Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products

Guido Moerkotte
University of Mannheim
68131 Mannheim
Germany
moerkotte@informatik.uni-mannheim.de

Thomas Neumann
Max Planck Institute for Informatics
66123 Saarbrücken
Germany
neumann@mpi-inf.mpg.de

ABSTRACT

Two approaches to derive dynamic programming algorithms for constructing join trees are described in the literature. We show analytically and experimentally that these two variants exhibit vastly diverging runtime behaviors for different query graphs. More specifically, each variant is superior to the other for one kind of query graph (chain or clique), but fails for the other. Moreover, neither of them handles star queries well. This motivates us to derive an algorithm that is superior to the two existing algorithms because it adapts to the search space implied by the query graph.

1. INTRODUCTION

For the overall performance of a database management system, the cost-based query optimizer is an essential piece of software. One important and complex problem any cost-based query optimizer has to solve is that of finding the optimal join order. In their seminal paper, Selinger et al. not only introduced cost-based query optimization but also proposed a dynamic programming algorithm to find the optimal join order for a given conjunctive query [7]. More precisely, they proposed to generate plans in the order of increasing size. Although they restricted the search space to left-deep trees, the general idea of their algorithm can be used to derive an algorithm to explore the space of bushy trees. Its pseudocode is shown in Fig. 1. The algorithm still forms the core of state of the art commercial query optimizers like the one of DB2 [2] and still is the foundation for further research on join ordering, e.g. in the context of distributed database management systems [3].

Given the widespread and prominent use of dynamic programming algorithms for finding a good join order, it came to us as a surprise that only two publications analyze the complexity of these algorithms. The first publication is the seminal paper by Ono and Lohman [5]. In order to understand their results, it is important to note that the algorithm (see Fig. 1) follows a generate-and-test approach. Ono and Lohman analyse the number of times the tests succeed (see Fig. 1). Obviously, this number is highly dependent on the query graph. Therefore, Ono and Lohman consider chain, star, and clique queries. As pointed out by Vance, this only gives a lower bound for all dynamic programming algorithms [9]. The real complexity is the number of times the code within the inner loop (the test of the first **if**-statement in the innermost loop) is executed. Moreover, Vance gives an analytical result for clique queries for an unoptimized version of the size-based variant described above [9].

Vance and Maier proposed an algorithm which generates subsets extremely fast [9, 10]. They use this routine to generate optimal bushy join trees containing cross products. The core idea is to generate partial plans in a different order than the algorithm mentioned above. If cross products are to be considered, their algorithm can hardly be improved. However, as generating cross products vastly increases the search space [5], it is a very interesting exercise to modify their algorithm such that it excludes cross products. Modifying it such that it also follows the generate-and-test paradigm results in the algorithm given in Fig. 2. For this algorithm, no complexity bounds are known.

Our first contribution is that we analytically analyze the time complexity of both algorithms for chain, cycle, star, and clique queries (Sec. 2). This allows us to analytically compare their performance on these query graph instances. Let us call the first algorithm `DPsize` and the second `DPsub`. Then the comparison of these algorithms reveals (see also Sec. 2):

1. For chain and cycle queries, `DPsize` is highly superior to `DPsub`.
2. For star and clique queries, `DPsub` is highly superior to `DPsize`.
3. Both algorithms are far worse than the lower bound given by Ono and Lohman.

These findings immediately spawn the following question: Is it possible to derive a dynamic programming algorithm whose complexity meets the lower bound derived by Ono

```
DPsize
Input: a connected query graph with relations $R = \{R_0, \ldots, R_{n-1}\}$
Output: an optimal bushy join tree without cross products

for all $R_i \in R$ {
    BestPlan($\{R_i\}$) = $R_i$;
}
for all $1 < s \leq n$ ascending // size of plan
for all $1 \leq s_1 < s$ { // size of left subplan
    $s_2 = s - s_1$; // size of right subplan
    for all $S_1 \subset R : |S_1| = s_1$
            $S_2 \subset R : |S_2| = s_2$ {
        ++InnerCounter;
        if ($\emptyset \neq S_1 \cap S_2$) continue;
        if not ($S_1$ connected to $S_2$) continue;
        ++CsgCmpPairCounter;
        $p_1$=BestPlan($S_1$);
        $p_2$=BestPlan($S_2$);
        CurrPlan = CreateJoinTree($p_1, p_2$);
        if (cost(BestPlan($S_1 \cup S_2$)) > cost(CurrPlan)) {
            BestPlan($S_1 \cup S_2$) = CurrPlan;
        }
    }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan($\{R_0, \ldots, R_{n-1}\}$);
```

Figure 1: Algorithm DPsize

and Lohman? If so, this variant would clearly outperform the other variants. The positive answer to this question is presented in Section 3.

Next, we were interested in the practical implication of our theoretical analysis. Hence, we implemented all three algorithms in a plan generator and ran experiments. Some typical results are presented in Section 4. Section 5 concludes the paper.

2. ALGORITHMS AND ANALYSIS

In this section, we present two dynamic programming algorithms to generate optimal bushy trees without cross products. The first two subsections discuss the pseudocode of these algorithms. We start by discussing the common infrastructure used by all our algorithms (including our new one). Section 2.1 sketches the most prominent size-chained variant. Section 2.2 presents two variants based on fast subset generation [9, 10]. Both subsections give analytical results on the time complexity of the algorithm they present. The analytical results independent of any algorithm are presented in Section 2.3.2 after introducing some definitions in Subsection 2.3.1. Section 2.4 applies the formulas presented so far and draws important conclusions.

2.1 Size-Driven Enumeration

In general, dynamic programming generates solutions for a larger problem in a bottom-up fashion by combining solutions for smaller problems [1]. Taking this description literally, we can construct optimal plans of size n by joining plans P_1 and P_2 of size k and $n-k$. We just have to take care that (1) the sets of relations contained in P_1 and P_2 do not overlap, and (2) there is a join predicate connecting a relation P_1 with a relation in P_2. After this remark, we are prepared to understand the pseudocode for algorithm DPsize (see Fig. 1). A table BestPlan associates with each set of relations the best plan found so far. The algorithm starts by initializing this table with plans of size one, i.e. single relations. After that, it constructs plans of increasing size (loop over s). Thereby, the first size considered is two, since plans of size one have already been constructed. Every plan joining n relations can be constructed by joining a plan containing s_1 relations with a plan containing s_2 relations. Thereby, $s_i > 0$ and $s_1 + s_2 = n$ must hold. Thus, the pseudocode loops over s_1 and sets s_2 accordingly. Since for every possible size there exist many plans, two more loops are necessary in order to loop over the plans of sizes s_1 and s_2. Then, conditions (1) and (2) from above are tested. Only if their outcome is positive, we consider joining the plans p_1 and p_2. The result is a plan CurrPlan. Let S be the relations contained in CurrPlan. If BestPlan does not contain a plan for the relations in S or the one it contains is more expensive than CurrPlan, we register CurrPlan with BestPlan.

The algorithm DPsize can be made more efficient in case of $s_1 = s_2$. The algorithm as stated cycles through all plans p_1 joining s_1 relations. For each such plan, all plans p_2 of size s_2 are tested. Assume that plans of equal size are represented as a linked list. If $s_1 = s_2$, then it is possible to iterate through the list for retrieving all plans p_1. For p_2 we consider the plans succeeding p_1 in the list. Thus, the complexity can be decreased from $s_1 * s_2$ to $s_1 * s_2/2$ The following formulas are valid only for the variant of DPsize where this optimization has been incorporated (see [4] for details).

