inner node w, $\phi(w_1) \cap \phi(w_2) = \emptyset$ and $\phi(w_1) \cup \phi(w_2) = \phi(w)$ hold ◇

The following definition captures the equivalence, contained in the previous one, in terms of the tree structure.

DEFINITION 3.2
Let t_1 and t_2 be processing trees over G. t_1 and t_2 are *equivalent*, short $t_1 \cong t_2$, iff there exists an isomorphism $h : W_1 \to W_2$ between the nodes of t_1 and those of t_2 such that:

$$w_j \text{ is son of } w_i \text{ in } t_1 \Rightarrow h(w_j) \text{ is son of } h(w_i) \text{ in } t_2 \tag{3.1}$$

And in case w is leaf, w and $h(w)$ identify the same base relation. ◇

Note, that in contrast to the notion of *isomorphic binary trees* where only the shape of a tree is considered this equivalence also takes different leaves into account.

The algorithm given in Figure 2 converts a sequence L of a query graph's edges into a valid processing tree. Starting with a forest of $|V|$ trivial trees that consist of a root node only, trees are merged pairwise by adding a common root node to both trees, indicated by $\oplus$.

LEMMA 3.3
For a sequence L over a query graph's edges, algorithm MERGETREE constructs one single tree only, i.e. $T_{|L|}$ contains only one tree.

P r o o f : Assume, to the contrary, $T_{|L|} = \{t_1, t_2, \ldots, t_m\}$. Thus, no edges between leaves of $t_1, \ldots, t_m$ were in L, which means, that G was not connected which contradicts the definition of query graphs. □

LEMMA 3.4
The processing tree computed by MERGETREE is a valid tree in the sense of Definition 3.1.

P r o o f : T_i always contains valid processing trees for disjointed subgraphs of G:

$\underline{i = 0:}$ all elements t_j of T_0 are trivial trees of height 0, thus valid trees for the Graphs $G_{t_j}(\{v_j\}, \emptyset)$.

$\underline{i \to i+1:}$ let (n_a, n_b) be the next edge that is to be added and t_a and t_b be elements of T_i with leaves n_a and n_b, respectively. Note, that T_i contains at least a tree of height 0 with leaf v for every possible leaf. If the edge connects nodes, that are leaves of the same tree the shape of the processing tree does not change, according to its definition, and thus the proposition holds. Otherwise, the edge connects the two graphs G_{t_a} and G_{t_b}, with $G_{t_i} = G(\{\text{leaves of } t_i\})$. Therefore, G_{t_a}, G_{t_b} and $G_{t_a \oplus t_b}$ fulfill the conditions of Definition 3.1 which completes the proof. □

Algorithm MergeTree
Input L sequence of edges,
n_i nodes of the query graph
Output $T_{|L|}$ processing tree

$i \leftarrow 0$
$t_j \leftarrow n_j$
$T_0 \leftarrow \bigcup_j t_j$
$L_0 \leftarrow L$
while $|L_i| > 0$ **do**
  let (n_a, n_b) be e_1 of L_i
  let $t_a \in T_i$ be tree where n_a is leaf
  let $t_b \in T_i$ be tree where n_b is leaf
  if $t_a \neq t_b$ **do**
    $t \leftarrow t_a \oplus t_b$
    $T_{i+1} \leftarrow (T_i \setminus \{t_a, t_b\}) \cup t$
  done
  annotate deepest common ancestor of n_a and n_b
    with predicate of e_1
  $L_{i+1} \leftarrow L_i \setminus \{e_1\}$
  $i \leftarrow i + 1$
done

Figure 2: Algorithm MergeTree

Proposition 3.5
For a query graph $G(V, E)$, MergeTree converts every sequence over E into a valid processing tree.[1]

P r o o f : Follows immediately from Lemma 3.3 and 3.4. □

For the remainder of this paper, we refer to the forest of processing trees generated from the sequence L by MergeTree as $T(L)$. Two forests are equivalent if every tree of one forest has an equivalent in the other forest. If $T(L)$ contains only one element, we identify this element also with $T(L)$, due to isomorphism.

