A New, Highly Efficient, and Easy To Implement Top-Down Join Enumeration Algorithm

Pit Fender #1, Guido Moerkotte #2

#Database Research Group, University of Mannheim
68131 Mannheim, Germany
1fender@informatik.uni-mannheim.de
3moerkotte@informatik.uni-mannheim.de

***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 non-clique 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.

I. 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. The process of choosing a suitable plan from all alternatives is known as query optimization. The basis of this choice are a cost model and statistics over the data. Essential for the costs of a plan is the execution order of join operations in its operator tree, since the runtime of plans with different join orders can vary by several orders of magnitude. 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 this challenge. This requires a little preparation.

For every subset S of relations that induces a `c`onnected `s`ub`g`raph (`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 I 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 Sec. IV-D). 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 $O(1)$. However, for cyclic query graphs the complexity increases and reaches a maximum for cliques, where it is $O(n^2)$ for n relations (see appendix).

[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.

TABLE I

		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

Fortunate Observation. The number of connected subgraphs (#csg) is far lower than the number of csg-cmp-pairs. This is very fortunate since (1) #csg is the number of times 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 [5].

Contribution. In this paper, 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 MINCUTLAZY, showing its $O(n^2)$ complexity for clique queries,
- propose branch partitioning as an entirely new and easy to implement algorithm for top-down join enumeration,
- show analytically that the complexity of branch partitioning is in $O(1)$ for acyclic graphs, cycle graphs and clique queries, and
- present an in-depth performance evaluation indicating that the new enumerating algorithm is nearly as efficient as DPCCP.

Besides its efficiency, our new algorithm has the great advantage that it is far easier to implement than the one developed by DeHaan and Tompa. Whereas they need to build and maintain a complex data structure called biconnection tree, 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: Sec. II recalls some preliminaries. Sec. III presents our new algorithm. Section IV presents our thorough performance evaluation. Sec. V concludes the paper. The appendix contains the complexity analysis of MINCUTLAZY.

II. PRELIMINARIES

Before we start with our discussions, we give some fundamentals. In the first subsection, 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.

A. Important Notions

In this subsection, 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
- $\exists (v_1, v_2) \in E \mid v_1 \in S_1 \land v_2 \in S_2$.

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 $\mathcal{P}_{con}(V) = \{S \subseteq V \mid G_{|S} \text{ is connected } \land |S| > 1\}$ 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 I) for certain graph shapes correspond to $|P_{ccp}^{sym}|$. Thus, this explains the 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:

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

The next definition is rather standard, for more details see [9].

Definition 2.4: 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 $V_i = \{v \mid (v = u \vee v = w) \wedge (v, w) \in E_i\}$, 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 $\langle v_0, v_1, v_2, ..., v_l \rangle$, 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 2.5: Let $G = (V, E)$ be a connected undirected graph and $BCC = \{G_1^{BCC}(V_1, E_1), ..., G_k^{BCC}(V_k, E_k)\}$ 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 $E_{tree} = \{(s_{V_i}, v) \mid s_{V_i} \in V_{sn} \wedge v \in V_i\}$

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.

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

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

B. 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 subsubsection, we present our generic top-down algorithm. Afterwards, we explain the naive partitioning strategy.

1) Generic Top-Down Join Enumeration: Our generic top-down join enumeration algorithm TDPLANGEN is based on memoization. We present its pseudocode in Figure 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 build 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.

```
TDPLANGEN($G$)
    ▷ Input: connected G=(V,E), $V = \bigcup_{1 \leq i \leq |V|} \{R_i\}$
    ▷ Output: an optimal join tree for $G$
1   for $i \leftarrow 1$ to $n$
2       do $BestTree(\{R_i\}) \leftarrow R_i$
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 $(S_1, S_2) \in P_{ccp}^{sym}(S)$
3           do BUILDTREE($G_{|S}$, TDPGSUB($G_{|S_1}$),
                         TDPGSUB($G_{|S_2}$))
4   return $BestTree(S)$
```

Fig. 1. Pseudocode for TDPLANGEN

The pseudocode of BUILDTREE is given in Figure 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 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. Note that because of line 3, $BestTree[S]$ in line 6 cannot be NULL. Estimating the costs of the two possible join trees

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

Fig. 2. Pseudocode for BUILDTREE

```
PARTITION$_{naive}(G)$
    ▷ Input: a connected graph $G = (V, E)$
    ▷ Output: $P_{ccp}^{sym}(V)$
1   for all $S \subset V \wedge S \neq \emptyset$
2       do if $max_{index}(S) \leq max_{index}(V \setminus S)$ &&
              $G_{|S}$ is connected &&
              $G_{|V \setminus S}$ is connected
3             then emit($S, V \setminus S$)
```

Fig. 3. Pseudocode for naive partitioning

at the same time rather than separately and comparing them is more efficient, e.g., for cost functions as given in [5], where $card(T_x) \leq card(T_y) \Rightarrow cost(T_x \bowtie T_y) \leq cost(T_y \bowtie T_x)$ holds, with $card$ is the number of tuples or pages and T_x, T_y are (intermediate) relations.

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 Figure 3. In line 1, all $2^{|V|} - 2$ possible non-empty and proper subsets of V are enumerated. For rapid subset enumeration, the method described in [6] 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 $V \setminus S$ in line 2. Three conditions have to be met so that a partition $(S, V \setminus S)$ is a ccp. We check the connectivity of $G_{|S}$ and $G_{|V \setminus S}$ in line 2. The third condition that S needs to be connected to $V \setminus S$ is ensured implicitly by the requirement that the graph handed over as input is connected.

III. GRAPH-BASED JOIN PARTITIONING

DeHaan and Tompa [4] proposed the most efficient top-down join enumeration algorithm known in the literature for the search space of bushy join trees without cross products (Appendix A). As we show, it has a complexity of $O(1)$ for chain, star and cycle queries and $O(|S|^2)$ for clique queries (Appendix B). In this section, we introduce a new join partitioning algorithm called MINCUTBRANCH and study its complexity.

