Reassessing Top-Down Join Enumeration

Pit Fender and Guido Moerkotte

Abstract—Finding an optimal execution order of join operations is a crucial task in every cost-based query optimizer. Since there are many possible join trees for a given query, the overhead of the join (tree) enumeration algorithm per valid join tree should be minimal. In the case of a clique-shaped query graph, the best known top-down algorithm has a complexity of $\Theta(n^2)$ per join tree, where n is the number of relations. In this paper, we present an algorithm that has an according $O(1)$ complexity in this case. We show experimentally that this more theoretical result has indeed a high impact on the performance in other nonclique settings. This is especially true for cyclic query graphs. Further, we evaluate the performance of our new algorithm and compare it with the best top-down and bottom-up algorithms described in the literature.

Index Terms—Query optimization, join ordering, top-down join enumeration, memoization, graph partitioning, minimal cut.

1 Introduction

For a DBMS that provides support for a declarative query language like SQL, the query optimizer is a crucial piece of software. The declarative nature of a query allows it to be translated into many equivalent evaluation plans. Essential for the costs of a plan is the execution order of join operations in its operator tree. An exhaustive search for an optimal solution over all possible operator trees is computationally infeasible. To decrease complexity, the search space must be restricted. For the optimization problem discussed in this document, a well-accepted heuristic is applied: we consider all possible bushy join trees [1], but exclude cross products from the search, presuming that all considered queries span a connected query graph [2].

When designing a query optimizer, there are two strategies to find an optimal join order: bottom-up join enumeration via dynamic programming, and top-down join enumeration through memoization. Both approaches (naturally) have to explore the same search space and both face the same challenges. Let us briefly recall the main challenge. This requires a little preparation.

For every subset S of relations that induces a connected subgraph (`csg` for short), the optimal join tree must be constructed. In order to determine the best join tree for a given subset S of relations, the plan generator must enumerate all partitions (S_1, S_2) of S such that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$. Furthermore, since we exclude cross products, S_1 and S_2 must induce connected subgraphs of our query graph, and there must be two relations $R_1 \in S_1$ and $R_2 \in S_2$ such that they are connected by an edge, i.e., there must exist a join predicate involving attributes in R_1 and R_2. Let us call such a partition (S_1, S_2) a *csg-cmp-pair* (or `ccp` for short). Denote by T_i the best plan for S_i. Then the query optimizer has to consider the plans $T_1 \bowtie T_2$ for all csg-cmp-pairs (S_1, S_2).

One possibility to generate all csg-cmp-pairs for a set S of relations is to consider all subsets $S_1 \subset S$, define $S_2 = S \setminus S_1$, and then check the above conditions. Let us call such a procedure *naive generate and test* or `ngt` for short.

Table 1 gives for $n = 5, 10, 15, 20$ relations the number of connected subgraphs (#csg), the number of csg-cmp-pairs (#ccp), and the number of generated subsets S_1 for the naive generate and test algorithm (#ngt). These numbers were determined analytically ([2], [3]), but the formulas are not very intuitive. Therefore, we decided to illustrate our points with some explicit numbers.[1]

Challenge. The number of subsets considered by naive generate and test is several orders of magnitude higher than the number of csg-cmp-pairs. Thus, this approach is too inefficient to be useful (see also Section 4). Hence, the challenge is to generate only valid csg-cmp-pairs and to do this with as little overhead as possible. For quite a long time, no efficient enumerator for csg-cmp-pairs was known. In bottom-up join enumeration, all the connected subsets for a given set are already generated. Therefore, an enumeration strategy for dynamic programming that is not generate-and-test based should be easier to design. Moerkotte and Neumann [3] presented a dynamic programming variant called DPCCP generating csg-cmp-pairs within constant time $O(1)$ each. DeHaan and Tompa took up the even greater challenge and came up with a minimal graph cut partitioning algorithm called MINCUTLAZY for top-down join enumeration [4].[2] In case of acyclic query graphs, the complexity for generating a csg-cmp-pair is also $O(1)$ [5]. However, for cyclic query graphs the complexity increases and reaches a maximum for cliques, where it is $O(n^2)$ for n relations [5].

Fortunate observation. The number of connected subgraphs (#csg) is far lower than the number of csg-cmp-pairs.

• *The authors are with the Database Research Group, University of Mannheim, B6, 29, Gebäudeteil C, Mannheim 68131, Germany. E-mail: pfender@db.informatik.uni-mannheim.de, moerkotte@informatik.uni-mannheim.de.*

Manuscript received 27 May 2011; revised 17 Oct. 2011; accepted 18 Oct. 2011; published online 10 Nov. 2011.
Recommended for acceptance by S. Abiteboul, C. Koch, K.-L. Tan, and J. Pei. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDESI-2011-05-0298.
Digital Object Identifier no. 10.1109/TKDE.2011.235.

1. The difference by a factor of two between #ccp and #ngt for cliques is explained in the next section.

2. Actually, they presented two algorithms, but one does not completely enumerate all csg-cmp-pairs (see Sections 3.3 and 6.3).

TABLE 1
Counters for Different Graphs

		5	10	15	20
chain	#csg	15	55	120	210
	#ccp	20	165	560	1330
	#ngt	84	3962	130798	4193840
star	#csg	20	521	16398	524307
	#ccp	32	2304	114688	4980736
	#ngt	130	38342	9533170	2323474358
cycle	#csg	21	91	211	381
	#ccp	40	405	1470	3610
	#ngt	140	11062	523836	22019294
clique	#csg	31	1023	32767	1048575
	#ccp	90	28501	7141686	1742343625
	#ngt	180	57002	14283372	3484687250

This is very fortunate, since 1) #csg is the number of times the cardinality estimation takes place, 2) #ccp is the number of times the join cost function is evaluated, and 3) the latter is an order of magnitude cheaper than the former. Indeed, a typical join cost function only takes a few arithmetic operations to evaluate [6].

Contribution. We propose a new and highly efficient top-down join enumeration algorithm and evaluate its performance. More specifically, we

- conduct a detailed complexity analysis of MINCUT LAZY, showing its $O(n^2)$ complexity for clique queries [5],
- discuss how MINCUTLAZY can be improved,
- present an advance generate and test partitioning algorithm called MINCUTAGAT,
- propose MINCUTBRANCH as an entirely new partitioning approach for top-down join enumeration,
- show analytically that the complexity of MINCUTBRANCH is in $O(1)$ for acyclic graphs, cycle graphs, and clique queries [5], and
- present an in-depth performance evaluation indicating that one new enumerating algorithm is nearly as efficient as DPCCP.

Apart from their efficiency, our new algorithms have the great advantage that they are much easier to implement than the one developed by DeHaan and Tompa. While they need to build and maintain a complex data structure called biconnection tree [5], we only need set operations, which can be implemented easily and efficiently using bit vectors.

Important note. Let us note that branch-and-bound pruning may be effective for some queries. However, pruning gives the same advantage to all top-down algorithms. Thus, we decided to ignore its effects here. Leaving out pruning has the additional advantage that a fair comparison of the raw performance with bottom-up approaches, which cannot prune easily, becomes possible.

Organization. This paper is organized as follows: Section 2 recalls some preliminaries. Section 3 explains MINCUTLAZY and discusses an improvement. Furthermore, two novel partitioning algorithms for top-down join enumeration are proposed. Section 4 presents a thorough performance evaluation. Section 5 concludes the paper. Section 6 contains a thorough discussion on biconnection trees and explains an improved connection test.

2 Preliminaries

Before we start with our discussions, we give some fundamentals. In the first section, we explain important notions and then continue with an introduction to top-down join enumeration. We present a generic memoization algorithm for join optimization that can be instantiated with different enumeration strategies for csg-cmp-pairs, which we also call partitioning strategies or algorithms. The last part of this section explains the naive generate-and-test partitioning algorithm.

2.1 Important Notions

In this section, we give some definitions that are important for a thorough understanding of the work presented here.

Our focus is to determine an optimal join order for a given query. The execution order of join operations is specified by an operator tree of the physical algebra. For our purposes, we want to abstract from that representation and give the notion of a *join tree*. A join tree is a binary tree where the leaf nodes specify the relations referenced in a query, and the inner nodes specify the two-way join operations. The edges of the join tree represent sets of joined relations. Two input sets of relations that qualify for a join so that no cross products need to be considered are called a *connected subgraph and its complement pair* or ccp for short [3].

Definition 2.1. *Let $G = (V, E)$ be a connected query graph, (S_1, S_2) is a* connected subgraph and its complement pair *(or* csg-cmp-pair*, or, even shorter,* ccp*) if the following holds:*

- *S_1 with $S_1 \subset V$ induces a connected graph $G_{|S_1}$,*
- *S_2 with $S_2 \subset V$ induces a connected graph $G_{|S_2}$,*
- *$S_1 \cap S_2 = \emptyset$, and*
- *$\exists (v_1, v_2) \in E | v_1 \in S_1 \wedge v_2 \in S_2$.*

The set of all possible ccps is denoted by P_{ccp}. We introduce the notion of *cmp-csg pairs for a set* to specify all those pairs of input sets that result in the same output set, if joined.

Definition 2.2. *Let $G = (V, E)$ be a connected query graph and S a set with $S \subseteq V$ that induces a connected subgraph $G_{|S}$. For $S_1, S_2 \subset V$, (S_1, S_2) is called a* ccp *for S if (S_1, S_2) is a ccp and $S_1 \cup S_2 = S$ holds.*

By $P_{ccp}(S)$, we denote the set of all ccps for S. Let $\mathcal{P}_{con}(V) = \{S \subseteq V | G_{|S} \text{ is connected} \wedge |S| > 1\}$ be the set of all connected subsets of V with more than one element, then $P_{ccp} = \cup_{S \in \mathcal{P}_{con}(V)} P_{ccp}(S)$ holds.

If (S_1, S_2) is a ccp, then (S_2, S_1) is one as well, and we consider them as symmetric pairs. We are interested in the set P_{ccp}^{sym} of all ccps, where symmetric pairs are accounted for only once, e.g., $(S_1, S_2) \in P_{ccp}^{sym}$ if $max_{index}(S_1) \leq max_{index}(S_2)$ holds, or $(S_2, S_1) \in P_{ccp}^{sym}$ otherwise. We give no constraints for choosing which one of two symmetric pairs should be member of P_{ccp}^{sym}, but leave this as a degree of freedom. Analogously, we denote the set of all ccps for a set S containing either (S_1, S_2) or (S_2, S_1) by $P_{ccp}^{sym}(S)$. The lower bounds for join enumeration given by Ono and Lohman [2] (see Table 1) for certain graph shapes correspond to $|P_{ccp}^{sym}|$. Thus, this explains by a factor of two difference between #ccp and #ngt for cliques.

Next, we define the neighborhood of a set of nodes.

Definition 2.3. *Let $G = (V, E)$ be an undirected graph, the neighborhood of a set $S \subseteq V$ is defined as*

$$\mathcal{N}(S) = \{w \in (V \setminus S) | v \in S \wedge (v, w) \in E\}.$$

The next definition is rather standard, for more details see [7].

Definition 2.4. *Let $G = (V, E)$ be an undirected connected graph. A vertex $a \in V$ is called articulation vertex if there exist two vertices $v \in V$ and $w \in V$, such that every path $v \overset{*}{\rightarrow} w$ in V must contain a.*

The articulation vertices of a connected graph $G = (V, E)$ are important when determining the biconnected components of a graph.