Definition 3.6
A sequence L over E is *rank-minimal* iff

$$\forall L' \in E^* : T(L) \cong T(L') \Rightarrow r(L) < r(L') \tag{3.2}$$

i.e. every sequence that generates an equivalent tree has greater rank. ◇

Monotonicity of the sequence elements implies *prefix monotonicity* of rank-minimality as follows:

Corollary 3.7
Let L be sequence over $F \subset E$ and $e \in E$ with $\eta(e) > \eta(f)$ for all f in L then

$$L \text{ is rank-minimal} \Leftrightarrow L :: e \text{ is rank-minimal} \tag{3.3}$$

[1] We present a discussion of the time complexity for all algorithms in Section 6.

holds.

Consequently, the η-sorted sequences are always rank-minimal.

4. Redundancies

The fact, that not every edge of a sequence prompts MergeTree to change the shape of some tree already suggests that sequences may contain certain redundancies. In deed, generating the $n!$ different sequences of length n would yield lots of equivalent trees. The redundancies resulting from edges in cyclic graphs is the concern of this section.

Definition 4.1
For a sequence $L = \langle e_1, e_2, \ldots, e_n \rangle$ the set of redundant edges for every element is given by:

$$\rho_L(e_i) = \{e_j | \eta(e_i) < \eta(e_j), T(L) \cong T(L \setminus \{e_j\}) \tag{4.1}$$

$R(L) = \bigcup_{i=1}^{n} \rho_L(i)$ is called *redundancy* of L. ◇

Clearly, redundancy is a property inherent to the sequence and not only to single edges. However, the redundant-edge property is transitive, i.e. for two edges e_i and e_j in L the following holds:

$$\rho_L(e_i) \cap \rho_L(e_j) \neq \emptyset \quad \Rightarrow \quad e_j \in \rho_L(e_i) \vee e_i \in \rho_L(e_j) \tag{4.2}$$

Removing the redundancy from a sequence does not affect the shape of the resulting tree, i.e. $T(L) \cong T(L \setminus R)$. Hence, redundancy is prefix monotonic, in the following sense $R(\langle e_1, \ldots, e_{n-1} \rangle) \subseteq R(\langle e_1, \ldots, e_{n-1}, e_n \rangle)$.

Proposition 4.2
Let L be a sequence over $F \subset E$ and $e \in E \setminus F$. Algorithm ComputeRedundancy computes $R(L :: e)$.

P r o o f : Since $R(L)$ is already input parameter, only the redundancy added by e has to be computed. Redundancy occurs if there is more than one edge between one component of $G(L)$ and G_e. For every component of $G(L)$ the algorithm checks all remaining edges in $E \setminus L \setminus R(L)$ whether they connect the graphs and all but the first fulfilling the condition are added to R'. Thus R' is $R(L :: e)$. □

5. Generating the Trees

To generate all non-isomorphic processing trees $P(G)$ for a given query graph G, we generate a minimal set $\widehat{L}$ of edge sequences that are isomorphic to $P(G)$. This set is given by

$$\widehat{L} \subseteq E^*, \qquad \text{with} \quad L_1, L_2 \in \widehat{L} \Rightarrow T(L_1) \ncong T(L_2) \tag{5.1}$$

An algorithmically more practical form is the following one:

$$\begin{aligned} \widehat{L} &= \{L \in E^* | L \text{ rank-minimal}\} \\ &= \{L \in E^* | L = L' \setminus R(L') :: \widetilde{R}(L'), \\ &\qquad\qquad L' \in E^*, \quad L' \setminus R(L') \text{ rank-minimal}\} \end{aligned} \tag{5.2}$$

Algorithm ComputeRedundancy
Input L sequence of edges
R set of redundant edges w.r.t. L,
e edge with $e \in E \setminus R$

let G_e be subgraph of $G(L :: e)$ containing e
foreach connected subgraph $G' \in G(L :: e) \setminus G_e$ **do**
  $found \leftarrow false$
  foreach $f \in E \setminus L \setminus R$ **do**
    if f connects G' with G_e **do**
      if *found* **do**
        $R' \leftarrow R' \cup \{f\}$
      done
      $found \leftarrow true$
    done
  done
done