A. 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 $P_{ccp}^{sym}(S)$ so that symmetric ccps are emitted only once. In Figure 4 and 5, we give the algorithm's pseudocode with PARTITION$_{MinCutBranch}$ and MINCUTBRANCH. We call the instantiated generic memoization variant TDMINCUTBRANCH with a TD as a prefix to indicate the *t*op-*d*own algorithm that is based on branch partitioning.

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 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.

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 $D_1, D_2, ..., D_k \subset (S \setminus C)$ that are disjoint and not connected to each other. Hence, those subsets $D_1, D_2, ...D_k$ can only be adjacent to C. Let $v_1, v_2, ..., v_l$ 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 with $1 \leq y \leq l$ 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 towards 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).

B. The Algorithm in Detail

In the following, we take a closer look at the pseudocode given in Figures 4 and 5. 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 line 1 and line 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 only explored 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 $\mathcal{N}(C)$ 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 member of $S \setminus (R_{tmp} \cap X)$. We know that $v_{old} \in \mathcal{N}(C)$ must hold, because v_{old} being a member of X 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 Figure 7. 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, a 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). That means 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

```
PARTITION$_{MinCutBranch}(S)$
    ▷ Input: a connected set $S$
    ▷ Output: $P_{ccp}^{sym}(S)$
1   $t \leftarrow$ arbitrary vertex of $S$
2   MINCUTBRANCH$(S, \{t\}, \emptyset, \{t\})$
```

Fig. 4. Pseudocode for PARTITION$_{MinCutBranch}$

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 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 III-D). 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 III-C. Before we return the call, we join L to the final result set R.

C. 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 that 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) any more. But as we need to ensure that R is correctly computed, we have to add those neighbors for which we want to avoid unnecessary calls to MINCUTBRANCH to N_X (line 20).

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.

D. Exploring Restricted Neighbors

Finally, we explain REACHABLE. 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 once no vertex is left to be added.

E. Two Examples

We illustrate the execution of MINCUTBRANCH by two examples. Tables II and III show the execution steps when the

```
MINCUTBRANCH$(S, C, X, L)$
    ▷ Input: connected sets $S, C$ with $C, X \subset S$, $|L| = 1$
    ▷ Output: ccps for $S$
1   $R \leftarrow \emptyset$
2   $R_{tmp} \leftarrow \emptyset$
3   $N_L \leftarrow ((\mathcal{N}(L) \cap S) \setminus C) \setminus X$
4   $N_X \leftarrow ((\mathcal{N}(L) \cap S) \setminus C) \cap X$
5   $N_B \leftarrow (((\mathcal{N}(C) \cap S) \setminus C) \setminus N_L) \setminus X$
6   while $N_L \neq \emptyset \vee N_X \neq \emptyset \vee N_B \cap R_{tmp} \neq \emptyset$
7        do if $(N_B \cup N_L) \cap R_{tmp} \neq \emptyset$        ▷ case (1)
8             then $v \leftarrow$ a element of $((N_B \cup N_L) \cap R_{tmp})$
9                  MINCUTBRANCH$(S, C \cup \{v\}, X', \{v\})$
10                 $N_L \leftarrow N_L \setminus \{v\}$
11                 $N_B \leftarrow N_B \setminus \{v\}$
12            else $X' \leftarrow X$
13                 if $N_L \neq \emptyset$                  ▷ case (2)
14                   then $v \leftarrow$ a element of $N_L$
15                        $R_{tmp} \leftarrow$ MINCUTBRANCH(
                                   $S, C \cup \{v\}, X', \{v\})$
16                        $N_L \leftarrow N_L \setminus \{v\}$
                                                    ▷ case (3)
17                   else  $v \leftarrow$ a element of $N_X$
18                        $R_{tmp} \leftarrow$ REACHABLE(
                                        $S, C \cup \{v\}, \{v\})$
19                 $N_X \leftarrow N_X \setminus R_{tmp}$
20                 if $R_{tmp} \cap X \neq \emptyset$
21                   then $N_X \leftarrow N_X \cup (N_L \setminus R_{tmp})$
22                        $N_L \leftarrow N_L \cap R_{tmp}$
23                        $N_B \leftarrow N_B \cap R_{tmp}$
24                 if $(S \setminus R_{tmp}) \cap X \neq \emptyset$
25                   then $N_L \leftarrow N_L \setminus R_{tmp}$
26                        $N_B \leftarrow N_B \setminus R_{tmp}$
27                   else  emit $(S \setminus R_{tmp}, R_{tmp})$
28                 $R \leftarrow R \cup R_{tmp}$
29          $X' \leftarrow X' \cup \{v\}$
30  return $R \cup L$
```

Fig. 5. Pseudocode for MINCUTBRANCH

```
REACHABLE$(S, C, L)$
    ▷ Input: a connected set $S$, $C \subseteq S$, $L \subseteq C$, $|L| = 1$
    ▷ Output: connected set $R$ adjacent to $C$
1   $R \leftarrow L$
2   $N \leftarrow (\mathcal{N}(L) \cap S) \setminus C$
3   while $N \neq \emptyset$
4        do $R \leftarrow R \cup N$
5           $N \leftarrow ((\mathcal{N}(N) \cap S) \setminus C)$
6   return $R$
```

Fig. 6. Pseudocode for REACHABLE

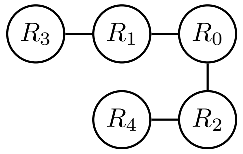

Fig. 7. Chain Query

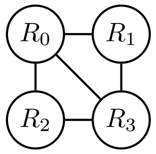

Fig. 8. Cyclic Query

chain query of Figure 7 or, respectively, the cyclic query of Figure 8 is given as input. The first column named *level* keeps track of the recursion level. The root invocation is indicated with a 0. Column 2 shows which *case* in the parent invocation has initiated the current call. Due to the lack of space, we omit invocations where $N_L = N_X = N_B = \emptyset$, because they return immediately to the parent invocation by avoiding the loop of lines 6 to 29.