We now come to the first important contribution of our paper. If the counter InnerCounter is initialized with zero at the beginning of the algorithm DPsize, then we are able to derive analytically its value after DPsize terminates. Since this value of the inner counter depends on the query graph, we have to distinguish several cases. For chain, cycle, star, and clique queries, we denote by $I^{\text{chain}}_{\text{DPsize}}$, $I^{\text{cycle}}_{\text{DPsize}}$, $I^{\text{star}}_{\text{DPsize}}$, and $I^{\text{clique}}_{\text{DPsize}}$ the value of InnerCounter after termination of algorithm DPsize.

For chain queries, we then have: $I^{\text{chain}}_{\text{DPsize}}(n) =$

$$\begin{cases} 1/48(5n^4 + 6n^3 - 14n^2 - 12n) & n \text{ even} \\ 1/48(5n^4 + 6n^3 - 14n^2 - 6n + 11) & n \text{ odd} \end{cases}$$

For cycle queries, we have: $I^{\text{cycle}}_{\text{DPsize}}(n) =$

$$\begin{cases} \frac{1}{4}(n^4 - n^3 - n^2) & n \text{ even} \\ \frac{1}{4}(n^4 - n^3 - n^2 + n) & n \text{ odd} \end{cases}$$

For star queries, we have: $I^{\text{star}}_{\text{DPsize}}(n) =$

$$\begin{cases} 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + q(n) & n \text{ even} \\ 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + 1/4\binom{n-1}{(n-1)/2} + q(n) & n \text{ odd} \end{cases}$$

with $q(n) = n2^{n-1} - 5 * 2^{n-3} + 1/2(n^2 - 5n + 4)$. For clique queries, we have: $I^{\text{clique}}_{\text{DPsize}}(n) =$

$$\begin{cases} 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} - 1/4\binom{n}{n/2} + 1 & n \text{ even} \\ 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} + 1 & n \text{ odd} \end{cases}$$

```
DPsub
Input: a connected query graph with relations $R = \{R_0, \ldots, R_{n-1}\}$
Output: an optimal bushy join tree
for all $R_i \in R$ {
    BestPlan($\{R_i\}$) = $R_i$;
}
for $1 \leq i < 2^n - 1$ ascending {
    $S = \{R_j \in R | (\lfloor i/2^j \rfloor \, mod \, 2) = 1\}$
    if not (connected $S$) continue;      // *
    for all $S_1 \subset S$, $S_1 \neq \emptyset$ do {
        ++InnerCounter;
        $S_2 = S \setminus S_1$;
        if ($S_2 = \emptyset$) continue;
        if not (connected $S_1$) continue;
        if not (connected $S_2$) continue;
        if not ($S_1$ connected to $S_2$) continue;
        ++CsgCmpPairCounter;
        $p_1$ = BestPlan($S_1$);
        $p_2$ = BestPlan($S_2$);
        CurrPlan = CreateJoinTree($p_1$, $p_2$);
        if (cost(BestPlan($S$)) > cost(CurrPlan)) {
            BestPlan($S$) = CurrPlan;
        }
    }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan($\{R_0, \ldots, R_{n-1}\}$);
```

Figure 2: Algorithm DPsub

Note that $\binom{2n}{n}$ is in the order of $\Theta(4^n/\sqrt{n})$.

Proofs of the above formulas as well as implementation details for the algorithm `DPsize` can be found in [4].

2.2 Subset-Driven Enumeration

Fig. 2 presents the pseudocode for the algorithm `DPsub`. The algorithm first initializes the table `BestPlan` with all possible plans containing a single relation. Then, the main loop starts. It iterates over all possible non-empty subsets of $\{R_0, \ldots, R_{n-1}\}$ and constructs the best possible plan for each of them. The enumeration makes use of a bitvector representation of sets: The integer i induces the current subset S with its binary representation. Taken as bitvectors, the integers in the range from 1 to $2^n - 1$ exactly represent the set of all non-empty subsets of $\{R_0, \ldots, R_{n-1}\}$, including the set itself. Further, by starting with 1 and incrementing by 1, the enumeration order is valid for dynamic programming: for every subset, all its subsets are generated before the subset itself.

This enumeration is very fast, since increment by one is a very fast operation. However, the relations contained in S may not induce a connected subgraph of the query graph. Therefore, we must test for connectedness. The goal of the next loop over all subsets of S is to find the best plan joining all the relations in S. Therefore, S_1 ranges over all non-empty, strict subsets of S. This can be done very efficiently by applying the code snippet of Vance and Maier [9, 10]. Then, the subset of relations contained in S but not in S_1 is assigned to S_2. Clearly, S_1 and S_2 are disjoint. Hence, only connectedness tests have to be performed. Since we want to avoid cross products, S_1 and S_2 both must induce connected subgraphs of the query graph, and there must be a join predicate between a relation in S_1 and one in S_2. If these conditions are fulfilled, we can construct a plan `CurrPlan` by joining the plans associated with S_1 and S_2. If `BestPlan` does not contain a plan for the relations in S or the one it contains is more expensive than `CurrPlan`, we register `CurrPlan` with `BestPlan`.

For chain, cycle, star, and clique queries, we denote by $I^{\text{chain}}_{\text{DPsub}}$, $I^{\text{cycle}}_{\text{DPsub}}$, $I^{\text{star}}_{\text{DPsub}}$, and $I^{\text{clique}}_{\text{DPsub}}$ the value of `InnerCounter` after termination of algorithm `DPsub`.

For chains, we have

$$I^{\text{chain}}_{\text{DPsub}}(n) = 2^{n+2} - n^n - 3n - 4 \quad (1)$$

For cycles, we have

$$I^{\text{cycle}}_{\text{DPsub}}(n) = n2^n + 2^n - 2n^2 - 2 \quad (2)$$

For stars, we have

$$I^{\text{star}}_{\text{DPsub}}(n) = 2 * 3^{n-1} - 2^n \quad (3)$$

For cliques, we have

$$I^{\text{clique}}_{\text{DPsub}}(n) = 3^n - 2^{n+1} + 1 \quad (4)$$

The number of failures for the additional check can easily be calculated as $2^n - \#\texttt{csg}(n) - 1$, where $\#\texttt{csg}(n)$ denotes the number of non-empty connected subgraphs contained in the query graph.

2.3 Algorithm-Independent Results

2.3.1 *Definition of* #`csg` *and* #`ccp`

Consider a join ordering problem with n relations $R_0, \ldots, R_{n-1}$. We assume the query graph to be connected. Any subset S of $\{R_0, \ldots, R_{n-1}\}$ induces a subgraph of the query graph. If the subgraph induced by S is connected, we call S a *connected subset* or simply *connected*. For a given query graph G in n relations, we denote by $\#\texttt{csg}_G$ the number of non-empty connected subgraphs/subsets. For a given kind of query graph, every n uniquely determines a query graph. Since the kind of query graph will always be clear from the context, we write $\#\texttt{csg}(n)$.

Let S_1 and S_2 be two subsets of $\{R_0, \ldots, R_{n-1}\}$. If there is a join predicate between a relation in S_1 and another relation in S_2, we call S_1 and S_2 *connected*. Since we want to enumerate only bushy trees without cross products, we are only interested in connected sets S_1 and S_2 which are connected. Moreover, in order to form a valid join tree for relations in $S := S_1 \cup S_2$, S_1 and S_2 may not overlap, i.e. $S_1 \cap S_2 = \emptyset$.