Figure 3: Algorithm ComputeRedundancy

The algorithm ExpandSequence, given in Figure 4, generates the sought sequences. The main loop iterates over the number of edges and calls itself recursively up to n times. The deepest recursion is reached once the sequence is either complete, or can be completed immediately by adding the redundancy as a η-sorted suffix. Since the basic design of the loop allows for the maximal possible set of $n!$ sequences, the algorithm has only to avoid the generation of equivalents, that is, discard not rank-minimal prefixes. According to Corollary 3.7, rank-minimality is prefix monotonic, thus we can judge for each edge e whether $L :: e$ is a valid prefix or not.

Lemma 5.1
Let L be a rank-minimal sequence over $F \subset E$, and $e \in E \setminus R(L)$ with $\eta(e) > \eta(f)$ where f is the edge of L with $\eta(f) = \max_{e' \in F} \eta(e')$. Then $r(L :: e)$ is rank-minimal.

Proof: Since rank-minimality is prefix monotonic, $L :: e$ is the rank-minimal sequence containing all elements of L and e. □

Lemma 5.2
Let L be a rank-minimal sequence over $F \subset E$, and $e \in E \setminus R(L)$ with $\eta(e) < \eta(f)$ where f is defined as above. The following holds

$$r(L :: e) \text{ rank-minimal} \Leftrightarrow e \text{ is adjacent to } G_f \tag{5.3}$$

where G_f denotes the component of $G(L)$ that covers f.

Proof: Let e_i be the elements of L, i.e. $L = \langle e_1, \ldots, e_{|L|} \rangle$.

Firstly, assume, to the contrary, e is not adjacent to G_f. Let e_k be $\min_i \{\eta(e_i) \in L | \eta(e) < \eta(e_i)\}$, and G_a, G_b the components of G connected by e. Furthermore, w.l.o.g. $|G_a| \leq |G_b|$.

$\underline{|G_b| = 1}$: e is not connected to any of $G(L)$ components and inserting it at any position in the sequence does not change $T(L)$.

Thus, $r(\langle e_1, \ldots, e_{k-1}, e, e_k, \ldots, e_{|L|}\rangle) < r(L :: e)$, i.e. $L :: e$ is not rank-minimal.

$\underline{|G_b| > 1}$: Let $\langle b_1, \ldots, b_{|G_b|}\rangle$ be the subsequence that defines G_b only, i.e. $L \setminus \overline{(E \setminus G_b)}$. If $|G_a| > 1$, $\langle a_1, \ldots, a_{|G_a|}\rangle$ is defined analogously. Otherwise, let $a_{|G_a|}$ be $b_{|G_b|}$. The first position in L where e can occur without affecting the equivalence is after both $b_{|G_b|}$ and $a_{|G_a|}$. Since both G_a and G_b are not connected to G_e, $r(\langle e_1, \ldots, e, e_{|L|}\rangle) < r(\langle e_1, \ldots, e_{|L|}, e\rangle) = r(L :: e)$, i.e. $L :: e$ is not rank-minimal.

To show the opposite direction, assume $L :: e$ is not rank-minimal. Then, a sequence L' with $T(L :: e) \cong T(L')$ exists where e is not the last element, i.e. $L' = \langle e_1, \ldots, e, \ldots, e_{|L|}\rangle$. However, inserting e before $e_{|L|}$ does not effect equivalence of $T(L') \cong T(L :: e)$ only if e is not adjacent to G_f. □

Proposition 5.3
Algorithm ExpandSequence computes all sequences of $\widehat{L}$ with prefix L.

P r o o f : The inner loop of the procedure potentially generates all prefixes. For every prefix, built incrementally, edges that are not already in the prefix or element of the redundancy of this prefix are checked for being added to the prefix. Thus, we show that only rank-minimal redundance-free prefixes are generated by induction over the prefix length.

$\underline{i = 1}$: trivial.