For all acyclic graphs, MINCUTBRANCH has only case 2 to consider. Table II confirms this for chain graphs. The maximal recursion depth depends on the position of the start vertex. Here, it is 3, but it is not shown, because the recursion with $L = \{R_3\}$ and $L = \{R_4\}$ is omitted. For the graph of Figure 8, we have the same recursion depth and again a recursion with $L = \{R_2\}$, following the third entry in Table III, is omitted. As can be seen for this example at the last three entries of Table III, there is a recursive invocation of MINCUTLAZY with $C = \{R_0, R_3\}$ and $X = \{R_1, R_2\}$ that does not emit any further ccps. Unfortunately, this is an execution overhead that cannot be avoided easily.

TABLE II
EXEMPLIFIED EXECUTION OF MINCUTBRANCH FOR THE GRAPH OF FIG. 7

level	case	C	L	X	N_L	N_X	N_B
0	-	$\{R_0\}$	$\{R_0\}$	$\emptyset$	$\{R_1, R_2\}$	$\emptyset$	$\emptyset$
1	2	$\{R_0, R_1\}$	$\{R_1\}$	$\emptyset$	$\{R_3\}$	$\emptyset$	$\emptyset$
1	MCB. returns $\{R_3\} \rightarrow$ emitting $(\{R_3\}, \{R_0, R_1, R_2, R_4\})$						
0	MCB. returns $\{R_3, R_1\} \rightarrow$ emitting $(\{R_1, R_3\}, \{R_0, R_2, R_4\})$						
1	2	$\{R_0, R_2\}$	$\{R_2\}$	$\emptyset$	$\{R_4\}$	$\emptyset$	$\emptyset$
1	MCB. returns $\{R_4\} \rightarrow$ emitting $(\{R_4\}, \{R_0, R_1, R_2, R_3\})$						
0	MCB. returns $\{R_4, R_2\} \rightarrow$ emitting $(\{R_2, R_4\}, \{R_0, R_1, R_3\})$						

TABLE III
EXEMPLIFIED EXECUTION OF MINCUTBRANCH FOR THE GRAPH OF FIG. 8

level	case	C	L	X	N_L	N_X	N_B
0	-	$\{R_0\}$	$\{R_0\}$	$\emptyset$	$\{R_1, R_2, R_3\}$	$\emptyset$	$\emptyset$
1	2	$\{R_0, R_1\}$	$\{R_1\}$	$\emptyset$	$\{R_3\}$	$\emptyset$	$\{R_2\}$
2	2	$\{R_0, R_1, R_3\}$	$\{R_3\}$	$\emptyset$	$\{R_2\}$	$\emptyset$	$\emptyset$
2	MCB. returns $\{R_2\} \rightarrow$ emitting $(\{R_2\}, \{R_0, R_1, R_3\})$						
1	MCB. returns $\{R_2, R_3\} \rightarrow$ emitting $(\{R_2, R_3\}, \{R_0, R_1\})$						
2	1	$\{R_0, R_1, R_2\}$	$\{R_2\}$	$\{R_3\}$	$\emptyset$	$\{R_3\}$	$\emptyset$
2	REACHABLE returns $\{R_3\} \rightarrow$ emitting $(\{R_3\}, \{R_0, R_1, R_2\})$						
0	MCB. returns $\{R_2, R_3, R_1\} \rightarrow$ emitting $(\{R_1, R_2, R_3\}, \{R_0\})$						
1	2	$\{R_0, R_2\}$	$\{R_2\}$	$\{R_1\}$	$\{R_3\}$	$\emptyset$	$\emptyset$
2	2	$\{R_0, R_2, R_3\}$	$\{R_3\}$	$\{R_1\}$	$\emptyset$	$\{R_1\}$	$\emptyset$
2	REACHABLE returns $\{R_1\} \rightarrow$ emitting $(\{R_1\}, \{R_0, R_2, R_3\})$						
1	MCB. returns $\{R_1, R_3\} \rightarrow$ emitting $(\{R_1, R_3\}, \{R_0, R_2\})$						
1	2	$\{R_0, R_3\}$	$\{R_3\}$	$\{R_1, R_2\}$	$\emptyset$	$\{R_1, R_2\}$	$\emptyset$
1	2 calls to REACHABLE return $\{R_1\}$ and $\{R_2\}$						
0	MCB. returns $\{R_1, R_2, R_3\}$						

F. Complexity of Branch Partitioning

We determine the complexity of MINCUTBRANCH to emit successive ccps by $O(\frac{i+r+l}{|P_{ccp}^{sym}(S)|})$, where i is the number of iterations of the loop in line 6, r is the number of all invocations of REACHABLE and l is the number of all iterations of the loop in line 3 of REACHABLE.

For acyclic graphs we know that $|P_{ccp}^{sym}(S)| = |S| - 1$ holds. Furthermore, no $v \in N_B \cup N_X$ will be processed. Therefore, $i = |S| - 1$ and $r = l = 0$ holds, since there is no call to REACHABLE. Hence, the complexity of MINCUTBRANCH to emit a ccp for acyclic graphs is in $O(1)$.

A cycle query has $|P_{ccp}^{sym}(S)| = \frac{1}{2}|S|^2 \setminus \frac{1}{2}|S|$ symmetric ccps for S. Each of the first $|S| - 1$ invocations processes a neighbor taken from the set N_L. That recursive descent is always initiated through line 15. There are $|S| - 2$ second invocations of the loop of line 6 calling MINCUTBRANCH from line 9. Those invocations process further $\sum_{k=1}^{|S|-2} k$ neighbors in total. Altogether, there are $|S| - 1 + |S| - 2| + \sum_{k=1}^{|S|-2} k = \frac{1}{2}|S|^2 + 12|S| - 2 = i$ neighbors processed. REACHABLE is called $r = |S| - 2$ times, and the loop of line 3 never iterates, so that $l = 0$ holds. Therefore, the total complexity per emitted ccp is $\frac{|S|^2 + 3|S| - 8}{|S|(|S|-1)}$, which decreases asymptotically to 1, so that the complexity is $O(1)$.