Definition 2.5. *Let $G = (V, E)$ be a connected undirected graph. A* biconnected component *is a connected subgraph $G_i^{BCC} = (V_i, E_i)$ of G with $V_i = \{v | (v = u \vee v = w) \wedge (v, w) \in E_i\}$, where the set of edges $E_i \subseteq E$ is maximal such that any two distinct edges $(u, w) \in E_i$ and $(x, y) \in E_i$ lie on a cycle $\langle v_0, v_1, v_2, \ldots, v_l \rangle$, where $u = v_0 \wedge u = v_l \wedge w = v_1 \wedge x = v_{j-1} \wedge y = v_j \wedge 0 < j < l$ and $\forall_{0 \leq i < j < l} v_i, v_j \in V \wedge v_i \neq v_j$ holds. If for an edge $(u, w) \in E_i$ no such cycle exists, the vertices $u, w \in V_i$ induce a biconnected component $G_i^{BCC} = (\{u, w\}, \{(u, w)\})$.*

DeHaan's and Tompa's MINCUTLAZY makes use of a data structure called biconnection tree. We give its definition:

Definition 2.6. *Let $G = (V, E)$ be a connected undirected graph and $BCC = \{G_1^{BCC}(V_1, E_1), \ldots, G_k^{BCC}(V_k, E_k)\}$ the set of biconnected components of which G consists such that $V = \bigcup_{1 \leq i \leq k} V_i$ holds. For an arbitrary vertex $t \in V$, a set of* vertex nodes *V_{vn} and a set of* set nodes *V_{sn} where $V_{tree} = V_{vn} \cup V_{sn}$ and $V_{vn} \cap V_{sn} = \emptyset$ holds, we call $\mathcal{T} = (V_{tree}, E_{tree}, t)$ a* biconnection tree *if*

- $V_{vn} = V$,
- $V_{sn} = \{s_{V_i} \mid s$ *representing a set of vertices V_i of a biconnected component $G_i^{BCC}(V_i, E_i)\}$, and*
- *the set of tree edges $E_{tree} = \{(s_{V_i}, v) | s_{V_i} \in V_{sn} \wedge v \in V_i\}$.*

The vertex t is called root of $\mathcal{T}$.

Within a biconnection tree $\mathcal{T}$, the descendants $\mathcal{D}_\mathcal{T}$ and the ancestors $\mathcal{A}_\mathcal{T}$ of an arbitrary vertex $v \in V$ can be defined as follows:

$$\mathcal{D}_\mathcal{T}(v) = \{u \in V \mid u \text{ occurs in a subtree of } \mathcal{T} \text{ rooted at } v\},$$

$$\mathcal{A}_\mathcal{T}(v) = \{u \in V \mid u \text{ is a vertex node on path } t \overset{*}{\rightarrow} v\}.$$

2.2 Basic Memoization

As an introduction to top-down join enumeration, we give a basic memoization variant called MEMOIZATIONBASIC, which we derive by utilizing a generic top-down algorithm that invokes a naive partitioning algorithm. In the first Section 2.2.1, we present our generic top-down algorithm. Afterwards, we explain the naive partitioning strategy.

```
TDPLANGEN(G)
    ▷ Input: connected G=(V,E), V = ⋃_{1≤i≤|V|} {R_i}
    ▷ Output: an optimal join tree for G
1   for i ← 1 to |V|
2       do BestTree({R_i}) ← R_i
3   return TDPGSUB(V)

TDPGSUB(G_|S)
    ▷ Input: connected sub graph G_|S
    ▷ Output: an optimal join tree for G_|S
1   if BestTree[S] = NULL
2     then for all (S_1, S_2) ∈ P_ccp^sym(S)
3          do BUILDTREE(G_|S, TDPGSUB(G_|S_1),
                        TDPGSUB(G_|S_2))
4   return BestTree[S]
```

Fig. 1. Pseudocode for TDPLANGEN.

2.2.1 Generic Top-Down Join Enumeration

Our generic top-down join enumeration algorithm TDPLANGEN is based on memoization. We present its pseudocode in Fig. 1. Like dynamic programming, TDPLANGEN initializes the building blocks for atomic relations first (line 2). Then, in line 3 the subroutine TDPGSUB is called, which traverses recursively through the search space. At the root invocation, the vertex set S corresponds to the vertex set V of the query graph. At every recursion step of TDPGSUB, all possible join trees of two optimal subjoin trees that together comprise the relations of S are built through BUILDTREE (line 3) that we explain later, and the cheapest join tree is kept. We enumerate the optimal subjoin trees by iterating over the elements (S_1, S_2) of $P_{ccp}^{sym}(S)$ in line 2. This way, we derive the two optimal subjoin trees, each comprising exactly the relations in S_1 or S_2, respectively, by recursive calls to TDPGSUB. Generating $P_{ccp}^{sym}(S)$ is the task of a partitioning algorithm. Depending on the choice of the partitioning strategy, the overall performance of TDPLANGEN can vary by orders of magnitude.

The recursive descent stops when either $|S| = 1$ or TDPGSUB has already been called for that $G_{|S}$. In both cases, the optimal join tree is already known. To prevent TDPGSUB from computing an optimal tree twice, $BestTree[S]$ is checked in line 1. $BestTree[S]$ yields a reference to an entry in an associative data structure called memotable. The data structure "memoizes" the optimal join tree generated for a set S. If $BestTree[S]$ equals NULL, this invocation of TDPGSUB will be the first one with $G_{|S}$ as input, and the optimal join tree of $G_{|S}$ has not been found yet.

The pseudocode of BUILDTREE is given in Fig. 2. It is used to compare the cost of the join trees that belong to the same $G_{|S}$. Since the symmetric pairs (S_1, S_2) and (S_2, S_1) (line 2 of TDPGSUB) are enumerated only once, we have to build two join trees (line 1 and line 4) and then compare their costs. We use the method CREATETREE, which takes two disjoint join trees as arguments and combines them to a new join tree. If different join implementations have to be considered, among all alternatives the cheapest join tree has to be built by CREATETREE. If the created join tree (line 1) is cheaper than $BestTree[S]$, or even no tree for S has been

```
BUILDTREE(G_|S, Tree_1, Tree_2)
    ▷ Input: inducing a graph G_|S, two sub join trees
1   CurrentTree ← CREATETREE(Tree_1, Tree_2)
2   if BestTree[S] = NULL ||
       cost(BestTree[S]) > cost(CurrentTree)
3      then BestTree[S] ← CurrentTree
4   CurrentTree ← CREATETREE(Tree_2, Tree_1)
5   if cost(BestTree[S]) > cost(CurrentTree)
6      then BestTree[S] ← CurrentTree
```

Fig. 2. Pseudocode for BUILDTREE.

built yet, $BestTree[S]$ gets registered with the $CurrentTree$. For building the second tree, we just exchange the arguments (line 4). Again, the costs of the new join tree are compared to the costs of $BestTree[S]$. Only if the new join tree has lower costs, $BestTree[S]$ gets registered with the new join tree. Estimating the costs of the two possible join trees at the same time rather than separately and comparing them is more efficient, e.g., for cost functions as given in [6], where $card(T_x) \leq card(T_y) \Rightarrow cost(T_x \bowtie T_y) \leq cost(T_y \bowtie T_x)$ holds, with $card$ as the number of tuples or pages and T_x, T_y as (intermediate) relations.

2.2.2 Naive Partitioning

As we have already seen, the generic top-down enumeration algorithm iterates over the elements of $P_{ccp}^{sym}(S)$. Now, we show how the ccps for S can be computed by a naive generate-and-test strategy. We call our algorithm PARTITION_{naive} and give its pseudocode in Fig. 3. In line 1, all $2^{|S|} - 2$ possible nonempty and proper subsets of S are enumerated. For rapid subset enumeration, the method described in [8] can be used. We demand that from every symmetric pair only one is emitted. There are many possible solutions, but we make sure that the relation with the highest index represented in the graph is always contained in the complement $S \setminus S_1$ in line 2. Three conditions have to be met so that a partition $(S_1, S \setminus S_1)$ is a ccp. We check the connectivity of $G_{|S}$ and $G_{|S \setminus S_1}$ in line 2. The third condition that S_1 needs to be connected to $S \setminus S_1$ is ensured implicitly by the requirement that the set S handed over as input is connected.

3 GRAPH-BASED JOIN ENUMERATION

We start off by recapturing DeHaan's and Tompa's lazy minimal cut partitioning algorithm. In the following section, we propose an improvement of their work. Then, Section 3.3 discusses DeHaan's and Tompa's optimistic minimal cut partitioning strategy. We follow with our own minimal cut

```
PARTITION_naive(S)
    ▷ Input: connected set S
    ▷ Output: P_ccp^sym(S)
1   for all S_1 ⊂ S ∧ S_1 ≠ ∅
2       do if max_index(S_1) ≤ max_index(S \ S_1) ∧
              S_1 is connected ∧ S \ S_1 is connected
3          then emit(S_1, S \ S_1)
```

Fig. 3. Pseudocode for naive partitioning.

```
PARTITION_MinCutLazy(S)
    ▷ Input: connected set S
    ▷ Output: P_ccp^sym(S)
1   t ← arbitrary vertex of S
2   MINCUTLAZY(S, ∅, ∅, {t}, NULL, t)

MINCUTLAZY(S, C, C_diff, X, T', t)
    ▷ Input: connected set S, C ∩ X = ∅, C_diff ⊆ C
    ▷ Output: ccps for S
1   if C ≠ ∅
2      then emit (C, S \ C)
3   if N(C) ⊆ X
4      then return
5   if ISUSABLE(T', C_diff)
6      then T ← T'
7      else T ← BUILDBCT(S \ C, t)
8   P ← {v ∈ N(C) | v ∈ (S \ X) ∧
           (D_T(v) ∩ N(C)) = {v}}
9   X' ← X
10  for all v ∈ P
11     do MINCUTLAZY(S, C ∪ D_T(v), D_T(v), X', T, t)
12        X' ← X' ∪ A_T(v)
    ▷ N(∅) = S \ {t}
```

Fig. 4. Pseudocode for MINCUTLAZY.

partitioning algorithm called MINCUTAGAT for top-down join enumeration that overcomes the flaws of the previously discussed optimistic partitioning algorithm. Upon that, we improve our proposed algorithm with a novel technique and present an entirely new join partitioning algorithm called MINCUTBRANCH.

3.1 Lazy Minimal Cut Partitioning

DeHaan and Tompa [4] proposed a partitioning algorithm named MINCUTLAZY that generates the ccps for a set. We give its pseudocode in Fig. 4. MINCUTLAZY can be embedded into a top-down memoization algorithm like TDPLANGEN, as given in Section 2.2.1. We refer to the top-down join enumeration algorithm instantiated with DeHaan's and Tompa's MINCUTLAZY as TDMINCUTLAZY.

DeHaan's and Tompa's partitioning algorithm starts with one-element sets of the relations C and expands them recursively by the descendants $\mathcal{D}_{\mathcal{T}}(v)$ of a neighbor $v \in \mathcal{N}(C)$, but does not cause the complement $S \setminus C$ to become disconnected. Duplicates are avoided through a restricted set X that is enhanced by the ancestors $\mathcal{A}_{\mathcal{T}}(v)$ of a C's neighbor $v \in \mathcal{N}(C)$ after every recursive call. The calculation of descendants and ancestors is based on a biconnection tree structure $\mathcal{T}$ that is computed by a call to BUILDBCT (Section 6.1). We analyze the complexity to build a biconnection tree in Section 6.1.2. It is $O(|S|^2)$. To avoid the unnecessary recomputation of the biconnection tree at every invocation of MINCUTLAZY, the reusability test ISUSABLE [4] is proposed. Since the test returns false negatives, the partitioning algorithm in the worst case constructs a biconnection tree for every emitted partition. But in the best case, that is, for all acyclic graphs, only one biconnection tree is constructed. We give a detailed complexity analysis in [5].