$\underline{i \to i + 1}$: Let $e \in E \setminus L \setminus R(L)$ be the edge we check. Either $\eta(e) > \eta(e_i)$ for all e_i that are in L so far. The proposition follows with Lemma 5.1. Otherwise, if e is adjacent to the component covering η_{max} and Lemma 5.2 completes the proof. □

Invoking ExpandSequence with all possible prefixes $L_i = \langle e_i\rangle$ yields $\widehat{L}$.[2]

Corollary 5.4
The number of non-isomorphic trees enumerated by ExpandSequence is $N_{clique}(n) = \frac{(2n-2)!}{(n-1)!}2^{1-n}$ if the query graph forms a clique of size n, and $N_{chain}(n) = \frac{(2n-2)!}{n!(n-1)!}$ in case the query graph is a chain.

P r o o f : Independent of the query graph's shape, all processing trees have n leaves, 2^{n-1} nodes in total, and 2^{n-2} inner nodes. Every isomorphic class has $2^{n-2} \cdot 2$ elements. The numbers for spaces *including* isomorphic trees [5] are known to be $N^{iso}_{clique}(n) = \frac{(2n-2)!}{(n-1)!}$ and $N^{iso}_{chain}(n) = \frac{(2n-2)!}{n!(n-1)!}2^{n-1}$. Dividing those figures by 2^{n-1} yields the proposition. □

Finally, with a simple modification we can restrict the algorithm to enumerate only linear processing trees—the most prominent group of trees since the early days of query optimization [8, 4]. When dropping the condition $\eta(e) > \eta_{\max}$,

[2]The actual implementation, in fact, expands all sequences from the empty sequence. However, for simplicity we omitted the parts necessary for proper treatment of $\langle\rangle$.

Algorithm ExpandSequence
Input L sequence of edges,
R set of redundant edges w.r.t. L

if $E \setminus L \setminus R = \emptyset$ **do**
MergeTree$(L :: \widetilde{R})$
return
done

$\eta_{\max} \leftarrow \max_i \{\eta(e_i) | e_i \in L\}$
let $G_{\eta_{max}}$ be subgraph of $G(L)$ covering $e_{\eta_{max}}$
foreach $e \in E \setminus L \setminus R$ **do**
if ($\eta(e) > \eta_{\max} \ \vee$ e is adjacent to $G_{\eta_{max}}$) **do**
$L' \leftarrow L :: e$
$R' \leftarrow$ ComputeRedundancy(L', R, e)
ExpandSequence(L', R')
done
done

Figure 4: Algorithm ExpandSequence

we append only edges that are adjacent to $G_{\eta_{max}}$, so at least one of its nodes is already part of the tree, i.e. we add either the mere predicate or a subtree that consists of a leaf only. Thus, the result is a processing tree of height $n-1$.

Corollary 5.5
The number of non-isomorphic *linear* trees enumerated by ExpandSequence is $N_{clique}(n) = \frac{n!}{2}$ if the query graph forms a clique of size n. In case the query graph is a chain we enumerate $N_{chain}(n) = 2^{n-2}$ trees.

P r o o f : In either case we need to focus on the non-redundant edges only. For the clique, every edge that connects to a node which is not part of the prefix yet, is non-redundant. Thus, for every prefix of length 1, there are $(n-2)!$ completions. The very first edge can be chosen in $\binom{n}{2} = \frac{n!}{2(n-2)!}$ ways. Therefore, the total number is $\frac{n!}{2(n-2)!} \cdot (n-2)! = \frac{n!}{2}$.

The situation for a chain is as follows. Selecting a prefix of lenght 1 splits the query graph into two subchains that are to be merged by ExpandSequence. This can be done in $\binom{l_r - l_l}{l_l}$ ways, with l_l and l_r is length of the left and right sub-chain, respectively. Applying this to all $n-1$ prefixes of length 1 yields $N_{chain}(n) = \sum_{k=0}^{n-2} \binom{n-2}{k}$ which can be rewritten to 2^{n-2}. □

6. Discussion

In this section we scrutinize the presented techniques with respect to an efficient implementation.