Considering clique queries, we know that $|P_{ccp}^{sym}(S)| = 2^{|S|-1} - 1$ holds. There are $2^{|S|-1}$ neighbors processed that are element of N_L. Furthermore, there are $2^{|S|-2} - 1$ neighbors processed that are element of N_X. Therefore $i = 2^{|S|-1} + 2^{|S|-2} \setminus 1 = \frac{3}{4}2^{|S|} - 1$ and $r = 2^{|S|-2} - 1$ holds. The number of iterations through the loop of line 3 must be $|S| - 2$ times less than there are calls to REACHABLE, so that $l = 2^{|S|-2} - |S| - 3$ holds. To emit all symmetric ccps, the complexity is $\frac{5}{4}2^{|S|} - |S| - 5$. Per emitted ccp the complexity increases asymptotically to $\frac{5}{2}$. Hence, the complexity for clique queries is in $O(1)$.

IV. EVALUATION

This section summarizes our experimental findings. We start by briefly describing our setup. Then, we concentrate on the enumeration costs per ccp, which, as we have argued in the introduction, is fundamental (Sec. IV-B). Sec. IV-C compares the complete execution times of TDMINCUTBRANCH and TDMINCUTLAZY. Finally, Sec.IV-D is devoted to a comparison of these algorithms with the best bottom-up join enumerator. For completeness, it also shows the depressing results for MEMOIZATIONBASIC, which underlines the challenge discussed in the introduction.

A. 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 csg-cmp-pairs. 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 pre-calculated 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

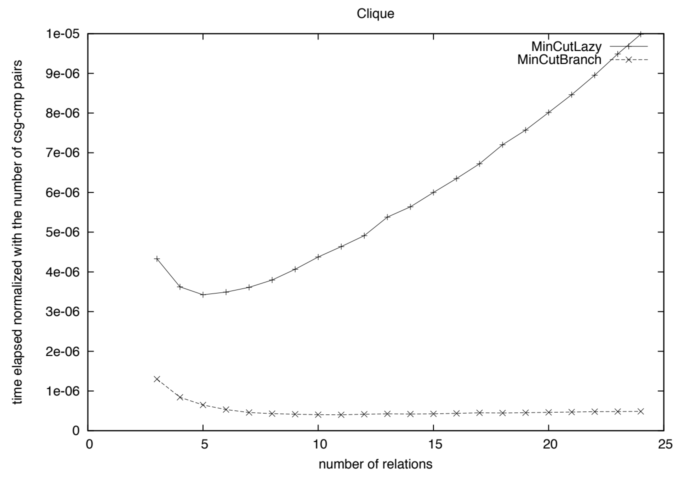

Fig. 9. Cost per emitted ccp of MinCutLazy and MinCutBranch for clique queries.

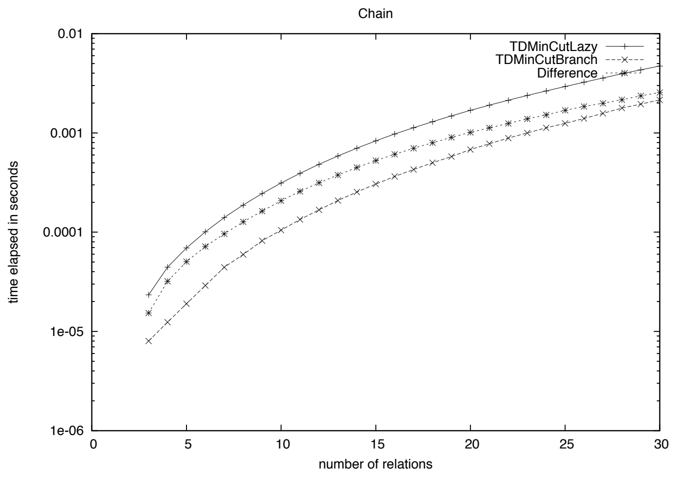

Fig. 10. Performance of TDMinCutLazy and TDMinCutBranch with chain queries.

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.

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.

B. Partitioning Costs

We have analyzed the complexity of lazy minimal cut partitioning (Appendix B) and branch partitioning (Section III-F) for chain, star, cycle and clique queries. For both partitioning strategies, the complexity is in $O(1)$ for chain, star and cycle queries per emitted ccp. But when clique queries are considered, MinCutLazy has a complexity in $O(|S|^2)$ whereas MinCutBranch is still in $O(1)$. Therefore, we have measured the partitioning costs and discuss them here for clique queries. Figure 9 shows our results with the number of vertices on the abscissa and the execution time per emitted ccp on the ordinate.

The costs per emitted ccp are decreasing for a small number of vertices. But with five and more vertices, the costs for lazy minimal cut partitioning are increasing again. The increase is quadratic. For MinCutBranch the costs are dropping for less than ten vertices. After that, they are slightly increasing. The decrease at the start of both curves is due to some instantiation overhead that becomes negligible compared to the other processing costs when a higher number of vertices is considered. Our results support a quadratic increase as proven by our complexity analysis, but this effect is rather weak for the number of vertices considered here. Note that the effect was weakened by our implementations, since we have used inline assembler instructions to minimize the accessing cost of the data structure MinCutLazy relies on.

For MinCutBranch, we have proven that the complexity is increasing asymptotically to a constant factor. Our results show a very weak increase. This is caused by an increasing number of cache misses for an increasing number of vertices.

In summary, our results show that the performance differences between the two algorithms for clique queries are strongly increasing with a higher number of vertices.

For the other three graph shapes, MinCutBranch clearly dominates MinCutLazy, but the differences are not as strong as for clique queries, because both algorithms have only a constant overhead in those scenarios.

C. Costs of Plan Generation

The Figures 10 to 17 present the performance results for TDMinCutLazy and TDMinCutBranch for certain query shapes. Contrary to the previous section, we do not measure the cost for *one* call to the partitioning algorithm but the overall cost for one call to TDPlanGen. This includes the cost for all calls to BuildTree and the costs of *all* calls to Partition$_i$.

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 (Figures 10 to 14), we give the number of vertices on the abscissa and the execution time in log scale on the ordinate. We connect the averaged execution times with lines.

Since for randomly generated cyclic queries the performance results of the algorithms deviate significantly for the same number of vertices, we show the results separately for different numbers of vertices. At the abscissa, we chose to display the number of edges and again the execution time in log scale on the ordinate. We do not present the exact results, but results smoothed by Bezier curves (Figures 15 to 17).