TABLE 2
Number of Biconnection Tree Buildings for TDMINCUTLAZY and TDMINCUTLAZYIMP

	Chain			Star			Clique	
$\|V\|$	$\#t$	$min(\#t_{imp})$	$max(\#t_{imp})$	$\#t$	$min(\#t_{imp})$	$max(\#t_{imp})$	$\#t$	$\#t_{imp}$
3	3	1	2	3	1	2	5	4
4	6	2	4	7	1	4	18	15
5	10	3	7	15	1	7	58	50
10	45	17	37	511	1	37	14757	14283
15	105	43	92	16383	1	92	3587219	3570928
20	190	82	172	524287	1	172	871696090	871171975

3.2 Improved Version

In this section, we present an improvement of TDMINCUTLAZY. We call the improvement TDMINCUTLAZYIMP. First, we explain the idea of our improvement and then, we analyze the number of tree buildings for both versions.

3.2.1 Global Reuse of the Biconnection Tree

In [4], the reusability of the biconnection tree within a call to the minimal cut partitioning algorithm is discussed. We propose a global reuse of the biconnection tree. We observe that once MINCUTLAZY emits a ccp $(C, S \setminus C)$ for a set S, the top-down memoization algorithm recursively invokes itself, once with the C then with $(S \setminus C)$ as the next S. Hence, the minimal cut partitioning algorithm is called again with connected subsets of the old S. Our idea is to share the biconnection tree with the memoization algorithm TDPLANGEN and reuse it for the subsequent call to the partitioning algorithm with $S \setminus C$ as the next S. Therefore, the recursion of MINCUTLAZY must be transformed into an iterative stack implementation such that the emitted ccps for a set are available before the call of the partitioning algorithm returns to TDPLANGEN. This way, one instance of the biconnection tree can be shared at a time. Note that the existing tree cannot be reused for the top-down descent with C, since the root t of the biconnection tree is not an element of C, which is necessary, as pointed out in [4].

3.2.2 Analyzing the Number of Tree Buildings

We have analyzed the potential of our improvement. Therefore, we derived formulas (Section 6.2) for the number of biconnection tree buildings in the best and the worst case that TDMINCUTLAZY needs for enumerating chain, star, and clique query graphs

In Table 2, we give the number of tree buildings for TDMINCUTLAZY with $\#t$ and for TDMINCUTLAZYIMP with $\#t_{imp}$.

3.3 Optimistic Graph Partitioning

DeHaan and Tompa [9] propose a partitioning algorithm called MINCUTOPTIMISTIC. In [4], they describe their algorithm as an optimistic minimal cut partitioning strategy and introduce it as a simplification of MINCUTLAZY. We give the pseudocode in Fig. 5. The algorithm does not exploit the information of a biconnection tree, but relies on a more sophisticated generate-and-test approach. In contrast to naive partitioning (Section 2.2.2), it enlarges C systematically by the neighbors $\mathcal{N}(C)$ of C (line 4 of MINCUTOPTIMISTIC). That way, only the complement needs to be checked for connectivity. Like MINCUTLAZY, it defines the neighbors in the root invocation as $S \setminus \{t\}$, but starts the recursive descent with all elements of the initial neighborhood $\mathcal{N}(\emptyset)$, including any articulation vertices [10]. To avoid enumerating the same set C twice, an exclusive filter set X is used. But MINCUTOPTIMISTIC is not correct, because it does not enumerate completely, as the counterexample in Section 6.3 shows.

3.4 Advanced Generate and Test

This section describes a novel partitioning algorithm that we have developed by adopting the basic concept of optimistic partitioning. Due to a sophisticated generate-and-test approach, the algorithm emits all ccps for a connected vertex set S for which $S \subseteq V$ holds and where V is the vertex set of the query graph $G = (V, E)$. The new algorithm is called MINCUTAGAT, and its pseudocode is given in Fig. 6. We denote the instantiated top-down memoization variant by TDMINCUTAGAT.

The partitioning algorithm is invoked by $\text{PARTITION}_{MinCutAGaT}$, which in turn immediately calls the recursive component MINCUTAGAT. The main idea is to successively enhance a connected set C with one of its neighbors at every recursive iteration. The initial set for C consisting of an arbitrary vertex t is handed over in line 1 of $\text{PARTITION}_{MinCutAGaT}$. Since S and $C \subset S$ are connected, for every possible complement $S \setminus C$ there must exist a join edge (v_1, v_2) where $v_1 \in C$ and $v_2 \in (S \setminus C)$ holds. If $S \setminus C$ is connected, then the partition $(C, S \setminus C)$ is a ccp and can be emitted (line 1). As a requirement implied by the definition of $P^{sym}_{ccp}(S)$, duplicate ccps are prohibited and symmetric

```
PARTITION_MinCutOptimistic(S)
    ▷ Input: connected set S
    ▷ Output: P^sym_ccp(S)
1   t ← arbitrary vertex of S
2   MINCUTOPTIMISTIC(S, ∅, {t}, t)

MINCUTOPTIMISTIC(S, C, X, t)
    ▷ Input: S, disjoint sets C, X ⊆ S
    ▷ Output: ccps for S
1   if C ≠ ∅
2      then emit (C, S \ C)
3   X' ← X
4   for v ∈ (N(C) \ X)      ▷ N(∅) = S \ {t}
5         do C' ← C ∪ {v}
6            if (S \ C is connected
7               then MINCUTOPTIMISTIC(S, C', X')
8            X' ← X' ∪ {v}
```

Fig. 5. Pseudocode for MINCUTOPTIMISTIC.

```
PARTITION_MinCutAGaT(S)
    ▷ Input: a connected set S, arbitrary vertex t ∈ S
    ▷ Output: all ccps for S
1   MINCUTAGAT(S, {t}, ∅, {t})

MINCUTAGAT(S, C, X, T)
    ▷ Input: connected set S, C ∩ X = ∅
    ▷ Output: ccps for S
 1  if ISCONNECTEDIMP(S, C, T)
 2     then emit (C, S \ C)
 3          T' ← ∅
 4     else T' ← S
 5  if |C| + 1 ≥ |S|
 6     then return
 7  X' ← X
 8  for v ∈ (N(C) \ X)
 9        do MINCUTAGAT(S, C ∪ {v}, X', T' ∪ {v})
10           X' ← X' ∪ {v}
```

Fig. 6. Pseudocode for MINCUTAGAT.

ccps have to be emitted only once. The latter constraint is ensured automatically because the start vertex t is always element of C so that $t \notin (S \setminus C)$ holds. For the first constraint, we introduce an exclusive filter set X that keeps track of vertices v added to C in different branches of recursion. So once a vertex v is added to X (line 10) in an ancestor invocation of MINCUTAGAT, it cannot be chosen as a neighbor again and is filtered out (line 8). If there is only one vertex left in S that is not already a member of C, we can stop our recursive descent, because otherwise C's complement would be empty in the next child invocation. We check for this condition in line 5 and exit if it is true.

For MINCUTAGAT, we introduce an improved connection test which we call ISCONNECTEDIMP (line 1). Instead of ensuring that all vertices in a complement $S \setminus (C \cup \{v\})$ are connected to each other, our novel test only ensures that the neighbors $\mathcal{N}(C \cup \{v\})$ are connected to each other within the complement. In other words, we test for a weaker condition that checks if all elements of $\mathcal{N}(C \cup \{v\})$ lie on a common path that does not contain any vertices of $C \cup \{v\}$. We can even further simplify this condition once we know that the complement $S \setminus C$ is already proven to be connected. In such cases, it is sufficient to check if all elements of $\mathcal{N}(\{v\}) \setminus C$ are connected within $S \setminus (C \cup \{v\})$. On average, ISCONNECTEDIMP is cheaper to execute than a common connection test. In the best case, it is in $O(1)$. The worst case is identical to the complexity of the common connection test, which is in $O(|S \setminus (C \cup \{v\})|)$. We explain ISCONNECTEDIMP in detail in Section 6.4. To decide if it is sufficient to check only the neighbors of v, we introduce the set T. If the current partition $(C, S \setminus C)$ is a ccp (line 1), then the connection test for the next child invocations needs to check only for the neighbors of the v which is added to C so that the next T is set to $T \leftarrow \emptyset \cup \{v\}$ (lines 3 and 9). Otherwise, all neighbors of $C \cup \{v\}$ need to be checked so that the next T is set to $T \leftarrow C \cup \{v\}$ (lines 4 and 9).

```
PARTITION_MinCutBranch(S)
    ▷ Input: a connected set S
    ▷ Output: P^sym_ccp(S)
1   t ← arbitrary vertex of S
2   MINCUTBRANCH(S, {t}, ∅, {t})
```

Fig. 7. Pseudocode for $\text{PARTITION}_{MinCutBranch}$.

3.5 Branch Partitioning

With MINCUTAGAT, we have shown how the naive partitioning approach can be improved by allowing for only one connection test. With branch partitioning, we explain how this remaining test can be made redundant.

3.5.1 Branch Partitioning—An Overview

This section presents our novel partitioning algorithm named branch partitioning, which we denote by MINCUTBRANCH. The partitioning algorithm is invoked by TDPGSUB to compute for a given connected vertex set S all possible partitions into two disjoint interconnected sets (S_1, S_2) that are ccps for S. The output of branch partitioning is a set $P^{sym}_{ccp}(S)$ so that symmetric ccps are emitted only once. In Figs. 7 and 8, we give the algorithm's pseudocode with $\text{PARTITION}_{MinCutBranch}$ and MINCUTBRANCH. We call the instantiated generic memoization variant TDMINCUTBRANCH.

The algorithm's approach is to recursively enlarge a set C by members of its neighborhood $\mathcal{N}(C)$, starting with a single vertex $t \in S$. This way, we ensure that at every instance of the algorithm's execution C is *c*onnected. If at some point of enlarging C its complement $S \setminus C$ in S is connected as well, the algorithm has found a ccp for S. Besides, the connectivity of the C's complement branch partitioning has to meet some more constraints before emitting a ccp: 1) Symmetric ccps are emitted once, 2) the emission of duplicates has to be avoided, and 3) all ccps for S have to be computed as long as they comply with constraint (1).

Constraint (1) is ensured because the start vertex t—arbitrarily chosen during the initialization of the partitioning algorithm in line 1 of $\text{PARTITION}_{MinCutBranch}$—is always contained in C and, therefore, can never be part of its complement. For the second constraint, the algorithm uses a filter set X of neighbors to e*x*clude from processing. After every recursive self-invocation of the algorithm, the neighbor $v \in \mathcal{N}(C)$ that was used to enlarge C is added to X. Later, we will see in detail how this works. For constraint (3), it is sufficient to ensure that all possible connected subsets of S are considered when enlarging C.