Summarizing, during plan generation we are interested in pairs (S_1, S_2) where

- S_1 is a non-empty subset of $\{R_0, \ldots, R_{n-1}\}$, and

- S_2 is a non-empty subset of $\{R_0, \ldots, R_{n-1}\}$

such that

1. S_1 is connected,
2. S_2 is connected,
3. $S_1 \cap S_2 = \emptyset$,
4. there exist nodes $v_1 \in S_1$ and $v_2 \in S_2$ such that there is an edge between v_1 and v_2 in the query graph.

These conditions imply that both S_1 and S_2 are strict subsets of $\{R_0, \ldots, R_{n-1}\}$. Let us call a pair (S_1, S_2) fulfilling these conditions a *csg-cmp-pair*. Here, *csg* is the abbreviation of connected subgraph and *cmp* is the abbreviation of complement. The latter was chosen to emphasize the aspect of disjointness.

In the following, we are interested in (1) the number of connected, non-empty subsets and (2) the number of csg-cmp-pairs. Obviously, these numbers depend on the query graph. For csg-cmp-pairs, it is also important to note that if (S_1, S_2) is a csg-cmp-pair, then (S_2, S_1) is one as well. We denote the total number of csg-cmp-pairs including symmetric pairs by $\#$`ccp`. Given this, we immediately understand the `CsgCmpPairCounter` in the algorithms. After termination it gives us $\#$`ccp`. Ono and Lohman counted the number of csg-cmp-pairs by excluding symmetric pairs. Hence, their formulas return $\#$`ccp` divided by two. It is important to note that $\#$`ccp` only depends on the query graph. That is, the value of `CsgCmpPairCounter` (and, hence, for `OnoLohmanCounter`) after termination is the same for `DPsize`, `DPsub`, and the new algorithm `DPccp`. It is very important to note that for any correct dynamic programming algorithm $\#$`ccp` provides a lower bound on the number of calls to `CreateJoinTree`.

2.3.2 *Formulas for* $\#$`csg` *and* $\#$`ccp`

We analyse the join ordering problem for chain, cycle, and clique queries. For each kind of query graph, we calculate the number of connected subgraphs ($\#$`csg`) and the number of csg-cmp-pairs ($\#$`ccp`).

For a chain query in n relations, we have

$$\begin{aligned} \#\texttt{csg}(n) &= \frac{n(n+1)}{2} && (5)\\ \#\texttt{ccp}(n) &= \frac{(n+1)^3-(n+1)^2+2(n+1)}{3} && (6) \end{aligned}$$

This result is due to Ono and Lohman [5].

For a cycle query in n relations, we have

$$\begin{aligned} \#\texttt{csg}(n) &= n^2-n+1 && (7)\\ \#\texttt{ccp}(n) &= n^3-2n^2+n && (8) \end{aligned}$$

For star queries in n relations, we have

$$\begin{aligned} \#\texttt{csg}(n) &= 2^{n-1}+n-1 && (9)\\ \#\texttt{ccp}(n) &= (n-1)2^{n-2} && (10) \end{aligned}$$

This result is due to Ono and Lohman [5].

For clique queries in n relations, we have

$$\begin{aligned} \#\texttt{csg}(n) &= 2^n-1 && (11)\\ \#\texttt{ccp}(n) &= 3^n-2^{n+1}+1 && (12) \end{aligned}$$

This result is due to Ono and Lohman [5].

2.4 Sample Numbers

Fig. 3 contains tables with values produced by our formulas for input query graph sizes between 2 and 20. For different kinds of query graphs, it shows the number of csg-cmp-pairs ($\#$`ccp`). and the values for the inner counter after termination of `DPsize` and `DPsub` applied to the different query graphs.

Looking at these numbers, we observe the following:

- For chain and cycle queries, the `DPsize` soon becomes much faster than `DPsub`.
- For star and clique queries, the `DPsub` soon becomes much faster than `DPsize`.
- Except for clique queries, the number of csg-cmp-pairs is orders of magnitude less than the value of *InnerCounter* for all DP-variants.

From the latter observation we can conclude that in almost all cases the tests performed by both algorithms in their innermost loop fail. Both algorithms are far away from the theoretical lower bound given by $\#$`ccp`. This conclusion motivates us to derive a new algorithm whose *InnerCounter* value is equal to the number of csg-cmp-pairs.

3. THE NEW ALGORITHM DPCCP

3.1 Problem Statement

The algorithm `DPsub` solves the join ordering problem for a given subset S of relations by considering all pairs of disjoint subproblems which were already solved. Since the enumeration of subsets is very fast, this is a very efficient strategy if the search space is dense, e.g. for clique queries. However, if the search space is sparse, e.g. for chain queries, the `DPsub` algorithm considers many subproblems which are not connected and, therefore, are not relevant for the solution, i.e. the tests in the innermost loop fail for the majority of cases. The main idea of our algorithm `DPccp` is that it only considers pairs of connected subproblems. More precisely, the algorithm considers exactly the csg-cmp-pairs of a graph. Note that this is also the lower bound for any dynamic programming algorithm [9].

Thus, our goal is to efficiently enumerate all csg-cmp-pairs (S_1, S_2). Clearly, we want to enumerate every pair once and only once. Further, the enumeration must be performed in an order valid for dynamic programming. That is, whenever a pair (S_1, S_2) is generated, all non-empty subsets of S_1 and S_2 must have been generated before as a component of a pair. The last requirement is that the overhead for generating a single csg-cmp-pair must be constant or at most linear. This condition is necessary in order to beat `DPsize` and `DPsub`.

	Chain			Cycle		
n	#ccp	DPsub	DPsize	#ccp	DPsub	DPsize
2	1	2	1	1	2	1
5	20	84	73	40	140	120
10	165	3962	1135	405	11062	2225
15	560	130798	5628	1470	523836	11760
20	1330	4193840	17545	3610	22019294	37900
	Star			Clique		
n	#ccp	DPsub	DPsize	#ccp	DPsub	DPsize
2	1	2	1	1	2	1
5	32	130	110	90	180	280
10	2304	38342	57888	28501	57002	306991
15	114688	9533170	57305929	7141686	14283372	307173877
20	4980736	2323474358	59892991338	1742343625	3484687250	309338182241

Figure 3: Size of the search space for different graph structures

```
DPccp
Input: a connected query graph with relations $R = \{R_0, \dots, R_{n-1}\}$
Output: an optimal bushy join tree
for all $R_i \in R)$ {
    BestPlan($\{R_i\}$) = $R_i$;
}
for all csg-cmp-pairs $(S_1, S_2)$, $S = S_1 \cup S_2$ {
    ++InnerCounter;
    ++OnoLohmanCounter;
    $p_1$ = BestPlan($S_1$);
    $p_2$ = BestPlan($S_2$);
    CurrPlan = CreateJoinTree($p_1$, $p_2$);
    if (cost(BestPlan($S$)) > cost(CurrPlan)) {
        BestPlan($S$) = CurrPlan;
    }
    CurrPlan = CreateJoinTree($p_2$, $p_1$);
    if (cost(BestPlan($S$)) > cost(CurrPlan)) {
        BestPlan($S$) = CurrPlan;
    }
}
CsgCmpPairCounter = 2 * OnoLohmanCounter;
return BestPlan($\{R_0, \dots, R_{n-1}\}$);
```

Figure 4: Algorithm `DPccp`

If we meet all these requirements, the algorithm `DPccp` is easily specified: iterate over all csg-cmp-pairs (S_1, S_2) and consider joining the best plans associated with them. Figure 4 shows the pseudocode. The first steps of an example enumeration are shown in Figure 5. Thick lines mark the connected subsets while thin lines mark possible join edges. Note that the algorithm explicitly exploits join commutativity. This is due to our enumeration algorithm developed below. If (S_1, S_2) is a csg-cmp-pair, then either (S_1, S_2) or (S_2, S_1) will be generated, but never both of them. An alternative is to modify `CreateJoinTree` to take care of commutativity.