In all charts, we also present the differences of TDMinCutLazy's and TDMinCutBranch's runtime. Since the

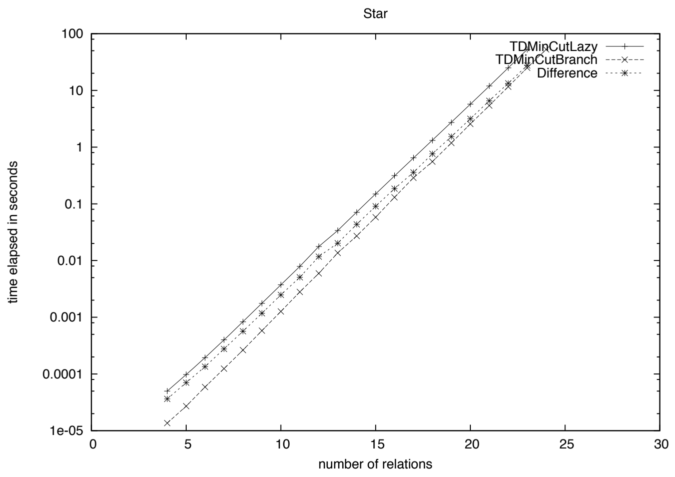

Fig. 11. Performance of TDMINCUTLAZY and TDMINCUTBRANCH with star queries.

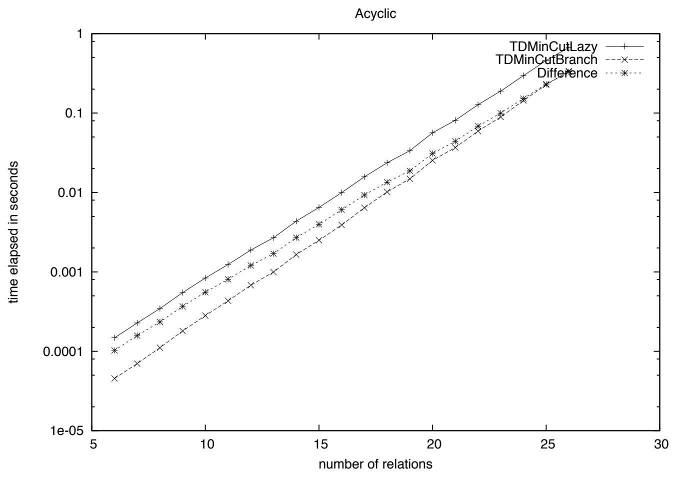

Fig. 12. Performance of TDMINCUTLAZY and TDMINCUTBRANCH with random acyclic queries that are neither chain nor star queries.

effort of the join cost calculations is exactly the same for both algorithms (for a given query graph of course), the difference of both runtimes equals the difference of the partitioning costs for both algorithms. Hence, the time spent for all calls to BUILDTREE is not contained in the introduced *difference* curve any more. By normalized runtime we denote the quotient of TDMINCUTLAZY's execution time and TDMINCUTBRANCH's execution time. Although the normalized runtime can be easily calculated from the absolute runtimes, we include the normalized runtimes in separate charts in our technical report [7].

For all scenarios, we can see that TDMINCUTBRANCH dominates TDMINCUTLAZY. Since the *difference* curve is always above TDMINCUTBRANCH, the runtime of TDMINCUTLAZY is at least twice as high as TDMINCUTBRANCH's runtime. Furthermore, the differences in the partitioning costs alone exceed *all* costs of MINCUTBRANCH. But note that the time spent for cardinality estimation and plan cost computation in today's database environments would be a multiple of the time spent with our implementation of BUILDTREE. Nevertheless, this would not change the shape of our *difference* curves!

For chain, star and random acyclic queries, the runtimes of both graphs are converging up to a factor of two. This is not surprising, since TDMINCUTLAZY has to build just one biconnection tree per call to the partitioning algorithm. Which means that the fixed overhead for allocating the biconnection tree data structure becomes negligible with an increasing number of vertices. Among acyclic graphs, star queries have the highest number of ccps and chain queries the lowest number of ccps. Hence, star queries have the highest execution times for the same number of vertices and chain queries have the lowest. Random acyclic graphs range in the middle. The normalized runtimes range between a factor of 2 to 3 for all acyclic query shapes, as can be seen in detail in [7].

In terms of join enumeration, cycle queries are the least complex queries among cyclic queries. The results for cycle queries in Figure 13 resembles the chart for chain queries. This is as expected, since only the input of the root invocation of TDPLANGEN is a cycle, for all other invocations the cycle is broken off into a chain. Cycle queries are on one side of the spectrum of cyclic graphs and clique queries, as the most complex queries, on the other. Here, we have the highest runtime, because clique queries have the highest number of ccps. One can see that the *difference* of both runtimes almost converges with the runtime of TDMINCUTLAZY. We have measured the execution times for up to 16 vertices. As displayed, the normalized runtime of TDMINCUTLAZY increases to 5. Since the partitioning costs are in $O(|S|^2)$ (also compare Section IV-B), the normalized runtime would increase as well for more vertices.

As we have seen from our experiments, the normalized runtime for cyclic queries ranges between a factor of two (cycle queries) to a factor of five (clique queries). In Figure 15 and Figure 17, we show the results for random cyclic queries. We observe that the *difference* curve is shifting towards the curve of TDMINCUTLAZY with an increasing number of vertices. For those random graph shapes, the normalized runtime ranges between a factor of 3 to 6, rising with the number of relations and join predicates.

D. Overall Comparison

The previous section has shown that our new algorithm is clearly superior to TDMINCUTLAZY. But now we want to see how it compares to MEMOIZATIONBASIC and state-of-the-art bottom-up join enumeration with Moerkotte's and Neumann's DPCCP. Due to the lack of space, we do not present our results as charts, but summarize them in Table IV for acyclic graphs and in Table V for cyclic graphs. We give the results as relative factors, where we compare an algorithm's runtime to DPCCP's runtime. We give the minimal value, the maximal value and the average value over all results for a certain query type. Note that these aggregated values are taken from an average value over multiple runs of the algorithms with the same input.