Checking for the connectivity of the complement set adds a linear overhead per test. Furthermore, there are certain scenarios, e.g., when star queries are considered, where constructing every possible connected subset C of S produces an exponential overhead because most of the complements $S \setminus C$ are not connected and the partitions $(C, S \setminus C)$ computed this way are not valid ccps. For branch partitioning, we propose a novel technique, which ensures that no partitions are generated that are not a ccp at the same time. As a positive side effect, the additional check for connectivity can be eliminated.

```
MINCUTBRANCH(S, C, X, L)
    ▷ Input: connected sets S, C ∧ C, X ⊂ S ∧ |L| = 1
    ▷ Output: ccps for S
 1  R ← ∅
 2  R_tmp ← ∅
 3  N_L ← ((N(L) ∩ S) \ C) \ X
 4  N_X ← ((N(L) ∩ S) \ C) ∩ X
 5  N_B ← (((N(C) ∩ S) \ C) \ N_L) \ X
 6  while N_L ≠ ∅ ∨ N_X ≠ ∅ ∨ N_B ∩ R_tmp ≠ ∅
 7        do if (N_B ∪ N_L) ∩ R_tmp ≠ ∅      ▷ case (1)
 8              then v ← element of ((N_B ∪ N_L) ∩ R_tmp)
 9                   MINCUTBRANCH(
                                  S, C ∪ {v}, X', {v})
10                   N_L ← N_L \ {v}
11                   N_B ← N_B \ {v}
12              else X' ← X
13                   if N_L ≠ ∅                ▷ case (2)
14                     then v ← element of N_L
15                          R_tmp ← MINCUTBRANCH(
                                  S, C ∪ {v}, X', {v})
16                          N_L ← N_L \ {v}
                                              ▷ case (3)
17                     else v ← element of N_X
18                          R_tmp ← REACHABLE(
                                  S, C ∪ {v}, {v})
19                   N_X ← N_X \ R_tmp
20                   if R_tmp ∩ X ≠ ∅
21                     then N_X ← N_X ∪ (N_L \ R_tmp)
22                          N_L ← N_L ∩ R_tmp
23                          N_B ← N_B ∩ R_tmp
24                   if (S \ R_tmp) ∩ X ≠ ∅
25                     then N_L ← N_L \ R_tmp
26                          N_B ← N_B \ R_tmp
27                     else emit (S \ R_tmp, R_tmp)
28                   R ← R ∪ R_tmp
29             X' ← X' ∪ {v}
30  return R ∪ L
```

Fig. 8. Pseudocode for MINCUTBRANCH.

Before we explain our technique, we have to make some observations. From the recursive process of enlarging C, we know that the number of members in C must increase by one in every iteration. Furthermore, if a partition $(C, S \setminus C)$ is not a ccp for S, then $S \setminus C$ consists of $k \geq 2$ connected subsets $D_1, D_2, \ldots, D_k \subset (S \setminus C)$ that are disjoint and not connected to each other. Hence, those subsets $D_1, D_2, \ldots, D_k$ can only be adjacent to C. Let $v_1, v_2, \ldots, v_l$ be all the members of C's neighborhood $\mathcal{N}(C)$. Then every D_x with $1 \leq x \leq k$ must contain at least one such v_y where $1 \leq y \leq l$ and $k \leq l$ holds. The first ccp after enlarging C by members of $S \setminus C$ would be generated when all subsets D_x with $1 \leq x \leq k$ but one are joined to C.

Having made these observations, we are ready to explain our basic idea. The key principle is to exploit information about how $S \setminus C$ is connected from all of MINCUTLAZY's child invocations. Therefore, we introduce a new input parameter L and a result set R. The one-element set L contains the *l*ast vertex v that was added to C through the parent invocation. The result set R of a child invocation contains the maximally enlarged and connected set D_x such that $L \subseteq R$ holds. Note that the concept of R as MINCUTBRANCH's return value is different from the partitions which branch partitioning has to emit. We compute R by joining the result sets R_{tmp} from the child invocations with L. But we have to be careful to include only those R_{tmp} that are adjacent to L. Hence, we need to distinguish between $\mathcal{N}(L)$ and $(\mathcal{N}(C) \setminus \mathcal{N}(L))$: only those R_{tmp} can be joined to R where $\mathcal{N}(L) \cap R_{tmp} \neq \emptyset$ holds.

To make use of the connected sets R_{tmp} that are adjacent to v, we postpone the emission of ccps toward the end. Instead of enlarging C with all but one R_{tmp} when the complement $S \setminus C$ is not connected, we introduce an optimization which simply emits $(S \setminus R_{tmp}, R_{tmp})$ right away. Note that if $S \setminus C$ is connected, there exists only one R_{tmp} with $R_{tmp} = S \setminus C$ and $(S \setminus R_{tmp}, R_{tmp}) = (C, S \setminus C)$ holds. We have said that due to constraint (3), all connected subsets of S have to be considered as values for the set C. Through the optimization certain connected sets $S \setminus R_{tmp}$ are skipped. Because we avoid only those $S \setminus R_{tmp}$ where the complement $S \setminus (S \setminus R_{tmp}) = R_{tmp}$ is not a connected set, our optimization is still sufficient to meet constraint (3).

3.5.2 The Algorithm in Detail

In the following, we take a closer look at the pseudocode given in Figs. 7 and 8. $\text{PARTITION}_{MinCutBranch}$ calls MINCUTBRANCH the first time with $C = L = \{t\}$, where t is an arbitrary vertex. This ensures constraint (1) because the complement R_{tmp} cannot contain t at any instance of MINCUTBRANCH's execution. In lines 1 and 2, the result sets R and R_{tmp} are initialized.

When processing the neighbors of C, the primary interest lies on the neighbors of the recently added vertex $v \in L$ because they are important for the computation of the return value. Therefore, in line 3 the set N_L is introduced to store all the neighbors that certainly need to be processed, i.e., all neighbors of L that are not in X. The other neighbors of C which, at the same time, are not neighbors of L, are explored only if they belong to the result set R_{tmp} of one of the child invocations called with a neighbor of L. We store the neighbors of this category in the set N_B that holds all neighbors of C *b*ut not those that are in $\mathcal{N}(L)$ and, additionally, are not in X (line 5). Special care has to be taken before processing neighbors of L that are also elements of X, whereas the set X holds former neighbors that have been processed in an ancestor invocation. Now only those neighbors of L that are also element of X and are not contained in one of the result sets R_{tmp} need to be processed. We compute those candidates in line 4 and store them into N_X. Whether the other neighbors that are contained by the last two sets N_B and N_X are processed or not is decided dynamically during the loop in lines 6 to 29.

The loop (lines 6 to 29) consists of three cases. To understand these cases, we have to learn about the additional requirement that exists due to our duplicate avoidance technique. As already mentioned, we use the filter set X to exclude its members from being processed as a new L in a child invocation of MINCUTBRANCH. Moreover, if a complement $S \setminus R_{tmp}$ is not disjoint with X, then $(S \setminus R_{tmp}, R_{tmp})$ is a duplicate and has already been emitted. For explaining this fact, we denote by v_{old} a

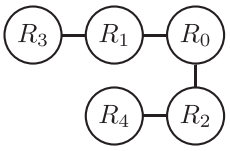

Fig. 9. Chain query graph of five relations.

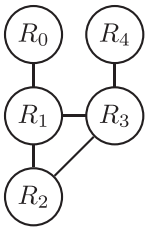

Fig. 10. Cyclic query graph of five relations.

member of $S \setminus (R_{tmp} \cap X)$. We know that $v_{old} \in \mathcal{N}(C)$ must hold, because as v_{old} is a member of X this implies that v_{old} was processed as a v in an ancestor invocation of MINCUTBRANCH as a neighbor of a C_{old}. As we will see later, v_{old} must be connected to v within $S \setminus C_{old}$ with $C_{old} \subset C$. Hence, a recursive descent started from a child invocation with a $C = C_{old} \cup \{v_{old}\}$ and an $L = \{v_{old}\}$ must have returned at one point with the same R_{tmp} as our current value. Therefore, the partition $(S \setminus R_{tmp}, R_{tmp})$ has already been emitted. We implement the test for duplicates in line 24 and emit the ccp in line 27.

Let us now consider the chain query of Fig. 9. We choose R_0 as the initial C. In the root invocation of MINCUTBRANCH, we first process R_1. When the child invocation returns, R_{tmp} equals $\{R_1, R_3\}$. Before processing R_2 as the second neighbor, we add R_0 to X'. In the next child invocation of MINCUTBRANCH with $C = \{R_0, R_2\}$, $L = \{R_2\}$, and $X = \{R_0\}$, a further recursive call with $C = \{R_0, R_2, R_4\}$, $L = \{R_4\}$, and $X = \{R_0\}$ would return a $R = \{R_4\}$. But instead of emitting the ccp $(\{R_4\}, \{R_0, R_1, R_2, R_3\})$, we would falsely assume that it is a duplicate because $(S \setminus R_{tmp}) \cap X \neq \emptyset$ holds. To solve this problem, X' needs to be reset to X once a new neighbor v is chosen that is not part of R yet (line 12).

As a consequence, we specify the processing order of the three sets N_L, N_B, and N_X dynamically and define three cases: case (1) is checked in line 7; It is true if a child invocation has started with a $v_x \in N_L$ (case (2)), or a $v_x \in N_X$ (case (3)) and a $v_y \in N_L$ or $v_y \in N_B$ is part of the returned R_{tmp}. Since the next invocation which we start with $L = \{v_y\}$ must return the same R_{tmp}, we do not have to save its return value and have no partition to emit, since it is already emitted. Note that the child invocation's excluded filter set is set to X', which in turn we must have set or reset to our own X in line 12 by processing case (2) or (3) before. After processing v_y, we delete it from its originating set, which is either N_L (line 10) or N_B (line 11).

Lines 13 to 16 cover case (2). If the condition of case (1) is not valid and there are elements of N_L left, we have to consider case (2). This means that N_L is not empty and either R_{tmp} is empty and no neighbor has been processed yet, or no other $v \in N_L$ is part of the current R_{tmp}. As explained for our duplicate avoidance technique, we have to set or reset the new input parameter X' to our current input parameter X. Because this also needs to be done for case (3), we move this task to line 12. Once the child invocation returns, we save the result in R_{tmp}. Note that $R_{tmp} \cap R = \emptyset$ holds. Later in line 28, R_{tmp} is joined with R. Having processed the current v, it is subtracted from N_L in line 16.

Case (3) ensures that all those neighbors $v \in N_X$ are processed that are not part of any returned R_{tmp}. A child invocation which is started with such a $L = \{v\}$ could not emit any further ccps because of the condition in line 24. As we only have to compute R_{tmp}, we use REACHABLE (Section 3.5.4). Because it is constructed solely for this task, it is a simpler and therefore more efficient method. By line 19, we avoid further unnecessary calls to REACHABLE. Note that also the results of case (2) are used to minimize N_X.

Lines 20 to 26 will be explained in Section 3.5.3. Before we return the call, we join L to the final result set R. In [5], two examples explain how MINCUTBRANCH works. Furthermore, [5] gives a complexity analysis.

3.5.3 Two Optimization Techniques

The lines 20 to 26 specify two optimization techniques that are not a requirement for the branch partitioning algorithm. The first technique considers cases where R_{tmp} contains elements of X. In this case, all other invocations of MINCUTBRANCH and their child invocations with neighbors of C that are disjoint from R_{tmp} cannot emit any partitions because the R'_{tmp} that they produce must be disjoint with R_{tmp} so that $S \setminus R'_{tmp}$ cannot be disjoint with X (line 24) anymore. But as we need to ensure that R is correctly computed, we have to add to N_X (line 20) those neighbors for which we want to avoid unnecessary calls to MINCUTBRANCH.