The rest of this section is organized as follows. The next subsection discusses an algorithm enumerating non-empty connected subsets S_1 of $\{R_0, \dots, R_{n-1}\}$. Subsection 3.3 then shows how to enumerate the complements S_2 such that (S_1, S_2) is a csg-cmp-pair. Finally, Subsection 3.4 contains the correctness proofs for the algorithms.

3.2 Enumerating Connected Subsets

Scanning the literature for algorithms enumerating all connected subgraphs, we found only two algorithms. The first one turned out to be highly inefficient [6]. From the second one, we took the basic idea of using a breadth-first numbering of the nodes in the query graph [8]. For a number of reasons, we could not use the algorithm directly: the algorithm was flawed; it maintained a set of all generated connected subgraphs and had to test every generated one against those already generated in order to avoid duplicates; it did not generate the subgraphs in an order expedient for dynamic programming.

Let us start the exposition by fixing some notations. Let $G = (V, E)$ be an undirected graph. For a node $v \in V$ define the *neighborhood* $\mathcal{N}(v)$ of v as $\mathcal{N}(v) := \{v' | (v, v') \in E\}$. For a subset $S \subseteq V$ of V we define the *neighborhood* of S as $\mathcal{N}(S) := \cup_{v \in S} \mathcal{N}(v) \setminus S$. The neighborhood of a set of nodes thus consists of all nodes reachable by a single edge. Note that for all $S, S' \subset V$ we have $\mathcal{N}(S \cup S') = (\mathcal{N}(S) \cup \mathcal{N}(S')) \setminus (S \cup S')$. This allows for an efficient bottom-up calculation of neighborhoods.

The following statement gives a hint on how to construct an enumeration procedure for connected subsets. Let S be a connected subset of an undirected graph G and S' be any subset of $\mathcal{N}(S)$. Then $S \cup S'$ is connected. As a consequence, a connected subset can be enlarged by adding any subset of its neighborhood.

We could generate all connected subsets as follows. For every node $v_i \in V$ we perform the following enumeration steps: First, we emit $\{v_i\}$ as a connected subset. Then, we expand $\{v_i\}$ by calling a routine that extends a given connected set to bigger connected sets. Let the routine be called with some connected set S. It then calculates the neighborhood $\mathcal{N}(S)$. For every non-empty subset $N \subseteq \mathcal{N}(S)$, it emits $S' = S \cup N$ as a further connected subset and recursively calls itself with S'. The problem with this routine is that it produces duplicates.

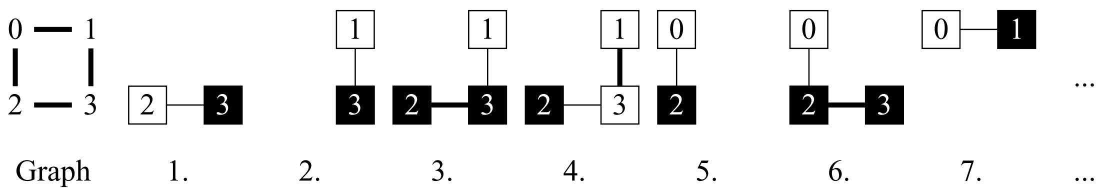

Figure 5: Enumeration Example for `DPccp`

This is the point where the breadth-first numbering comes into play. Let $V = \{v_0, \ldots, v_{n-1}\}$, where the indices are consistent with a breadth-first numbering produced by a breadth-first search starting at node v_0 [1] (see Section 3.4.1 for a formal definition). The idea is to use the numbering to define an enumeration order: In order to avoid duplicates, the algorithm enumerates connected subgraphs for every node v_i, but restricts them to contain no v_j with $j < i$. Using the definition $\mathcal{B}_i = \{v_j | j \leq i\}$, the pseudocode looks as follows:

```
EnumerateCsg
Input: a connected query graph $G = (V, E)$
Precondition: nodes in $V$ are numbered according to a breadth-first search
Output: emits all subsets of $V$ inducing a connected subgraph of $G$
for all $i \in [n-1, \ldots, 0]$ descending {
    emit $\{v_i\}$;
    EnumerateCsgRec($G$, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec($G$, $S$, $X$)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec($G$, $(S \cup S')$, $(X \cup N)$);
}
```

Let us consider an example. Figure 6 contains a query graph whose nodes are numbered in a breadth-first fashion. The calls to `EnumerateCsgRec` are contained in the table in Figure 7. In this table, S and X are the arguments of `EnumerateCsgRec`. N is the local variable after its initialization. The column `emit/`S contains the connected subset emitted, which then becomes the argument of the recursive call to `EnumerateCsgRec` (labelled by $\rightarrow$). Since listing all calls is too lengthy, only a subset of the calls is listed.

3.3 Enumerating Complements of Connected Subgraphs

Generating the connected subsets is an important first step but clearly not sufficient: we have to generate all csg-cmp-pairs. The basic idea to do so is as follows. Algorithm `EnumerateCsg` is used to create the first component S_1 of every csg-cmp-pair. Then, for each such S_1, we generate all its complement components S_2. This can be done by calling

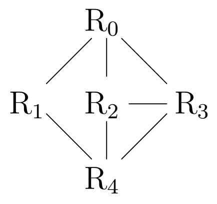

Figure 6: Sample graph to illustrate `EnumerateCsgRec`

EnumerateCsgRec			
S	X	N	emit/S
$\{4\}$	$\{0,1,2,3,4\}$	$\emptyset$	
$\{3\}$	$\{0,1,2,3\}$	$\{4\}$	
			$\{3,4\}$
$\{2\}$	$\{0,1,2\}$	$\{3,4\}$	
			$\{2,3\}$
			$\{2,4\}$
			$\{2,3,4\}$
$\{1\}$	$\{0,1\}$	$\{4\}$	
			$\{1,4\}$
$\rightarrow \{1,4\}$	$\{0,1,4\}$	$\{2,3\}$	
			$\{1,2,4\}$
			$\{1,3,4\}$
			$\{1,2,3,4\}$
$\{0\}$	$\{0\}$	$\{1,2,3\}$	
			$\{0,1\}$
			$\{0,2\}$
			$\{0,3\}$
			$\{0,1,2\}$
			$\{0,1,3\}$
			$\{0,2,3\}$
			$\{0,1,2,3\}$
$\rightarrow \{0,1\}$	$\{0,1,2,3\}$	$\{4\}$	
			$\{0,1,4\}$
$\rightarrow \{0,2\}$	$\{0,1,2,3\}$	$\{4\}$	
			$\{0,2,4\}$

Figure 7: Call sequence for Figure 6

`EnumerateCsgRec` with the correct parameters. Remember that we have to generate every csg-cmp-pair once and only once.

To achieve this, we use a similar technique as for connected subsets, using the breadth-first numbering to define an enumeration order: we consider only sets S_2 in the complement of S_1 (with (S_1, S_2) being a csg-cmp-pair) such that S_2 contains only v_j with j larger than any i with $v_i \in S_1$. This avoids the generation of duplicates.