We observe that MEMOIZATIONBASIC with a generate and test partitioning strategy performs certainly worst for acyclic graphs and graphs with a low number of edges relative

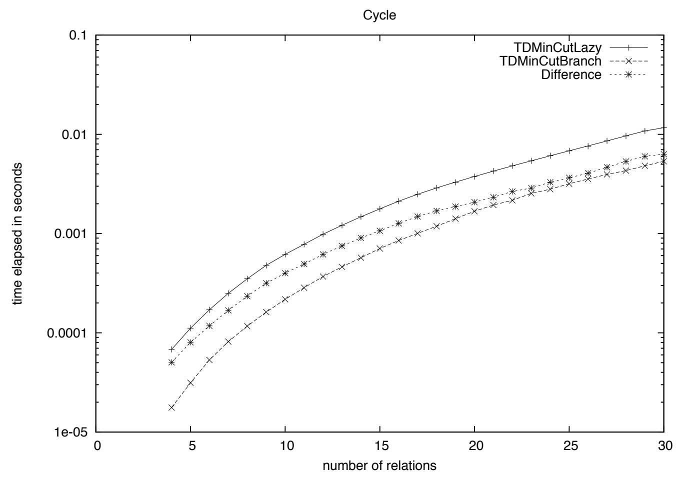

Fig. 13. Performance of TDMINCUTLAZY and TDMINCUTBRANCH with cycle queries.

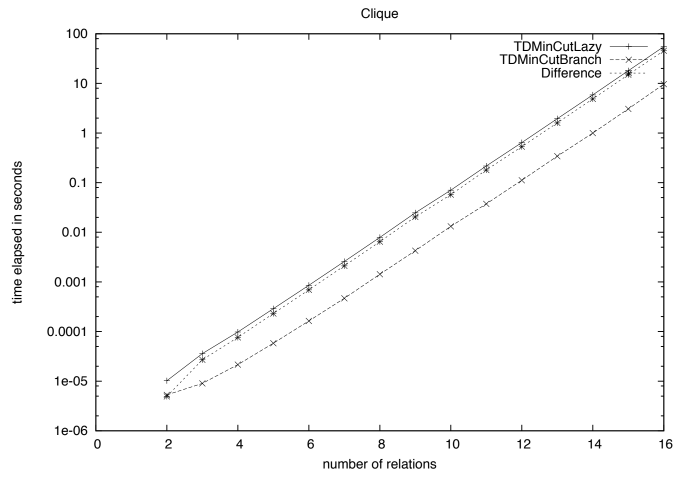

Fig. 14. Performance of TDMINCUTLAZY and TDMINCUTBRANCH with clique queries.

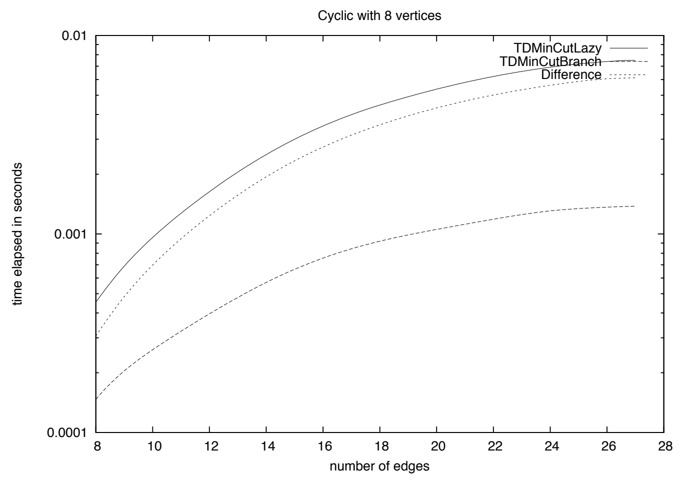

Fig. 15. Performance of TDMINCUTLAZY and TDMINCUTBRANCH for cyclic queries with 8 vertices.

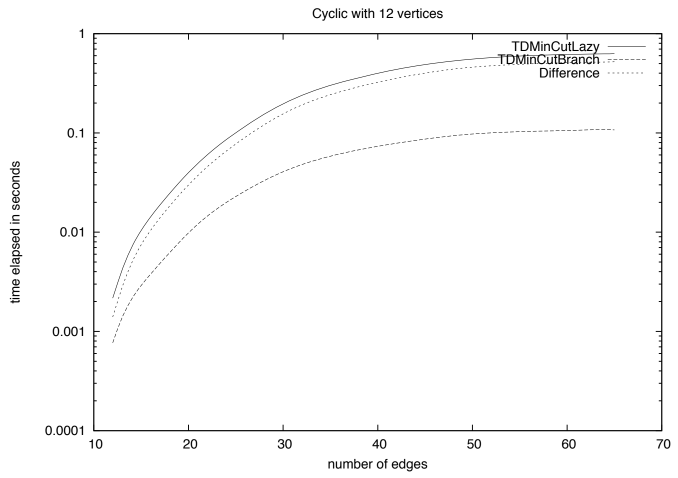

Fig. 16. Performance of TDMINCUTLAZY and TDMINCUTBRANCH for cyclic queries with 12 vertices.

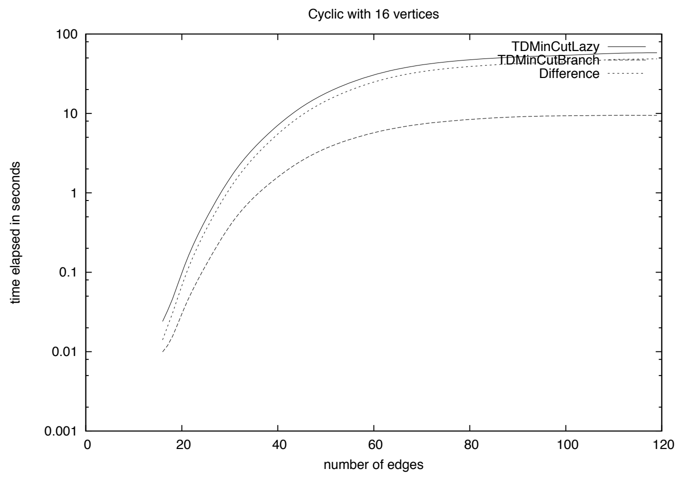

Fig. 17. Performance of TDMINCUTLAZY and TDMINCUTBRANCH for cyclic queries with 16 vertices.

to the number of vertices. With a factor of almost 4700, for chain queries the algorithm is a bad choice for acyclic graphs. TDMINCUTLAZY performs relatively well with a maximal factor of 3.2. 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. The lowest factor we could determine was at 0.66 for random acyclic queries.