The second optimization technique avoids exploring all the other neighbors of C which are also elements of R_{tmp} if the complement $S \setminus R_{tmp}$ is not disjoint with X. As already mentioned, if these neighbors were not subtracted from N_L and N_B, they would be processed in the next iterations of the loop, and the condition of line 8 would qualify. Hence, all resulting child invocations of MINCUTBRANCH in line 9 cannot be avoided, although they would not emit any ccps.

Our experiments have shown that the two optimization techniques help to improve worst case behavior by 15 percent. The average impact on the runtime is less than 1 percent.

3.5.4 Exploring Restricted Neighbors

Finally, we explain REACHABLE. The psuedocode is given in Fig. 11. As already mentioned, it is its aim to return the maximally enlarged and connected set adjacent to L. In line 1 of REACHABLE, the one element result set R is initialized with L. Enlarging R starts with the set of neighbors of L that are disjoint to C and lie in S. During the while loop in lines 3 to 5, all the neighbors of the neighbors from the previous iteration of the loop that are disjoint with C are added to R. The loop is exited as soon as there is no vertex left to be added.

```
REACHABLE(S, C, L)
   ▷ Input: a connected set S, C ⊆ S, L ⊆ C, |L| = 1
   ▷ Output: connected set R adjacent to C
1  R ← L
2  N ← (𝒩(L) ∩ S) \ C
3  while N ≠ ∅
4        do R ← R ∪ N
5           N ← ((𝒩(N) ∩ S) \ C)
6  return R
```

Fig. 11. Pseudocode for REACHABLE.

4 Evaluation

This section summarizes our experimental findings. We start by briefly describing our setup. Then we give an organizational overview (Section 4.2) and present our results (Section 4.3).

4.1 Experimental Setup

For all plan generators, no matter whether they work top-down or bottom-up, a shared optimizer infrastructure was established. It contains the common functions to instantiate, fill, and lookup the memotable, initialize and use plan classes, estimate cardinalities, calculate costs, and compare plans. Thus, the different plan generators differ only in those parts of the code responsible for enumerating ccps. Since, due to the fact that we ignore pruning, the cost calculation is immaterial for our investigation, we simply use C_{out}. It sums up the cardinalities of the intermediate results.

We store the precalculated ancestors, descendants required by MINCUTLAZY and neighbors of a vertex in an array of size $|V|$. To generate our workload, we have implemented a generic query graph generator. In a first step, it generates chain, star, cycle, and clique queries, as well as random acyclic and cyclic graphs. For the latter, edges are randomly added by selecting two relation's indices using uniformly distributed random numbers. In a second step, cardinalities and selectivities are attached using a random generator with a Gaussian distribution. Since we leave out pruning, these numbers do not influence the search space of the plan generators.

4.2 Organizational Overview

In our empirical analysis, we compare basic memoization, denoted by MEMOIZATIONBASIC, based on naive partitioning (Section 2.2.2) and TDMINCUTLAZY (Section 3.1) with the novel memoization variants: TDMINCUTLAZYIMP (Section 3.2.1), TDMINCUTAGAT (Section 3.4) and TDMINCUTBRANCH (Section 3.5). To put all five top-down join enumeration algorithms in perspective, we include the results of Moerkotte's and Neumann's DPCCP [3] as the state of the art in bottom-up join enumeration via dynamic programming.

For our experiments, we measured the execution time of the six different plan generators on the same workload. To minimize measurement errors, we computed the average for every algorithm run for a given input. For fixed query shapes that are chains, stars, cycles, and cliques, and for random acyclic graphs, we give the number of vertices on the abscissa and the execution time in log scale on the ordinate. We draw lines to connect the averaged execution times.

Since for randomly generated cyclic queries the algorithms' performance results deviate significantly for the same number of vertices, we show the results for different numbers of vertices separately. At the abscissa, we choose to display the number of edges and again the execution time in log scale on the ordinate. We do not present the exact results which still can deviate strongly, but results smoothed by Bezier curves.

As we will see, apart from some minor exceptions, Moerkotte's and Neumann's DPCCP is the algorithm which performs best. Therefore, it is interesting to evaluate the results in terms of the quotient of the algorithm's execution time and the execution time of DPCCP. For the following discussion, we refer to that quotient as the normed time.

We include only those query graphs in our evaluation that all plan generators could process in less than 100 seconds. Our workload consists of 25.500 query graphs. The number of vertices and edges for our random cyclic queries are uniformly distributed. We conducted all our experiments on an Intel Pentium D with 3.4 GHz, 2 Mbyte second-level cache and 3 Gbyte of RAM that runs openSUSE 11.0. We used the Intel C++ compiler with the O3 compiler option set.

4.3 Experiments

This section summarizes our experimental findings. We start with an evaluation of acyclic query graphs and present the results for cyclic graphs later. An evaluation of partitioning costs can be found in [5].

4.3.1 Acyclic Query Graphs

We give the results for chain queries in Fig. 12, for star queries in Fig. 13, and for random acyclic graphs that are neither chain nor star queries in Fig. 14. For all different acyclic graphs, the normed runtimes of MEMOIZATIONBASIC and TDMINCUTAGAT show an exponential increase, although the effect is much weaker for TDMINCUTAGAT. With a maximal normed runtime of almost 47,000 (Table 3) for chain queries, MEMOIZATIONBASIC is a terrible choice for acyclic graphs. DeHaan's and Tompa's TDMINCUTLAZY performs on a mediocre level with a maximal factor of 3.5 for star queries (Table 3). Since TDMINCUTLAZYIMP needs much fewer biconnection tree buildings when acyclic graphs are considered as our derived formulas (Section 3.2.2) show, it is distinctly faster than TDMINCUTLAZY (maximal factor of 1.8 for star queries). Except for star queries, where we have more cache misses with an increasing number of vertices, TDMINCUTBRANCH performs even better than the state of the art in dynamic programming DPCCP. The lowest factor we could determine for TDMINCUTBRANCH was at 0.66 for random acyclic queries.

4.3.2 Cyclic Query Graphs

Cycle and clique queries belong to the same group of cyclic graphs, but in terms of the number of ccps, they belong to two opposite sides of the spectrum. Cycles have the lowest number of edges that is possible for cyclic graphs, removing one edge would result in a chain query. We present their results in Fig. 15. Cliques have the maximal number of edges possible. The results for clique queries can be found

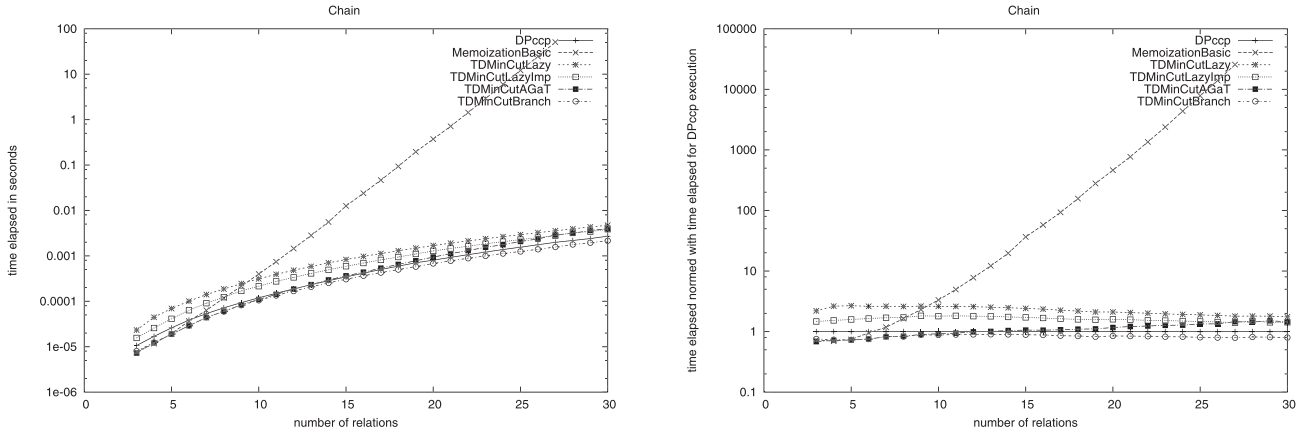

Fig. 12. Absolute and normed runtime results for chain queries.

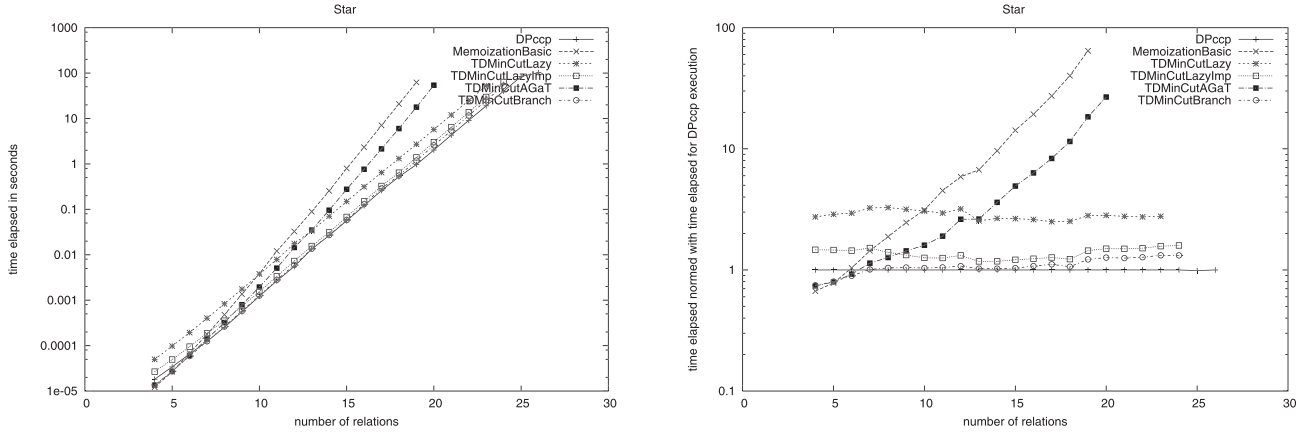

Fig. 13. Absolute and normed runtime results for star queries.

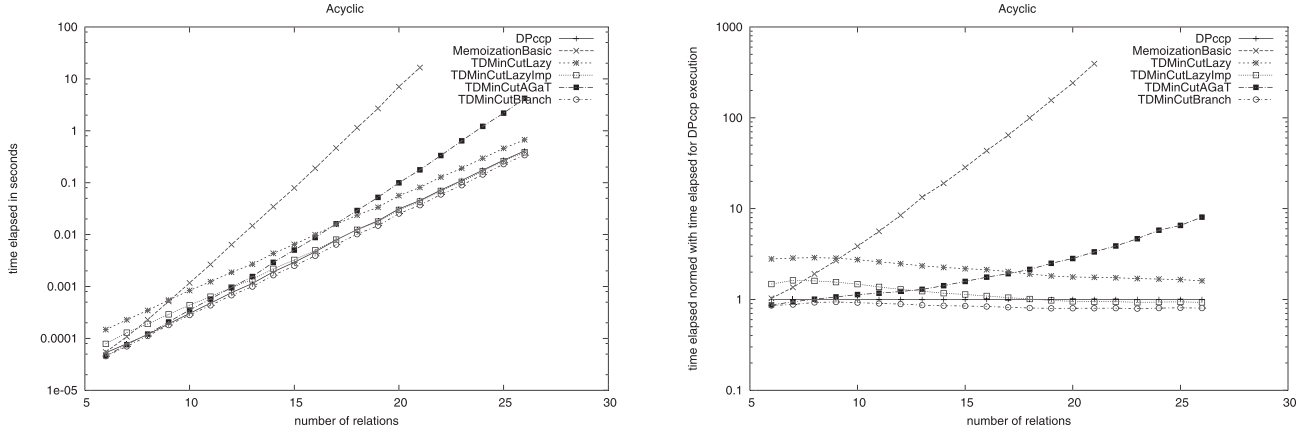

Fig. 14. Absolute and normed runtime results for random acyclic queries that are neither chain nor star queries.