We need some definitions to state the actual algorithm. Let $S_1 \subseteq V$ be a non-empty subset of V. Then, we define $\min(S_1) := \min(\{i | v_i \in S_1\})$. This is used to extract the starting node from which S_1 was constructed (see Lemma 9). Let $W \subset V$ be a non-empty subset of V. Then, we define $\mathcal{B}_i(W) := \{v_j | v_j \in W, j \leq i\}$. Using this notation, the algorithm to construct all S_2 for a given S_1 such that (S_1, S_2) is a csg-cmp-pair looks as follows:

```
EnumerateCmp
Input: a connected query graph $G = (V, E)$, a connected subset $S_1$
Precondition: nodes in $V$ are numbered according to a breadth-first search
Output: emits all complements $S_2$ for $S_1$ such that $(S_1, S_2)$ is a csg-cmp-pair
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending $i$) {
    emit $\{v_i\}$;
    EnumerateCsgRec($G$, $\{v_i\}$, $X \cup N$);
}
```

Algorithm `EnumerateCmp` considers all neighbors of S_1. First, they are used to determine those S_2 that contain only a single node. Then, for each neighbor of S_1, it recursively calls `EnumerateCsgRec` to create those S_2 that contain more than a single node. Note that here both nodes concerning the enumeration of S_1 ($\mathcal{B}_{\min(S_1)} \cup S_1$) and nodes concerning the enumeration of S_2 (N) have to be considered in order to guarantee a correct enumeration. Otherwise the combined algorithm would emit (commutative) duplicates.

Let us consider an example for algorithm `EnumerateCmp`. The underlying graph is again the one shown in Fig. 6. Assume `EnumerateCmp` is called with $S_1 = \{R_1\}$. In the first statement, the set $\{R_0, R_1\}$ is assigned to X. Then, the neighborhood is calculated. This results in

$$N = \{R_0, R_4\} \setminus \{R_0, R_1\} = \{R_4\}.$$

Hence, $\{R_4\}$ is emitted and together with $\{R_1\}$, it forms the csg-cmp-pair $(\{R_1\}, \{R_4\})$. Then, the recursive call to `EnumerateCsgRec` follows with arguments G, $\{R_4\}$, and $\{R_0, R_1, R_4\}$. Subsequent `EnumerateCsgRec` generates the connected sets $\{R_2, R_4\}$, $\{R_3, R_4\}$, and $\{R_2, R_3, R_4\}$, giving three more csg-cmp-pairs.

3.4 Correctness Proof

3.4.1 Preliminaries

The correctness of `DPccp` follows if the csg-cmp-pairs are enumerated correctly, as it simply enumerates all possible pairs and fills the DP table accordingly. Therefore, we only have to prove the correctness of the functions `EnumerateCsg`, `EnumerateCsgRec` and `EnumerateCmp`. The rest of this section is independent of the join ordering problem. Thus, we concentrate on undirected graphs.

Given a connected undirected graph $G = (V, E)$, we want to enumerate all vertices $V' \subseteq V$, such that $G' = (V', E_{|V'})$ is a connected subgraph of G. Thereby, $E_{|V'} = \{(v, v') \in E | v, v' \in V'\}$. We denote the direct neighbors of a node v by $N(v)$ defined as

$$\mathcal{N}(v) = \{v' \in V | (v, v') \in E\}.$$

Indirect neighbors are collected into sets $N_i(v)$, which contain the i-th generation of neighbors:

$$\begin{aligned} \mathcal{N}_0(v) &= \{v\} \\ \mathcal{N}_1(v) &= \mathcal{N}(v) \\ \mathcal{N}_{i+1}(v) &= (\cup_{v' \in \mathcal{N}_i(v)} \mathcal{N}(v')) \setminus (\cup_{j=0\ldots i} \mathcal{N}_j(v)) \end{aligned}$$

If a vertex $v \in V$ has a label, the label is determined by $L(v)$. The labels will be unique, therefore we can identify a vertex by its label: $v = v_{L(v)}$.

We assume that the graph G contains no self-cycles, i.e. $\nexists v \in V : (v, v) \in E$. If such edges exist, they can be safely removed as they do not affect the connected subsets. Further we assume that the vertices in the graph are labeled in a breadth-first manner. That is, we demand that

- there exists one vertex $v_0 \in V$ that has the label 0,
- the vertices in $\mathcal{N}_1(v_0)$ have labels in $[\ 1, |\mathcal{N}_1(v_0)|\ ]$,
- the vertices in $\mathcal{N}_k(v_0)$ have labels in $[\ \sum_{i=0}^{k-1} |\mathcal{N}_i(v_0)|\ ,\ \sum_{i=0}^{k} |\mathcal{N}_i(v_0)|\ ]$.

3.4.2 Correctness of `EnumerateCsg`

Lemma 1. *Algorithm* `EnumerateCsg` *terminates if* G *is a finite graph.*

Proof. `EnumerateCsg` performs a finite number of loop iterations ($|V|$). In each iteration, it constructs a finite set and passes it as an argument to `EnumerateCsgRec`. Thus, `EnumerateCsg` terminates if `EnumerateCsgRec` terminates for all inputs.

`EnumerateCsgRec` is called with three arguments, G, S and X. As $G = (V, E)$ is a finite graph and $S \subseteq V \wedge X \subseteq V$ (line 3 in `EnumerateCsg`, lines 1 and 6 in `EnumerateCsgRec`), S and X are also finite. In each recursion, `EnumerateCsgRec` considers the neighbors $N \subseteq V$ of S, ignoring vertices in X. It then evaluates each non-empty subset of N, calling `EnumerateCsgRec` recursively, enlarging X by N. As $X \subseteq V$ and X is enlarged by each call, the recursion depth of `EnumerateCsgRec` is limited by $|V|$. □

Lemma 2. *Algorithm* `EnumerateCsg` *enumerates only connected components.*

PROOF. By induction over the recursion depth n.

Base Case: $n = 0$
`EnumerateCsg` starts the enumeration with single vertices, which are connected components (lines 2-3).

Induction hypothesis: recursion depth n enumerates only connected components and passes them as parameter S to recursion depth $n + 1$

Induction step:
`EnumerateCsgRec` at recursion level $n+1$ is called with a connected component S (IH) and considers only vertices that are connected to vertices in S (N, line 1). As any vertex in N is directly connected to at least one vertex in S, any subset of N can be added to S to form a connected component (lines 2-3, 5-6). The claim follows. $\square$

LEMMA 3. *Given a connected, undirected graph* $G = (V, E)$*, a vertex* $v \in V$*, a natural number* $n \geq 0$*, and* $V'_n = \cup_{0 \leq i \leq n} \mathcal{N}_i(v)$*. Then* $(V'_n, E_{|V'_n})$ *is a connected component.*

PROOF. By induction over n.

Base Case: $n = 0$
$V'_0 = \mathcal{N}_0(v) = \{v\}$. Thus, $(V'_0, E_{|V'_0})$ is a connected component.

Induction hypothesis: $(V'_n, E_{|V'_n})$ is a connected component for a given, fixed n.