One of the sore points in general is the membership test for sets. Since it is used very frequently, it deserves attention before we focus on the actual

algorithms. As our sets are limited to small sizes we can represent them by bit-vectors which reduces both test and insert/remove operations to $O(1)$.

MergeTree can transform a sequence into its corresponding processing tree within $O(|E|^2)$ if we allow the non-recurring construction of a directory of leaves. The construction is in $O(|E|)$ and requires no updates during running time. Another critical issue is the traversal of connected components in ComputeRedundancy. Naively identifying the components anew with every invocation is in $O(|E|)$. However, the incremental nature of the changes to the components—adding an edge leaves all other members of the component unchanged—suggests a directory of components. The overhead caused by its maintenance is in $O(|V|)$. This reduces the cost of ComputeRedundancy from $O(|E|^2)$ to $O(|V| \cdot |E|)$. Hence, the construction of a processing tree is in $O(|V| \cdot |E|^2)$.

We deliberately used $O(|V|)$ and $O(|E|)$ despite the fact that $O(|E|)$ is in $O(|V|^2)$ since the number of edges exceeds the number of nodes slightly in typical database applications. Queries that correspond to clique graphs are of no practical impact and solely used as worst-case scenarios. In the majority of all cases $|E|$ is close to $|V|$.

Finally, what is not expressed by the time complexity is the extent of re-use. For simplicity we presented the single procedures as separate from each other as possible. However, to gain the necessary performance, the call to MergeTree in ExpandSequence should not be postponed until the sequence is complete and—as the notation suggests—be discarded afterwards, but handled incrementally. We modify MergeTree in a way that single edges can be added and removed from the tree or forest, respectively. The adding operation is then called before, the remove after the recursive invocation. With this modification, one tree is incrementally built, subsequently pruned, and merged again. Large parts of the tree and of the sequence prefix are not modified when going on to the next processing tree. In contrast to other enumeration techniques, the re-use is inherent in the method and does not require any additional memory nor running time spent on lookups.

7. Conclusion

In this paper, we addressed the question how to make use of the processing trees' topology in order to lower the number of trees that have to be visited, significantly. Our approach based on the expansion of sequence prefixes achieves a simple yet powerful algorithm for the enumeration of non-isomorphic processing trees. A method, that has direct application to existing query optimizers that are deploying enumeration techniques since it prunes the size of the search space massively by factor 2^k, for a query involving k joins.

Future Work. Though answering some of the questions raised in [2] some are left unanswered. Our further research is directed toward the investigation of the complexity of *counting* and the efficient generation of single non-isomorphic trees at random. A technique that proved superior to transformation-based optimization methods for the case of tree-shaped query graphs. Moreover, our long-term research goal is a characterization of the cost distribution in these spaces imposed by standard cost models.

References

1. C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, Randomized Join-Order Selection – Why Use Transformations? In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 85–95, Santiago, Chile, September 1994.
2. C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Uniformly-distributed Random Generation of Join Orders. In *Proc. Int'l. Conf. on Database Theory*, pages 280–293, Prague, Czechia, January 1995.
3. Y. E. Ioannidis and Y. C. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proc. ACM SIGMOD Int'l. Conf.*, pages 312–321, Atlantic City, NJ, USA, May 1990.
4. Y. E. Ioannidis and Y. C. Kang. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. In *Proc. ACM SIGMOD Int'l. Conf.*, pages 168–177, Denver, CO, USA, May 1991.
5. R. S. G. Lanzelotte, P. Valduriez, and M. Zaït. On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 493–504, Dublin, Ireland, August 1993.
6. A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. Duplicate-free Generation of Alternatives in Transformation-based Optimizers. In *Proc. Int'l. Conference on Database Systems for Advanced Applications*, pages 117–123, Melbourne, Australia, April 1997.
7. A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. The Complexity of Transformation-Based Join Enumeration. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 306–315, Athens, Greece, 1997.
8. P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM SIGMOD Int'l. Conf.*, pages 23–34, Boston, MA, USA, May 1979.

9. B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. In *Proc. ACM SIGMOD Int'l. Conf.*, pages 35–46, Montreal, Canada, June 1996.