TABLE 3
Minimum, Maximum, and Average of the Normalized Runtimes for Chain, Star, and Random Acyclic Queries

Algorithm	Chain			Star			Acyclic		
	min	max	avg	min	max	avg	min	max	avg
DPccp	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
MemoizationBasic	1.06	46826.65	1569.17	0.89	76.93	11.88	0.94	5646.15	150.87
TDMinCutLazy	1.69	2.85	2.27	2.19	3.50	2.93	1.48	3.22	2.23
TDMinCutLazyImp	1.22	2.35	1.64	0.90	1.77	1.34	0.82	2.35	1.20
TDMinCutAGaT	0.65	1.66	1.11	0.76	29.68	4.02	0.62	9.77	1.88
TDMinCutBranch	0.74	0.98	0.85	0.77	1.30	1.04	0.66	1.03	0.85

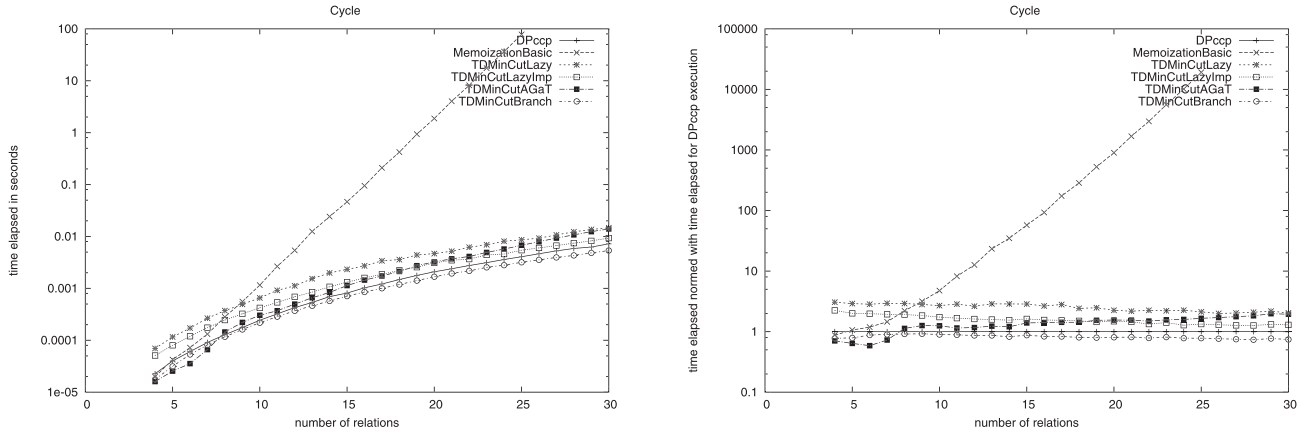

Fig. 15. Absolute and normed runtime results for cycle queries.

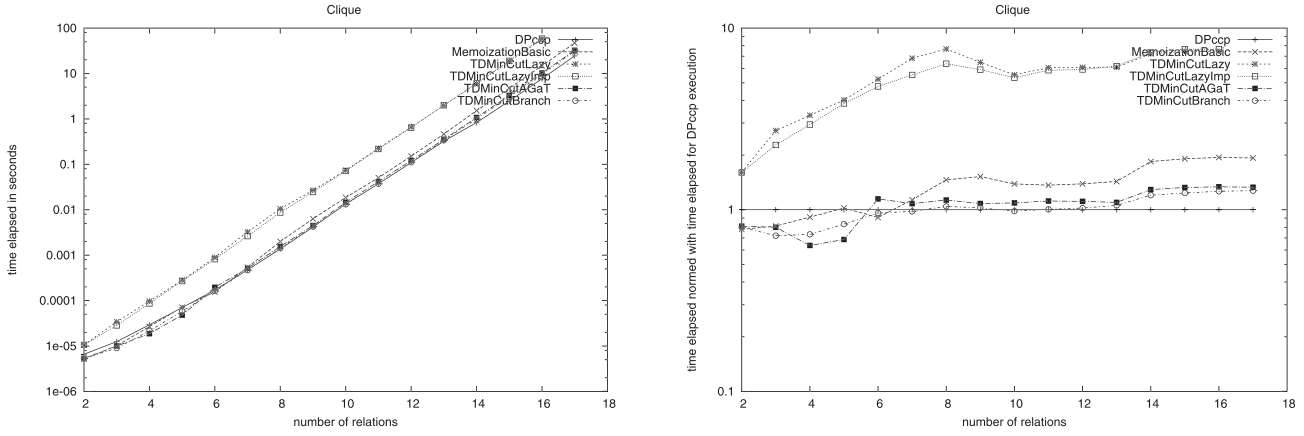

Fig. 16. Absolute and normed runtime results for clique queries.

TABLE 4
Minimum, Maximum, and Average of the Normalized Runtimes for Cycle, Clique, and Random Cyclic Queries

Algorithm	Cycle			Clique			Cyclic		
	min	max	avg	min	max	avg	min	max	avg
DPccp	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
MemoizationBasic	0.77	22951.02	3630.71	0.49	1.96	1.46	1.10	200.93	5.76
TDMinCutLazy	1.37	3.95	2.59	2.40	8.48	6.35	2.13	8.00	5.63
TDMinCutLazyImp	1.07	2.33	1.55	3.23	7.75	5.82	1.24	7.41	5.43
TDMinCutAGaT	0.63	1.79	1.38	0.72	1.41	1.15	0.80	5.58	1.30
TDMinCutBranch	0.74	0.98	0.84	0.79	1.29	1.06	0.78	1.47	1.13

in Fig. 16. Table 4 summarizes the normed runtimes for all evaluated algorithms.

As expected, the results for cycle queries resemble those for chain queries more than to the results for clique queries. From Table 4, we can see that except for TDMINCUTLAZYIMP and TDMINCUTBRANCH, the averaged normed runtime has increased, compared to the results for chain queries. Again, TDMINCUTBRANCH is the fastest algorithm.

When looking at cliques, the picture changes completely. Now MEMOIZATIONBASIC dominates TDMINCUTLAZY and TDMINCUTLAZYIMP. TDMINCUTAGAT performs better than MEMOIZATIONBASIC, since it relies only on one connection test per emitted ccp. This time, TDMINCUTBRANCH performs second best and is only dominated by DPCCP.

When we consider random cyclic graphs, we can trace the performance shift between cycle and clique queries. We display our results for 8 and 16 vertices in Figs. 17 and 18. The trends of the normed runtimes of TDMINCUTAGAT and MEMOIZATIONBASIC compared to the trends of TDMINCUTLAZY and TDMINCUTLAZYIMP, respectively, are oppositional. Whereas the normed runtime of TDMINCUTAGAT and MEMOIZATIONBASIC is decreasing with an increasing number of edges for a fixed number of vertices, TDMINCUTLAZYIMP's normed runtime is increasing. For a fixed number of vertices and a relatively small number of edges, the differences between TDMINCUTLAZY and TDMINCUTLAZYIMP are distinctive, but for a number of edges lying in the median span, the differences become indistinctive. Independent of the number of edges or vertices, TDMINCUTLAZYIMP dominates TDMINCUTLAZY.

It is obvious that TDMINCUTAGAT dominates MEMOIZATIONBASIC and TDMINCUTLAZY and its

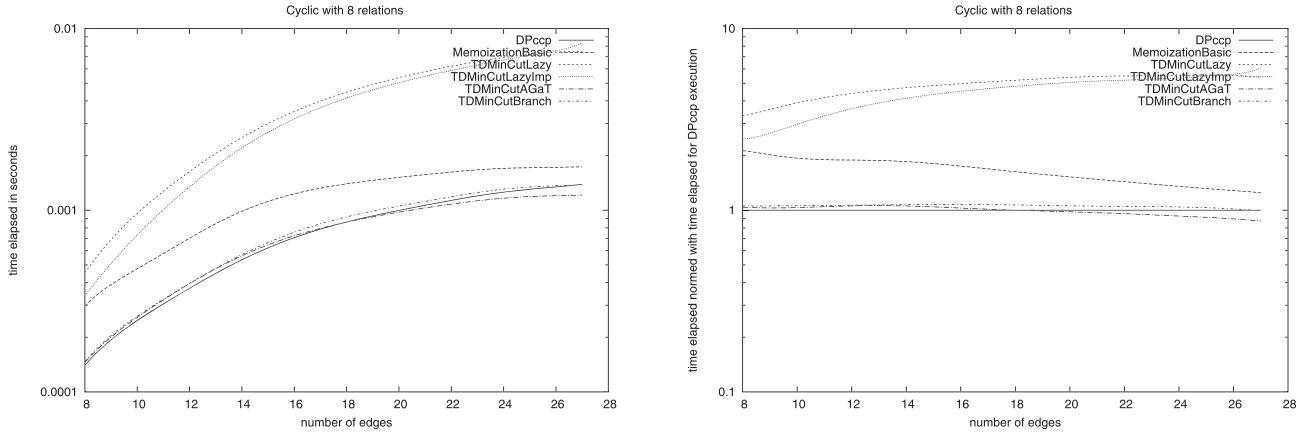

Fig. 17. Absolute and normed runtime results for cyclic queries with 8 vertices.

improved version with some minor exceptions of TDMINCUTLAZYIMP at the beginning of the spectrum.

Once more, all top-down join enumeration algorithms are outperformed by TDMINCUTBRANCH. With an averaged normed runtime of 1.1 for random cyclic queries, it is almost as efficient as DPCCP, which performs best. We can contrast the worst relative factor of 1.47 with a factor of $\frac{1}{0.78} = 1.28$ in the best case, being faster than DPCCP, although no branch-and-bound pruning is put in place.

5 Conclusion

With TDMINCUTLAZYIMP, we have improved upon existing work of DeHaan and Tompa that is especially successful for acyclic query graphs. Through MINCUTAGAT, we devised advancements of the naive generate and test paradigm. Finally, we presented an entirely partitioning approach for top-down join enumeration, which has two advantages over the existing work:

- It performs better.
- It is easier to implement.

The latter is due to the fact that it does not need complex data structures like the biconnection tree. Instead, it only relies on set operations, which can be implemented easily and efficiently using bit vectors.

Furthermore, the new algorithm exhibits about the same performance as the best-known bottom-up algorithm. Importantly, it does so *without* relying on pruning. Thus, as soon as the query is amenable for branch-and-bound pruning, our new top-down algorithm will be superior to the best bottom-up algorithm.

There are two major challenges for future work. The first is to extend MINCUTLAZY to hypergraphs. This is important, since not all queries have an equivalent query graph. Some need hypergraphs.

Currently, the conditions under which pruning is effective, that is for which kind of query which degree of pruning can be achieved, is unknown. Thus, the second, unequally more challenging problem for future work is to determine these conditions.

6 Important Details

6.1 Biconnection Tree Building

Unfortunately, DeHaan and Tompa omitted some important implementation details about how to build a biconnection tree efficiently. This section fills some gaps. First the biconnection tree construction (Section 6.1.1) is explained, and then implementation alternatives for precomputing the ancestors and descendants (Section 6.1.2) are discussed through a complexity analysis.