For random cyclic queries and an increasing number of vertices, MEMOIZATIONBASIC starts dominating TDMINCUTLAZY. For clique queries it performs on average more than 3 times faster. TDMINCUTBRANCH performs for these types of queries only second best. The average values of 1.09 for clique and 1.13 for random cyclic queries are very competitive, but the factors can also range to 1.47, which is still a very impressive result, compared to TDMINCUTLAZY and MEMOIZATIONBASIC.

As a result of our empirical analysis, we propose TDMINCUTBRANCH as the algorithm of choice for bottom-up processing, since it is the fastest algorithm for acyclic graphs and performs very well for random cyclic graphs. We can contrast the worst relative factor of 1.47 with a factor of $\frac{1}{0.66} = 1.52$ being faster than the state-of-the-art with DPccp, although no branch-and-bound pruning is put in place.

V. Conclusion

We presented a new top-down join enumeration algorithm, which has two advantages over the best bottom-up algorithm known so far:

- 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 our new algorithm 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.

Acknowledgment. We thank Simone Seeger for her help preparing this manuscript and the referees for their thorough feedback.

References

[1] Y. E. Ioannidis and Y. C. Kang, "Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization," *SIGMOD Rec.*, vol. 20, no. 2, pp. 168–177, 1991.

[2] K. Ono and G. M. Lohman, "Measuring the complexity of join enumeration in query optimization," in *VLDB*, 1990, pp. 314–325.

[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," in *VLDB*, 2006, pp. 930–941.

[4] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration," in *SIGMOD*, 2007, pp. 1098–1109.

[5] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla, "Seeking the truth about ad hoc join costs," *VLDB Journal*, vol. 6, pp. 241–256, 1997.

[6] B. Vance and D. Maier, "Rapid bushy join-order optimization with cartesian products," in *SIGMOD*, 1996, pp. 35–46.

[7] P. Fender and G. Moerkotte, "A new, highly efficient, and easy to implement top-down join enumeration algorithm (extended version)," University of Mannheim, Tech. Rep., 2010.

[8] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration (extended version)," University of Waterloo, Tech. Rep., 2007.

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

Appendix

A. 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 Figure 18. MinCutLazy can be used to instantiate the generic top-down algorithm to which we then refer 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 $\mathcal{D}_\mathcal{T}(v)$ [4] 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 $\mathcal{A}_\mathcal{T}(v)$ [4] of a $v \in \mathcal{N}(C)$ 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 BuildBccTree [8]. The biconnection tree building complexity is in the worst case in $O(|S|^2)$. To avoid the unnecessary re-computation 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.

```
Partition_{MinCutLazy}($S$)
   ▷ Input: connected set $S$
   ▷ Output: $P^{sym}_{ccp}(S)$
1  $t \leftarrow$ arbitrary vertex of $S$
2  MinCutLazy($S, \emptyset, \emptyset, \{t\}$, NULL, $t$)

MinCutLazy($S, C, C_{diff}, X, \mathcal{T}', t$)
    ▷ Input: connected set $S$, $C \cap X = \emptyset$, $C_{diff} \subseteq C$
    ▷ Output: ccps for $S$
 1  if $C \neq \emptyset$
 2     then emit $(C, S \setminus C)$
 3  if $\mathcal{N}(C) \subseteq X$
 4     then return
 5  if isUsable($\mathcal{T}', C_{diff}$)
 6     then $\mathcal{T} \leftarrow \mathcal{T}'$
 7     else $\mathcal{T} \leftarrow$ BuildBccTree($S \setminus C, t$)
 8  $P \leftarrow \{v \in \mathcal{N}(C) | v \in (S \setminus X) \wedge$
          $(\mathcal{D}_\mathcal{T}(v) \cap \mathcal{N}(C)) = \{v\}\}$
 9  $X' \leftarrow X$
10  for all $v \in P$
11        do MinCutLazy($S, C \cup \mathcal{D}_\mathcal{T}(v), \mathcal{D}_\mathcal{T}(v), X', \mathcal{T}, t$)
12           $X' \leftarrow X' \cup \mathcal{A}_\mathcal{T}(v)$
    ▷ $\mathcal{N}(\emptyset) = S \setminus \{t\}$
```

Fig. 18. Pseudocode for MinCutLazy

B. Complexity of Lazy Minimal Cut Partitioning

We analyze the complexity of DeHaan's and Tompa's MinCutLazy for fixed shape query graphs which are chain, star, cycle and clique queries. We base our calculations on Fender's and Moerkotte's analysis of the biconnection tree building [7] which is $|E| + 2|S| - 2 + |A|$, where A is the set of articulation vertices [9] of a $G = (S, E)$. Furthermore, our analysis presumes a simplification of line 8 which saves unnecessary iterations during the calculation of the pivot set. Now, the revised computation in line 8 is: $P \leftarrow \{v \in \mathcal{N}(C) \setminus X \mid (\mathcal{D}_\mathcal{T}(v) \cap \mathcal{N}(C)) = \{v\}\}$. For one

TABLE IV
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
TDMinCutBranch	0.74	0.98	0.85	0.77	1.30	1.04	0.66	1.03	0.85

TABLE V
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	0.97	4392.17	117.19	0.82	1.93	1.42	1.10	200.93	5.76
TDMinCutLazy	1.63	3.70	2.42	3.60	7.40	6.00	2.13	8.00	5.63
TDMinCutBranch	0.70	1.20	0.98	0.79	1.26	1.09	0.78	1.47	1.13

call to PARTITION$_{MinCutLazy}$, we compute the algorithm's complexity in the form of $O(\frac{O_t+O_u+O_p+O_i}{|P^{sym}ccp(S)|})$, where O_t is the complexity of all biconnection tree buildings, O_u the complexity of all usability tests, O_p the complexity for computing all pivot sets P, and O_i the complexity of all iterations of the loop in line 10.