Induction step: $n \rightarrow n + 1$
Per definition, $V'_{n+1} = V'_n \cup \mathcal{N}_{n+1}(v)$. $(V'_n, E_{|V'_n})$ is a connected component (IH), $\mathcal{N}_n(v) \subseteq V'_n$ (def.), all vertices in $\mathcal{N}_{n+1}(v)$ are connected to at least one vertex in $\mathcal{N}_n(v)$ (def.) It follows that $(V'_{n+1}, E_{|V'_{n+1}})$ is a connected component. $\square$

LEMMA 4. *Given a connected, undirected graph* $G = (V, E)$ *and a vertex* $v \in V$*. Then* $\exists n \geq 0$ *such that* $\forall_{0 \leq i \leq n} \mathcal{N}_i(v) \neq \emptyset$ *and* $\forall_{i > n} \mathcal{N}_i(v) = \emptyset$*.*

PROOF. From the definition of $\mathcal{N}_{i+1}(v)$ follows that if $(\mathcal{N}_i(v) = \emptyset) \implies (\mathcal{N}_{i+1}(v) = \emptyset)$. Further it follows from the definition of $\mathcal{N}_i$ that $\mathcal{N}_0(v) \neq \emptyset$, $\forall_i \mathcal{N}_i(v) \subseteq V$, and $\forall_{j<i} \mathcal{N}_i(v) \cap \mathcal{N}_j(v) = \emptyset \implies \mathcal{N}_{|V|}(v) = \emptyset$. $\Rightarrow n \in [0, |V|[$. $\square$

LEMMA 5. *Given a connected, undirected graph* $G = (V, E)$*,* $|V| > 1$ *and a set of vertices* $V' \subseteq V$ *such that* $(V', E_{|V'})$ *is a connected component. Then* $\exists v \in V'$ *such that* $(V' \setminus \{v\}, E_{|V' \setminus \{v\}})$ *is a connected component.*

PROOF. In the following, we consider $G' = (V', E_{|V'})$ as the base to compute $\mathcal{N}(v)$ and $\mathcal{N}_i(v)$ (G' is a connected undirected graph). Choose an arbitrary $v_0 \in V'$ and a natural number n such that $\mathcal{N}_n(v_0) \neq \emptyset \wedge \mathcal{N}_{n+1}(v_0) = \emptyset$ (Lemma 4). Note that $n > 0$ as $|V'| > 1$ and that $\cup_{0 \leq i \leq n} \mathcal{N}_i(v_0) = V'$. Now any $v \in \mathcal{N}_n(v_0)$ can be removed: $\cup_{0 \leq i < n} \mathcal{N}_i(v_0)$ forms a connected component (Lemma 3) and all vertices in $\mathcal{N}_n(v_0)$ are connected to at least one vertex in $\mathcal{N}_{n-1}(v_0)$. Since $n > 0$ and $\mathcal{N}_n(v_0) \neq \emptyset$ if follows that $\forall_{v \in \mathcal{N}_n(v_0)} (V' \setminus \{v\}, E_{|V' \setminus \{v\}})$ is a connected component. $\square$

LEMMA 6. *When* `EnumerateCsgRec` *is called with additional vertices, it enumerates at least the same components as without the vertices. More formally:*
$\{V \cup A | (V, E)$ *enumerated by* `EnumerateCsgRec`$(G, S, X)\} \subseteq$ $\{V | (V, E)$ *enumerated by* `EnumerateCsgRec`$(G, S \cup A, X)\}$

PROOF. In line 1: $N \subseteq \mathcal{N}(S \cup A)$. Therefore, at least the same combinations are enumerated in the first step. The same in further recursions, the increase of X does not affect the search space as all additional vertices in X are already examined in the first step. $\square$

LEMMA 7. *Algorithm* `EnumerateCsg` *enumerates all connected components consisting of a single vertex.*

PROOF. Line 1 iterates over all vertices, line 2 emits a graph consisting of the vertex. $\square$

LEMMA 8. *Algorithm* `EnumerateCsg` *enumerates all connected components.*

PROOF. By contradiction. We assume that not all connected components are enumerated. Thus $\exists V' \subseteq V \wedge V \neq \emptyset$ such that $(V', E_{|V'})$ is a connected component and V' is not enumerated. If several such V' exist, we choose V' such that $|V'|$ is minimal. Lemma 7 implies that $|V'| > 1$. Lemma 5 implies that $\exists v' \in V' : (V' \setminus \{v'\}, E_{|V' \setminus \{v'\}})$ is a connected component. As V' was chosen to be minimal, $(V' \setminus \{v'\}, E_{|V' \setminus \{v'\}})$ was enumerated.

Case 1: v' appeared in N during the enumeration of $V' \setminus \{v'\}$. This is a contradiction to the assumption that V' was not enumerated (Line 2, Line 5, Lemma 6).

Case 2: v' did not appear in N during the enumeration of $V' \setminus \{v'\}$. Since v' is connected to $V' \setminus \{v'\}$, it must have been excluded, i.e. $L(v') < min(\{L(v) | v \in V' \setminus \{v'\}\})$. Then `EnumerateCsg` will enumerate V' when selecting v' as the start vertex (the constructive proof of this claim is trivial). $\square$

LEMMA 9. *If* V' *and* V'' *are both enumerated and* $min(\{L(v) | v \in V'\}) = min(\{L(v) | v \in V''\})$*,* V' *and* V'' *are enumerated using the same start vertex.*

PROOF. `EnumerateCsg` iterates over all vertices (line 1) and starts the enumeration with a connected component consisting of just this vertex (lines 2-3). All vertices with a label smaller than the start vertex are excluded (line 3). Thus, the smallest label determines the start vertex. $\square$

LEMMA 10. *Algorithm* `EnumerateCsg` *enumerates all connected components only once.*

PROOF. By contradiction. We assume that $\exists V' \subseteq V$ that is enumerated at least twice. If multiple such V' exist, we choose V' such that $|V'|$ is minimal.

Case 1: $|V'| = 1$. As discussed in Lemma 7, `EnumerateCsg` enumerates all connected components consisting of a single vertex. As `EnumerateCsgRec` increases components by a non-empty, disjoint set, V' must have been enumerated by `EnumerateCsg`. But `EnumerateCsg` performs a single loop over all vertices, thus V' cannot have been produced twice.

Case 2: $|V'| > 1$. V' is enumerated by `EnumerateCsgRec`. Lemma 9 implies that both enumerations of V' started with the same vertex. Hence, the first call to `EnumerateCsgRec` is identical. Especially X is the same.

A single invocation of `EnumerateCsgRec` (without the recursive call) does not produce duplicates (lines 2-3 iterate only once). Hence, V' cannot be enumerated twice by a single call. V' cannot be enumerated by two different calls to `EnumerateCsgRec` with the same parameters, as $|V'|$ is minimal (otherwise S would be smaller and also be enumerated twice). Thus, $\exists S_1, S_2, X_1, X_2 \subseteq V$ such that $S_1 \neq S_2$, S_1, S_2, X_1, X_2 are constructed by `EnumerateCsgRec` starting from the same start vertex and both `EnumerateCsgRec` (G, S_1, X_1) and `EnumerateCsgRec`(G, C_2, X_2) enumerate V'. Hence, $(V' \setminus S_1) \cup X_1 = \emptyset \wedge (V' \setminus S_2) \cup X_2 = \emptyset$.

As both enumerations started with the same vertex, they have a common invocation path. As $S_1 \neq S_2$, there exists a invocation of `EnumerateCsgRec`, that recursively calls `EnumerateCsgRec` with S_1' and S_2' ($S_1' \neq S_2'$), which finally lead to S_1 and S_2 respectively.

Note that the exclusion filter X constructed in line 6 is the same for S_1' and S_2', but $S_1' \neq S_2'$. This implies $\exists v \in (S_1' \cup S_2') : v \notin (S_1' \cap S_2) \wedge v \in X$. From this, we can conclude that $((v \in S_1 \wedge v \notin S_2) \vee (v \notin S_1 \wedge v \in S_2)) \wedge (v \in X)$ and, hence, $v \in V' \wedge v \notin V'$. □

LEMMA 11. *If $V' \subset V'', n = |V''| - |V'| - 1$ and both $(V', E_{|V'})$ and $(V'', E_{|V''})$ are connected components, then $\exists V_1 \dots V_n$ such that $V' \subset V_1, V_i \subset V_{i+1}, V_n \subset V''$ and $(V_i, E_{|V_i})$ is a connected component $\forall 1 \leq i \leq n$.*

PROOF. By induction over n.