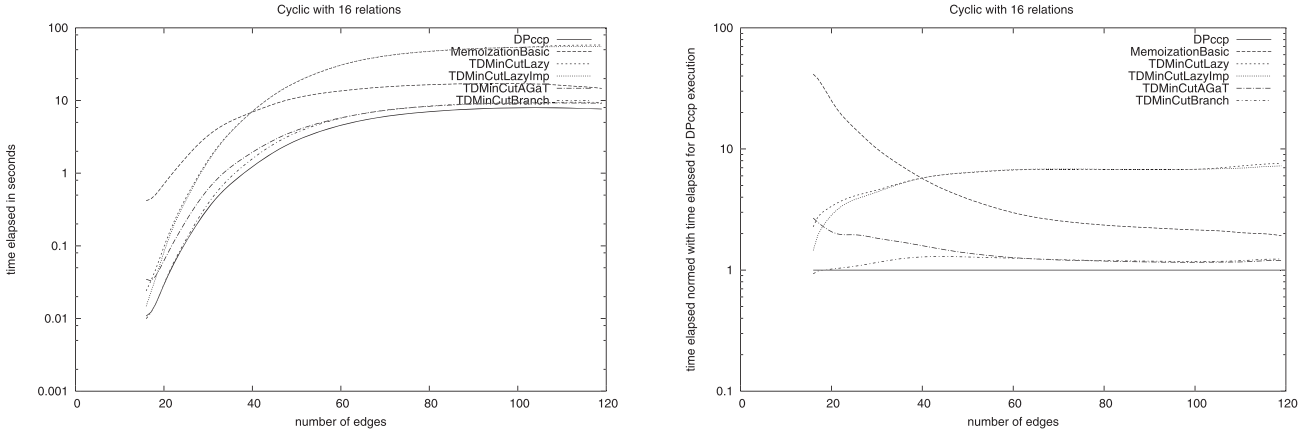

Fig. 18. Absolute and normed runtime results for cyclic queries with 16 vertices.

```
BUILDBCT(G, t)
    ▷ Input: connected query graph G = (V, E), t ∈ V
    ▷ Output: biconnection tree T for G rooted at t
 1  for each vertex v ∈ V
 2      do color[v] ← WHITE
 3         df[v] ← |V| + 1
 4         low[v] ← |V| + 1
 5         π[v] ← NIL
 6  count ← 0
 7  declare stack of edges est
 8  declare stack of vertex nodes vst
 9  declare stack of set nodes sst
10  BUILDBCTSUB(t)
11  return (V_sn ∪ V, E_tree, t)

BUILDBCTSUB(v)
    ▷ Input: vertex v ∈ V
 1  color[v] ← GRAY
 2  count ← count + 1
 3  df[v] ← count
 4  low[v] ← df[v]
 5  for all w ∈ N(v)
 6      do if color[w] = WHITE
 7            then PUSH(est, (v, w))
 8                 π[w] ← v
 9                 BUILDBCTSUB(w)
10                 PUSH(vst, v)
11                 if low[w] ≥ df[v]
12                   then bcc ← ∅
13                        repeat (e1, e2) ← POP(est)
14                               bcc ← bcc ∪ {e1} ∪ {e2}
15                        until (e1, e2) = (v, w)
16                        V_sn ← V_sn ∪ {s_bcc}
17                        while TOP(vst) ∈ bcc
18                            do c ← POP(vst)
19                               E_tree ← E_tree ∪ {(s_bcc, c)}
20                        PUSH(sst, s_bcc)
21                 low[v] ← MIN(low[v], low[w])
22            else if w ≠ π[v]
23                   then PUSH(est, (v, w))
24                        low[v] ← MIN(low[v], df[w])
25  color[v] ← BLACK
26  while v ∈ TOP(sst)
27      do s_bcc ← POP(sst)
28         E_tree ← E_tree ∪ {(s_bcc, v)}
```

Fig. 19. Pseudocode for BUILDBCT.

6.1.1 Bionnection Tree Construction

The pseudocode for constructing a biconnection tree is given in Fig. 19 as a modification of [9] [10]. The variables are initialized in BUILDBCT, and the recursive search and building procedure is implemented within BUILDBCTSUB. We presume that the details of depth-first search and preorder numbering are known and omit any further explanation.

The stack of graph edges est is used to distinguish between the tree edges and back edges that are not captured. Tree edges are those that lead to all the accessed vertices during the discovery of vertices. Back edges are the rest of the graph edges that would close the cycles to already visited vertices. We denote the set of tree edges with E_t and the set of back edges with E_b. It holds that $E = E_t \cup E_b$ with $E_b \cap E_t = \emptyset$. For the recognition of cycles, we need an additional field low. For its values, it holds:

$$low[v] = Min(\{df[v]\} \cup \{df[w] \mid w \in D\}),$$

where df is the preorder number and D is defined as

$$D = \{w \mid \exists x (x, w) \in E_b, w \overset{*}{\rightarrow} v \overset{*}{\rightarrow} x\}.$$

In other words, the set D includes all vertices w with a back edge $(x, w) \in E_b$, and v is a descendant of w and x a descendant of v in the directed spanning tree $S(V, E_t)$. Note that the vertex v is a descendant of w if $df[w] < df[v]$ holds. Hence, x is also a descendant of w, and it holds that $df[w] < df[v] < df[x]$. The calculation of $low[v]$ can be embedded into the depth-first search procedure if the formula is rewritten in terms of values of $low[s]$ at the direct children $s \in C(v)$ of v in $S(V, E_t)$ and of the preorder numbers of the vertices connected to v via back edges.

$$low[v] = Min(\{df[v]\} \cup \{low[s] | s \in C(v)\} \cup \{df[w] | w \in D\}).$$

The set $C(v)$ is defined as

$$C(v) = \{s | s \in \mathcal{N}(v) \wedge df[v] < df[s]\}.$$

Due to the recursive iteration, the final value of $low[v]$ is not known before the list of vertices adjacent to v is fully processed. If a vertex v is an articulation vertex and the entry point of the biconnection $G^{BCC} = (V_i, E_i)$ where $v, s \in V_i$ holds, it is recognized by $low[s] \geq df[v]$. This is also true for a root that is not an articulation vertex because it has just one set node as a child.

In line 4 of BUILDBCTSUB, the value of $low[v]$ is initialized. Since during processing $low[v] \leq df[v]$ holds, its preorder number is chosen. As the first part of the previous rewritten formula, the value of $low[v]$ is adjusted to the minimum value between $low[w]$ from the son w of v and itself in line 21. The second part of the definition is implemented in line 24. The check of line 22 ensures that $(v, w) \in E_b$ is really a back edge and not just a tree edge.

Knowing the value $low[w]$ of v's descendant w in $S(V, E_t)$, it could be determined whether v is an articulation vertex and as a vertex node, implicitly, the father of a set node $s_{\{v,w,\ldots\}}$ (line 11). If so, that set node $s_{\{v,w,\ldots\}}$ has to be constructed and added to the set of set nodes V_{sn} (line 16). Taking all edges from est (line 13) until the edge (v, w) from which w was accessed (line 15) corresponds to enumerating the recognized biconnected component. To complete the set node, all its child vertex nodes have to be attached by adding an edge to the edge set E_{tree} of the biconnection tree (line 19). They can be taken from the stack of vertex nodes vst (line 18).

If v is an articulation vertex, then all its set nodes are attached in line 28. The set nodes, which belong to v and all

contain v, have to be on top of sst. In case that v is not an articulation vertex, there will be no items on top of sst containing v. This follows from the observation that the set node corresponding to the biconnection is not constructed until the recursive call (line 9) returns to its articulation vertex or root t in $S(V, E_t)$.

6.1.2 Complexity of Biconnection Tree Construction

Now, we pay attention to the complexity of constructing a biconnection tree. Recognizing the biconnected components of a graph has a complexity of $O(|E|)$ [10]. But BUILDBCT contains three more loops, which we have to examine additionally.

The loop in line 13 iterates over the edges that are part of a biconnection, not including their back edges $b_i \in E_b$. Since a tree edge $e_i \in E_t$ is never pushed twice on est within all invocations of BUILDBCTSUB, this loop increases complexity by $|E_t|$, where $E_t = E \setminus E_b$. Note that $|E_t| = |V| - 1$ holds.

The second loop (18) iterates over vertices in vst and attaches each vertex to its corresponding set node. That adds $|V| - 1$ to the complexity. Note that the -1 comes from the root t, which does not have to be attached to a set node.

The third loop (26) runs over all set nodes that need to be linked to their corresponding articulation vertex or root t, respectively. If A is the set of all articulation vertices, then in all calls to BUILDBCTSUB there will be $|A|$ set nodes added to sst if $t \in A$, or $|A| + 1$ set nodes otherwise. This increases complexity by at least $|A|$.

6.1.3 Computation of Ancestors and Descendants

DeHaan and Tompa suggest [4] to incorporate the computation of $\mathcal{D}_T(v)$ and $\mathcal{A}_T(v)$ for all $v \in V$ into the biconnection tree building without increasing its complexity, but do not explain how this can be done.

The children as direct descendants can be part of more than one biconnected component bcc. Therefore, in addition to bcc (12) in BUILDBCTSUB, the field $descendants$ is introduced preferably before line 5. After a new bcc is constructed (16), the $descendants$ should be updated with the descendants of those vertices which have been calculated in previous recursions. This means adding $O(|bcc| - 1)$ for each bcc and in total $O(|V| - 1)$.

At the same time, to all those vertices in the biconnected component the direct ancestor v can be assigned. Since the other ancestors, which are all articulation vertices on the path to the root t, are not known at this point, only the parent can be linked.

To calculate $\mathcal{A}_T(v)$, the indirections of the parent articulation vertices and their ancestors need to be followed until the root t is reached. In the worst case scenario, i.e., a chain query where one end is the root and the other end has $|V| - 1$ ancestors, this means that $|V| - 2$ indirections have to be followed for one computation of $\mathcal{A}_T(v)$.

The other alternative is to update not only the direct descendants but the whole set. That means in the worst case with a chain query $|V| - 1$ update operations for the opposite end of t and $(|V| - 1)!$ updates in total. It is cheaper not to follow DeHaan's and Tompa's suggestion and to postpone the calculation of the ancestors to the end of the tree construction. Knowing all direct descendants, the computation can be done top-down in $O(|V| - 1)$ by revisiting the root and all the articulation vertices. Going down this path enables us to postpone the update operation of the current descendants with the descendants of the children in BUILDBCTSUB, which saves us $O(|V| - 1)$ compared to incorporating it in the biconnection tree building.

6.1.4 An Alternative to Tree Construction

Looking at MINCUTLAZY raises the question whether the overhead of the biconnection tree building is necessary, since just the ancestors and descendants of a vertex v need to be computed. The only method which depends on a biconnection tree structure is ISUSABLE. For this reason, we suggest to reformulate the usability test and to use just the information of the vertex set V_i of a $G_i^{Bcc}(V_i, E_i)$ a vertex v belongs to. If a vertex is an articulation vertex, i.e., an inner vertex node of the biconnection tree, it must be part of more than one set node or biconnected component, respectively (Definition 2.4). Therefore, we map only V_i to a vertex v in case that v is not the parent vertex node p of the set node s_{V_i} in a biconnection tree. To optimize the usability test, we suggest for the mapping of v to V_i that the vertex set V_i is reduced by $\{p\}$.

Following this suggestion, we do not need the stacks sst and vst and the loop in line 18 of BUILDBCTSUB anymore, which saves us $O(|V| - 1)$.