First, we consider chain queries. We know that $|A| = |S|-2$ holds and only one biconnection tree has to be built. Hence, $O_t = |S| - 1 + 2|S| - 2 + |S| - 2 = 4|S| - 5$ holds. The number of usability tests will be $|S|-3$ times at minimum and $|S|-2$ times at maximum at $O(1)$ cost each, because at every step $|\mathcal{D}_\mathcal{T}(v)| = 1$ holds. The condition of line 8 $(\mathcal{D}_\mathcal{T}(v) \cap \mathcal{N}(C)) = \{v\}$ for computing the pivot set P is evaluated $2|S|-4$ times at least and $2|S|-3$ times at most at $O(1)$ cost each. MINCUTLAZY is invoked $|S|-1$ times in line 11, which we account with $O(1)$ cost each. In total, the complexity is $4|S|-5+|S|-2+2|S|-3+|S| = 8|S|-11$, and there are $|S|-1$ ccps emitted, whereas symmetric ccps are counted only once. Therefore, the complexity of MINCUTLAZY to emit a ccp corresponds to $O(1)$.

Next, star queries are to be considered. Again, there will be only one biconnection tree building with a complexity of $|S| - 1 + 2|S| - 2 + 1 = 3|S| - 2$, because $|A| = 1$ holds. Depending on whether the hub of the star is chosen as the root vertex t of the biconnection tree, there is no usability test required and otherwise, one usability test at $O(1)$ cost. The condition of line 8, $(\mathcal{D}_\mathcal{T}(v) \cap \mathcal{N}(C)) = \{v\}$, is computed $|S|-1$ times at least and $|S|$ times at most at $O(1)$ cost each. There are $|S|-1$ times that MINCUTLAZY is invoking itself in line 11, which we account with $O(1)$ cost each. In total, the complexity is $3|S|-2+1+|S|+|S| = 5|S|-2$, and since there are $|S|-1$ ccps emitted, the complexity of MINCUTLAZY to emit a ccp is in $O(1)$.

Let us now consider cycle queries. First of all, we know that $|E| = |S|$ holds. There are $|S|$ many connected subgraphs of size k, with $k < |S|$ and $(|S|-1)|S| = |S|^2 - |S|$ in total. When we count symmetric ccps only once, the number of ccps is $\frac{1}{2}|S|^2 - \frac{1}{2}|S|$. Lazy minimal cut partitioning needs one initial biconnection tree building for S and at the most $|S|-2$ buildings for the complements of size $|S|-1$ that are chain graphs, because the cycle is broken off. This yields a worst case complexity of $|S|-2|S|-2+1+(|S|-1)(|S|-2+2(|S|-1)-2+|S|-3) = 4|S|^2 - 18|S| + 17$ for all tree buildings. MINCUTLAZY invokes itself $\frac{1}{2}|S|^2 - \frac{1}{2}|S|$ times. The same holds for the number of times that the condition of line 8, $(\mathcal{D}_\mathcal{T}(v) \cap \mathcal{N}(C)) = \{v\}$, is computed, again at $O(1)$ cost each. The number of tree usability tests evaluated is at least $|S|-1$ lower than MINCUTLAZY invokes itself, because of the early exit in line 1. Therefore, there are $\frac{1}{2}|S|^2 - \frac{3}{2}|S| + 1$ usability tests with $O(1)$ cost each. In total, the worst case complexity is $4|S|^2-18|S|+17+\frac{1}{2}|S|^2-\frac{3}{2}|S|+1+2(\frac{1}{2}|S|^2-\frac{1}{2}|S|) = \frac{11}{2}|S|^2 - \frac{41}{2}|S| + 18$. Therefore, the complexity of MINCUTLAZY to emit a ccp is in $O(1)$.

Finally, we analyze the algorithm's complexity for clique queries. For every clique it holds that $|E| = \frac{|S|(|S|-1)}{2}$. Since a powerset of a set with n elements has 2^n-1 nonempty subsets, for a clique $|P^{sym}ccp(S)| = 2^{|S|-1}-1$ holds. When determining the complexity of the biconnection tree buildings for clique queries, it is important how often the tree building algorithm is called for a subgraph $G_{S\setminus C}$. Since the vertex t, arbitrarily chosen during the invocation of PARTITION$_{MinCutLazy}$, must always be part of a complement $S \setminus C$, there are $\binom{|S|-1}{k-1}$ possible complements of size k. Because of the early exit in line 1 of MINCUTLAZY, there are no biconnection trees built for complements of size $k = 1$ or if $S \setminus C = X$ holds. This means that there are only $\binom{|S|-2}{k-2}$ biconnection tree buildings of size k. In total, there are $\sum_{K=1}^{|S|}\binom{|S|-2}{k-2} = 2^{|S|-2}$ biconnection tree buildings with a complexity of $\sum_{K=1}^{|S|}(\binom{|S|-2}{k-2}\frac{k^2+3k-4}{2}) = \frac{1}{32}2^{|S|}(|S|^2 + 11|S| - 2)$. There are as many tree usability tests as biconnection tree buildings, each at $O(1)$ costs, since $\forall v \in S : |\mathcal{D}_\mathcal{T}(v)| = 1$ holds. This results in a complexity of $2^{|S|-2}$ for all tests. MINCUTLAZY is called $2^{|S|-1}$ times, which we count with a complexity of $2^{|S|-1}$. The condition $(\mathcal{D}_\mathcal{T}(v) \cap \mathcal{N}(C)) = \{v\}$ is calculated $2^{|S|-1}$ times at cost of $O(1)$ each. The total complexity is $\frac{1}{32}2^{|S|}(|S|^2+11|S|-2)+2^{|S|-2}+2*2^{|S|-1} = \frac{1}{32}2^{|S|}(|S|^2+11|S|+38)$. When we assume that the number of ccps is $|P^{sym}ccp(S)| = 2^{|S|-1}$ instead of $2^{|S|-1}-1$, we have a complexity of $\frac{1}{16}|S|^2 + \frac{11}{16}|S| + \frac{19}{8}$ per emitted ccp, which corresponds to $O(|S|^2)$.