Base case: n=0. The sequence $V_1 \dots V_n$ is empty, i.e. it exists.

Induction hypothesis: A suitable sequence $V_1 \dots V_n$ exists for a given fixed n.

Induction step: $n \rightarrow n + 1$
Choose an arbitrary $v \in V'' \setminus V'$ that is connected to V'. As $V' \subset V''$ and V'' is a connected component, such a v exists. Choose $V_1 = V' \cup \{v\}$ as the first entry in the sequence. Now $n = |V''| - |V_1| - 1 \Rightarrow$ (IH) a sequence between V_1 and V'' exists, which can be used as $V_2 \dots V_n$. □

LEMMA 12. *If $V' \subset V''$ and both $(V', E_{|V'})$ and $(V'', E_{|V''})$ are connected components,* `EnumerateCsg` *enumerates $(V', E_{|V'})$ before $(V'', E_{|V''})$.*

PROOF. By contradiction. We assume that V'' is enumerated before V'. Using Lemma 11 we know that if such V' and V'' exist, at least one occurrence must have the property $|V'| + 1 = |V''|$. Therefore we can limit ourselves to this case.

Case 1: $min(\{L(v)|v \in V''\}) < min(\{L(v)|v \in V'\})$
V' and V'' are enumerated during different loop passes in `EnumerateCsg`, as `EnumerateCsg` determines the vertex with the lowest label in each connected component. This is a contradiction to the assumption that V'' is enumerated before V'', as the vertices are selected with the greatest label first (line 1).

Case 2: $min(\{L(v)|v \in V''\}) > min(\{L(v)|v \in V'\})$
This is a contradiction to $V' \subset V''$

Case 3: $min(\{L(v)|v \in V''\}) = min(\{L(v)|v \in V'\})$
As V' and V'' have the same minimal vertex id, they are enumerated during the same loop pass in `EnumerateCsg`. We now name the single vertex difference between V' and V'' v and consider how V' is enumerated.

Case 3.1: $v \notin X$ when enumerating V'
When enumerating V', v is not suppressed. This means that either a further loop iteration (line 2) or a recursive call (line 5) will enumerate V''. This is a contradiction, as we assumed that V'' was already enumerated and the algorithm produces no duplicates (Lemma 10). Note that V'' cannot be enumerated first in the same loop, as the loops enumerate subsets first (line 2).

Case 3.2: $v \in X$ when enumerating V'
$v \in X \Rightarrow v \in N$ in one recursion step (v is not the start vertex). This recursion step will also enumerate V'' later on ($v \in N$, X is unchanged), this is a contradiction to the assumption that V'' is enumerated first and to Lemma 10. □

THEOREM 1. *Algorithm* `EnumerateCsg` *is correct.*

PROOF. The theorem follows immediately from Lemma 1, Lemma 2, Lemma 8, Lemma 10, and Lemma 12. □

3.4.3 Correctness of `EnumerateCmp`

Besides enumerating the connected components themselves, the `DPccp` algorithm requires enumerating all connected components in the adjacent complement of the graph. More formally, given a connected graph $G = (V, E)$ and $V' \subseteq V$ such that $(V', E_{|V'})$ is a connected component, enumerate all $V'' \subseteq V \setminus V'$ such that $(V'', E_{|V''})$ and $(V' \cup V'', E_{|V' \cup V''})$ are connected components.

The algorithms presented suppress duplicates. This means that if V'' is enumerated for a given V', V' will not be enumerated if V'' is given as a *primary* connected component (i.e. as a first component in a csg-cmp-pair). Furthermore, a V'' is only enumerated if it was already enumerated as a primary connected component. This allows us to define a total ordering between disjoint connected components that matches the enumeration order used in `EnumerateCsg`:

$$V'' < V' \Leftrightarrow min(\{L(v)|v \in V'\}) < min(\{L(v)|v \in V''\}).$$

Using this ordering, we only enumerate V'' for V' if $V' < V''$. Note that the following condition holds:

$$V_1 \subset V_2 \wedge V_1 < V_3 \Rightarrow V_2 < V_3.$$

This allows terminating the construction of connected components early if any sub-component is already less than the primary connected component.

Enumeration of the connected components of the complement is similar to the normal enumeration algorithm. The algorithm performs a recursive construction by starting with the neighbours of the primary connected component. Each neighbor is selected, the ordering condition is checked, and the recursive construction starts with the single vertex. Note that the total ordering is enforced implicitly: The exclusion set X contains all vertices that would result in a violation.

3.4.4 Proofs

LEMMA 13. *Algorithm* `EnumerateCmp` *terminates if G is a finite graph.*

PROOF. `EnumerateCmp` performs a finite number of loop iterations ($|N_p| < |V|$). In each iteration it constructs a finite set and passes it as argument to `EnumerateCsgRec`. Thus `EnumerateCmp` terminates if `EnumerateCsgRec` terminates for all inputs. `EnumerateCsgRec` terminates as shown in Lemma 1. □

LEMMA 14. *Algorithm* `EnumerateCmp` *enumerates all connected components consisting of a single vertex (that satisfy the ordering and are connected to p).*

PROOF. The exclusion list in line 1 consists only of vertices that are either $\in S$ or would be rejected anyway, due to the ordering. Thus, the neighbourhood constructed in line 2 consists of all vertices adjacent to S that satisfy the ordering. They are used to construct connected components with a single vertex in lines 3-4. □

LEMMA 15. *Algorithm* `EnumerateCmp` *enumerates only adjacent connected components in the complement.*

PROOF. All connected components are constructed starting with a single vertex, which is adjacent (see proof of Lemma 14). The further construction is done by calling `EnumerateCsgRec`, which only constructs connected components (proof of Lemma 2). The exclusion list X constructed in line 1 allows only connected components from the complement. □

LEMMA 16. *Algorithm* `EnumerateCmp` *enumerates all adjacent connected components in the complement (that satisfy the ordering).*

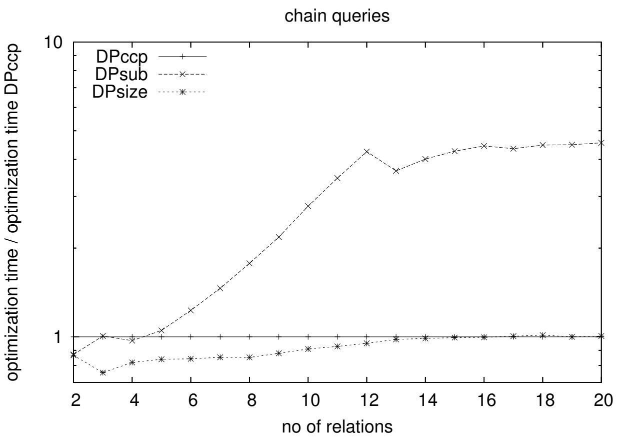

Figure 8: Relative performance for chain queries

PROOF. As shown in Lemma 14, all connected components consisting of a single vertex are enumerated. The proof for larger connected components is analogous to the proof of Lemma 8 as `EnumerateCsgRec` is reused.