We can conclude that in spite of our suggested improvements, the additional demands on the algorithm of [10] increase the complexity by $2 * |V| - 2 + |A|$ to an overall complexity of $|E| + 2 * |V| - 2 + |A|$.

6.2 Number of Biconnection Tree Buildings

The formulas we derive in the following hold if one of two strategies for choosing the root vertex for a new biconnection tree (or tree for short) is consistently applied: we either determine the root by appointing the vertex with the lowest index or the vertex with the highest index of the set.

6.2.1 Chain Queries

In case of TDMINCUTLAZY, there are as many tree buildings as there are calls to $\text{PARTITION}_{MinCutLazy}$ and as there are connected subsets S, $|S| > 1$ of V from the query graph $G = (V, E)$. We define k as the cardinality of a connected subset of V. Because we only account for subsets with more than one vertex, $2 \leq k \leq |V|$ holds. Since there are $|V| - k + 1$ different connected subsets of size k, the total number of subsets with more than one vertex is $\#t^{chain}(|V|) = \sum_{k=2}^{|V|}(|V| - k + 1) = \frac{|V|^2}{2} - \frac{|V|}{2}$.

We observe that only the tree for the whole set V can be reused. This is because in every possible case only one vertex is added to C so that $|\mathcal{D}_T(v)| = 1$ holds. Furthermore, the number of times the top-level tree for all vertices in V is reused depends on the choice of the root t. If the root is chosen from one end, we have the worst case, since there are only $|V| - 2$ different connected and proper subsets of V containing t. The best case we have if t is chosen as the middle vertex of the chain query. Then if $|V|$ is even, we have $2 * \sum_{i=2}^{\frac{1}{2}*|V|} i = \frac{|V|^2}{4} + \frac{|V|}{2} - 2$, and if $|V|$ is odd, we have $2 * \sum_{i=2}^{\frac{|V|-1}{2}} i = \frac{|V|^2}{4} + \frac{|V|}{2} - \frac{7}{4} = \lceil \frac{|V|^2}{4} + \frac{|V|}{2} - 2 \rceil$ connected and

TABLE 5
Emitted and Missing Partitions of the Proposed Optimistic Partitioning

level	S	T	$\mathcal{N}(S)$	S'
0	$\emptyset$	$\{R_0\}$	$\{R_1, R_2, R_3, R_4\}$	$\{R_1\}$ $\{R_2\}$
1	emitting $(\{R_2\}, \{R_0, R_1, R_3, R_4\})$			
1 0	$\{R_2\}$ $\emptyset$	$\{R_0, R_1\}$ $\{R_0\}$	$\{R_1, R_3\}$ $\{R_1, R_2, R_3, R_4\}$	$\{R_2, R_3\}$ $\{R_3\}$ $\{R_4\}$
1	emitting $(\{R_4\}, \{R_0, R_1, R_2, R_3\})$			
1	$\{R_4\}$	$\{R_3\}$	$(\{R_0, R_1, R_2, R_3\}$	-
missing $(\{R_0\}, \{R_1, R_2, R_3, R_4\})$ missing $(\{R_0, R_1\}, \{R_2, R_3, R_4\})$ missing $(\{R_0, R_1, R_2\}, \{R_3, R_4\})$				

proper subsets of V containing t. By subtracting our results for the number of times the tree can be reused from the number of connected subsets with more than one vertex, we get the final result. For the best case, we then have $\#t^{chain}_{imp}(|V|) = \lfloor \frac{(|V|-2)^2}{4} \rfloor + 1$, and for the worst case $\#t^{chain}_{imp}(|V|) = \frac{|V|^2-3*|V|}{2} + 2$ tree buildings.

6.2.2 Star Queries

For TDMINCUTLAZY, we have $\frac{|V|-1}{k-1}$ different connected subsets of size k and in total $\#t^{star}(|V|) = \sum_{k=2}^{|V|} \frac{|V|-1}{k-1} = 2^{|V|-1} - 1$ different connected subsets with more than one vertex. In the best case, the root t of the tree is chosen from the hub vertex of the star. Then we have $\#t^{star}_{imp}(|V|) = 1$. The worst case happens when the hub vertex has an index that would be the last or second last choice for the root of the trees among all other indices. We observe that we need one tree construction less if the hub's index is the third last choice for being the root and $i-2$ tree constructions less if it is the ith last choice. Therefore, for the worst case we have $\#t^{star}_{imp}(|V|) = \sum_{i=3}^{|V|}(i-2) + 1 = \frac{|V|^2-3*|V|}{2} + 2$ tree buildings.

6.2.3 Clique Queries

There are $\frac{|V|}{k}$ connected subgraphs with k vertices. For TDMINCUTLAZY, we need 2^{k-2} tree buildings per call to $\text{PARTITION}_{MinCutLazy}$. Since we need a tree only for graphs with more than one vertex, we have for the nonimproved version $\#t^{clique}(|V|) = \sum_{k=2}^{|V|} \frac{|V|}{k} * 2^{k-1} = \frac{1}{4} * 3^{|V|} - \frac{|V|}{2} - \frac{1}{4}$. We observe for $|V| > 2$ that for the improved version we reuse $2^{|V|} - |V|$ biconnection trees more than in the case of a clique query of $|V|-1$ vertices. We give the number of reuses by $\sum_{k=1}^{|V|-2}(2^k - k) = 2^{|V|-1} - \frac{|V|^2}{2} + \frac{3*|V|}{2} - 3$. The total number of tree buildings for TDMINCUTLAZYIMP is then the difference between the number of tree buildings of TDMINCUTLAZY and the number of reuses. We give it with $\#t^{clique}_{o}(|V|) = \frac{1}{4} * 3^{|V|} - 2^{|V|-1} + \frac{|V|^2}{2} - 2 * |V| + \frac{11}{4}$.

6.3 Optimistic Partitioning is Not Correct

As mentioned in Section 3.3, MINCUTOPTIMISTIC does not necessarily enumerate the complete set $P^{sym}_{ccp}(S)$. Therefore, the algorithm is not correct. We exemplify this by the query

```
ISCONNECTEDIMP(S, C, T)
    ▷ Input: C ⊂ S a connected set, T ⊆ C
    ▷ Output: if (S \ C) is connected TRUE, else FALSE
 1  N ← (N(T) ∩ S) \ C
 2  if |N| ≤ 1
 3      then return TRUE
 4  L ← ∅
 5  L' ← {n | n ∈ N}
 6  U ← N \ L'
 7  while L ≠ L' and U ≠ ∅
 8        do D ← L' \ L
 9           L ← L'
10           L' ← L' ∪ (N(D) ∩ S) \ C
11           U ← U \ L'
12  if U = ∅
13      then return TRUE
14      else return FALSE
```

Fig. 20. Pseudocode for ISCONNECTEDIMP.

graph G visualized in Fig. 10. When we give G as input to the algorithm (Fig. 5) and choose the start node $t = \{R_0\}$, it emits two partitions: $(\{R_2\}, \{R_0, R_1, R_3, R_4\})$ and $(\{R_4\}, \{R_0, R_1, R_2, R_3\})$. For this scenario, Table 5 lists the states of the algorithm during execution. As the reader can verify, there are five valid ccps, which means that the proposed algorithm misses three ccps for $\{R_0, R_1, R_2, R_3, R_4\}$ to enumerate.

6.4 An Improved Connection Test

With MINCUTAGAT, we have introduced a sophisticated generate-and-test-based partitioning algorithm for top-down join enumeration (Section 3.4). In this section, we explain the novel connection test on which MINCUTAGAT is based. The pseudocode is given in Fig. 20.

The purpose of ISCONNECTEDIMP is to check if the neighbors of T that are not in C lie on a common path containing only vertices of $S \setminus C$. In Line 1 of ISCONNECTEDIMP, we compute those neighbors of T as the set N. If N contains only one element, then $S \setminus C$ must be connected and the test returns with a positive result. As a starting point for the test, we chose an arbitrary vertex $n \in N$ (line 5) and assign it to L'. Throughout the test, the set U contains all the vertices of N that have not been reached by the generation of neighbors of $\{n\}$ yet. In the loop of Lines 7 to 11, the indirect neighborhood [3] of $\{n\}$ is computed by enlarging L' with a generation of neighbors of the previous L' in every iteration. Thereby, L holds the previous L' (line 9) and D holds all the new elements of L'. Note that except from the first iteration of the loop, $\mathcal{N}(L) \cap (S \setminus C) = D$ holds.

Because of $|D| < |L'|$, we compute the next generation of neighbors from D instead of L' (line 10), which is clearly more efficient. If we can reach new elements of N, we must subtract them from U (line 11). Once U equals the empty set, the loop is interrupted (line 7), and it is obvious that all $n \in N$ are reachable within $S \setminus C$ so that TRUE is returned (line 13). If, on the other hand, L' cannot be enlarged further so that $L = L'$ holds, we have computed all indirect neighbors of $\{n\}$ (within $S \setminus C$) and meet the loop's stop condition. In that case, we could not reach every element of N so that $N \setminus L' \neq \emptyset$ must hold, and we have to return

FALSE (line 14). Note that those vertices left in $U = N \setminus L'$ must belong to at least one different connected set that is only adjacent to C and not connected to L'.

ACKNOWLEDGMENTS

The authors thank the referees for their thorough feedback, Simone Seeger for her help preparing this manuscript, and Guy Lohman for the photographs.

REFERENCES

[1] Y.E. Ioannidis and Y.C. Kang, "Left-Deep versus Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization," *ACM SIGMOD Record,* vol. 20, no. 2, pp. 168-177, 1991.

[2] K. Ono and G.M. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization," *Proc. 16th Int'l Conf. Very Large Databases (VLDB),* pp. 314-325, 1990.

[3] G. Moerkotte and T. Neumann, "Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products," *Proc. 32nd Int'l Conf. Very Large Databases (VLDB),* pp. 930-941, 2006.

[4] D. DeHaan and F.W. Tompa, "Optimal Top-Down Join Enumeration," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD),* pp. 1098-1109, 2007.

[5] P. Fender and G. Moerkotte, "A New, Highly Efficient, and Easy to Implement Top-Down Join Enumeration Algorithm," *Proc. IEEE 27th Int'l Conf. Data Eng. (ICDE),* 2011.

[6] L.M. Haas, M.J. Carey, M. Livny, and A. Shukla, "Seeking the Truth About Ad Hoc Join Costs," *Very Large Database J.,* vol. 6, pp. 241-256, 1997.

[7] S. Baase and A.V. Gelder, *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 1999.

[8] B. Vance and D. Maier, "Rapid Bushy Join-Order Optimization with Cartesian Products," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '96),* pp. 35-46, 1996.

[9] D. DeHaan and F.W. Tompa, "Optimal Top-Down Join Enumeration (Extended Version)," technical report, Univ. of Waterloo, 2007.

[10] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

Pit Fender received the MSc degree (former diplom) from Dresden Technical University, Germany. Currently, he is working in the Database Research Group at Mannheim University, Germany. His main research interests include query optimization and query processing.

Guido Moerkotte studied computer science at the Universities of Dortmund, Massachusetts, and Karlsruhe University, from 1981 to 1987. The Karlsruhe University awarded him the diploma, doctorate, and postdoctoral lecture qualification, in 1987, 1989, and 1994, respectively. In 1994, he became an associate professor at the RWTH Aachen. Since 1996, he held a full professor position at Mannheim University, where he is a head of the Database Research Group. His research interests include databases and their applications, query optimization, and XML databases. He has (co)authored more than 100 publications and three books.