Note that Lemma 5, which is used in the proof, can be lifted to adjacent connected components: The constructive proof shows that actually two vertices could be removed, either v_0 or a vertex in $\mathcal{N}_n(v_0)$. Therefore, it is possible to a remove a vertex such that the connected component is still adjacent. □

LEMMA 17. *Algorithm* `EnumerateCmpl` *enumerates connected components only once.*

PROOF. See the proof of Lemma 10. `EnumerateCmp` determines the start vertex (vertices in N are only selected by `EnumerateCmp`), the recursive phase `EnumerateCsgRec` increases the connected component. As the recursive steps are identical, the proof can be reused. □

THEOREM 2. *Algorithm* `EnumerateCmp` *is correct.*

PROOF. The theorem follows immediately from Lemma 13, Lemma 15, Lemma 16, and Lemma 17. □

4. EVALUATION

While the analytical results of Section 2 already imply the performance characteristics of the different algorithms, we performed experiments to measure the costs of an actual implementation. We compared the execution time of the different algorithms for varying query graphs.

Figures 8 to 11 show the relative performance of the different algorithms for chain, cycle, star, and clique graphs. As the optimization time varies greatly with the query size, all performance numbers are given relative to `DPccp`, e.g. the optimization time of `DPccp` is always 1. Sample absolute running times are given in Figure 12 (for more see [4]).

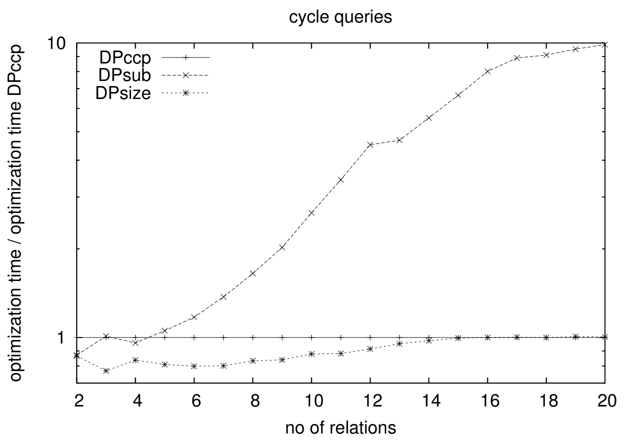

Figure 9: Relative performance for cycle queries

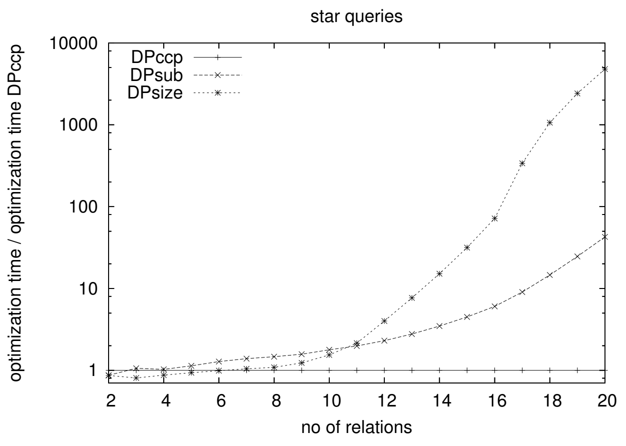

Figure 10: Relative performance for star queries

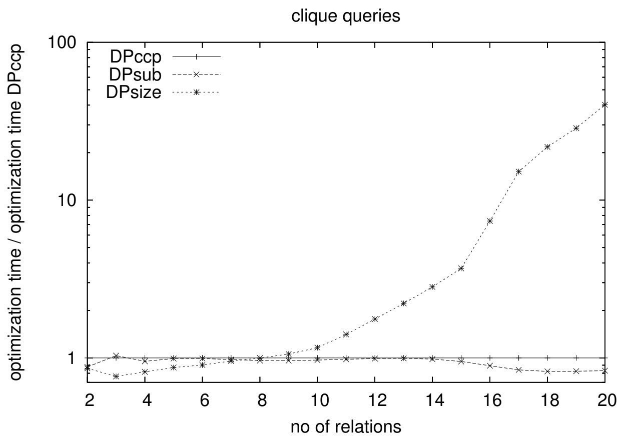

Figure 11: Relative performance for clique queries

n	`DPsize`	`DPsub`	`DPccp`
	chain queries		
5	7.7e-6	9.7e-6	9.2e-6
10	5.8e-5	0.00018	6.4e-5
15	0.0013	0.0056	0.0013
20	0.048	0.22	0.048
	cycle queries		
5	1.1e-5	1.5e-5	1.4e-5
10	0.0001	0.00031	0.00012
15	0.001	0.01	0.0015
20	0.049	0.47	0.048
	star queries		
5	9.8e-6	1.2e-5	1.0e-5
10	0.00069	0.0008	0.00044
15	0.71	0.1	0.022
20	4791	42.7	1.00
	clique queries		
5	2.1e-5	2.4e-5	2.4e-5
10	0.0058	0.0048	0.005
15	4.6	1.2	1.3
20	21294	439	529

Figure 12: Sample absolute running time (s)

As indicated by the theoretical investigations of Section 2, `DPsize` and `DPccp` are superior to `DPsub` for chain and cycle queries. For star and clique queries, `DPsub`, and `DPccp` are superior to `DPsize`.

The experiments show that overall, `DPccp` is the best algorithm. Independently of the query graph, it is either the fastest or nearly the fastest algorithm. While the other algorithms only perform well for certain graph structures, `DPccp` always adapts to the query graph. There is some overhead caused for the more complex enumeration, but this overhead is usually small. Only for clique queries the overhead shows somewhat, as the enumeration of `DPsub` is extremely efficient in a dense search space. Nonetheless, the overhead is always below 30%.

For star queries, `DPccp` is highly superior to both `DPsize` and `DPsub`. As the query size increases, the other algorithms become slower by multiple orders of magnitude. Furthermore, since star queries are of high practical importance in data warehouses and clique queries do not have any practical value, `DPccp` is the algorithm of choice.

5. CONCLUSION

The main contributions of this paper are the following: we analyzed the complexity of `DPsize` and `DPsub` both analytically and experimentally. The conclusions drawn in both cases are that

1. `DPsize` is superior to `DPsub` for chain and cycle queries, and

2. `DPsub` is superior to `DPsize` for star and clique queries.

We then designed an algorithm that efficiently enumerates csg-cmp-pairs in an order valid for dynamic programming.

This can be used to derive the join ordering algorithm `DPccp`, which is highly superior to `DPsize` and `DPsub`. `DPccp` should thus be the algorithm of choice when implementing a plan generator.

Acknowledgment. We thank Simone Seeger for her help preparing this manuscript.

6. REFERENCES

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001. 2nd Edition.

[2] P. Gassner, G. Lohman, and K. Schiefer. Query optimization in the IBM DB2 family. *IEEE Data Engineering Bulletin*, 16:4–18, Dec. 1993.

[3] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. on Database Systems*, 25(1):43–82, 2000.

[4] G. Moerkotte. Dp-counter analytics. Technical Report 2, University of Mannheim, 2006.

[5] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 314–325, 1990.

[6] G. Rücker and C. Rücker. Automatic enumeration of all connected subgraphs. *Commun. Math. Comput. Chem.*, 41:145–149, 2000.

[7] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, 1979.

[8] A. R. Sharafat and O. R. Ma'rouzi. A novel and efficient algorithm for scanning all minimal cutsets of a graph. *ArXiv Mathematics e-prints*, 2002.

[9] B. Vance. *Join-order Optimization with Cartesian Products*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1998.

[10] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, 1996.