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

February 25, 2025

1 Introduction

Problem: Find the optimal join order

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

for all R_i ∈ R {
    BestPlan({R_i}) = R_i;
}
for all 1 < s ≤ n ascending // size of plan
for all 1 ≤ s_1 < s { // size of left subplan
    s_2 = s − s_1; // size of right subplan
    for all S_1 ⊂ R : |S_1| = s_1
            S_2 ⊂ R : |S_2| = s_2 {
        ++InnerCounter;
        if (∅ ≠ S_1 ∩ 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 ∪ S_2)) > cost(CurrPlan)) {
            BestPlan(S_1 ∪ S_2) = CurrPlan;
        }
    }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan({R_0, ..., R_{n-1}});
```

Figure 1: Algorithm `DPsize`

2 Algorithms and Analysis

2.1 Size-Driven Enumeration

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:

1. the sets of relations contained in P_1 and P_2 do not overlap
2. there is a join predicate connecting a relation in P_1 with a relation in P_2

The algorithm `DPsize` can be made more efficient in case of $s_1 = s_2$. Assume that plans of equal size are repenseted as a linked list. If $s_1 = s_2$,

then it is possible to iterate through the list for retriving all plans p_1. For p_2 we consider the plans succeeding $p_1$in the list. Thus, the complexity can be decreased to $s_1 \times s_2/2$.

2.2 Subset-Driven Enumeration

```
DPsub
Input: a connected query graph with relations R =
{R_0, ..., R_{n-1}}
Output: an optimal bushy join tree
for all R_i ∈ R {
    BestPlan({R_i}) = R_i;
}
for 1 ≤ i < 2^n − 1 ascending {
    S = {R_j ∈ R|(⌊i/2^j⌋ mod 2) = 1}
    if not (connected S) continue;      // *
    for all S_1 ⊂ S, S_1 ≠ ∅ do {
        ++InnerCounter;
        S_2 = S \ S_1;
        if (S_2 = ∅) 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, ..., R_{n-1}});
```

Figure 2: Algorithm `DPsub`

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. 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 $\{R_0, \dots, R_{n-1}\}$. 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

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

- S_1 is a non-empty subset of $\{R_0, \dots, R_{n-1}\}$
- S_2 is a non-empty subset of $\{R_0, \dots, R_{n-1}\}$

s.t.

1. S_1 is connected
2. S_2 is connected
3. $S_1 \cap S_2$
4. there exists nodes $v_1 \in S_1$ and $v_2 \in S_2$ s.t. there is an edge between v_1 and v_2 in the query graph

Let's call such a pair **csg-cmp-pair**. Here csg is the abbreviation of connected subgraph and cmp is the abbreviation of complement.

In the following, we are interested in

1. the number of connected, non-empty subsets
2. the number of csg-cmp-pairs

We denote the total number of csg-cmp-pairs including symmetric pairs by $\#\texttt{ccp}$. Ono and Lohman counted the number of csg-cmp-pairs by excluding symmetric pairs

For any correct dynamic programming algorithm $\#\texttt{ccp}$ provides a lower bound on the number of calls to `CreateJoinTree`

3 The New Algorithm DPccp

3.1 Problem Statement

If the search space is sparse, the DPsub algorithm considers many subproblems which are not connected and, therefore, are not relevant for the solution.

The main idea of DPccp is that it only considers pairs of connected subproblems.

Thus, our goal is to efficiently enumerate all csg-cmp-pairs (S_1, S_2). Requirements:

1. Enumerate every pair once and only once.
2. 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.
3. overhead for generating a single csg-cmp-pair must be constant or at most linear

```
DPccp
Input: a connected query graph with relations R =
{R_0, ..., R_{n-1}}
Output: an optimal bushy join tree
for all R_i ∈ R) {
    BestPlan({R_i}) = R_i;
}
for all csg-cmp-pairs (S_1, S_2), S = S_1 ∪ 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, ..., R_{n-1}});
```

Figure 3: Algorithm DPccp

3.2 Enumerating Connected Subsets

Let $G = (V, E)$ be an undirected graph. For a node $v \in V$ define the **neighborhood** $\mathcal{N}(v)$ of v as $\mathcal{N}(v) := \{v' \mid (v, v') \in E\}$. For a subset $S \subseteq V$ of V we define the **neighborhood** of S as $\mathcal{N}(S) := \bigcup_{v \in S} \mathcal{N}(v) \setminus S$. 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.

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:

1. Emit $\{v_i\}$ as a connected subset
2. Expand $\{v_i\}$ by calling a routine that extends a given connected set to bigger connected sets
3. let the routine be called with some connected set S. It then calculates the neighborhood $\mathcal{N}(S)$
4. For every non-empty subset $N \subseteq \mathcal{N}(S)$, it emits $S' = S \cup N$ as a further connected subset and resurcively calls itself with S'

The problem with this routine is that it produces duplicates.

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. 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 $\mathcal{B}_i = \{v_j \mid j \leq i\}$, 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 sub-
graph of G
for all i ∈ [n − 1, . . . , 0] descending {
    emit {v_i};
    EnumerateCsgRec(G, {v_i}, B_i);
}

EnumerateCsgRec(G, S, X)
N = N(S) \ X;
for all S' ⊆ N, S' ≠ ∅, enumerate subsets first {
    emit (S ∪ S');
}
for all S' ⊆ N, S' ≠ ∅, enumerate subsets first {
    EnumerateCsgRec(G, (S ∪ S'), (X ∪ N));
}
```

Figure 4: Algorithm `EnumerateCsg`

Lets's consider an example. Figure **??** contains a query graph. The calls to `EnumerateCsgRec` are contained in the table 3.7. 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 $\rightarrow$)

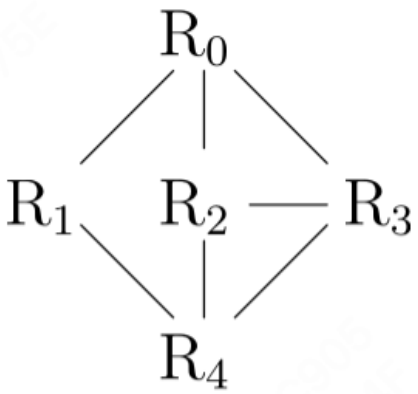

Figure 5: Sample graph to illustrate `EnumerateCsgRec`

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 6: Call sequence for Figure **??**

3.3 Enumerating Complements of Connected Subgraphs

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 `EnumerateCsgRec` with the correct parameters.

We need some definitions to state the actual algorithm., Let $S_1 \subseteq V$ be a non-empty subset of V. Then, we need to define $\min(S_1) := \min(\{i \mid v_i \in S_1\})$. This is used to extract the starting node from which S_1 was constructed. Let $W \subset V$ be a non-empty subset of V. Then we define $\mathcal{B}_i(W) := \{v_j \mid v_j \in W, j \leq i\}$.

`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, $\{v_i\}$, $X \cup N$);
}

Figure 7: Algorithm `EnumerateCmp`

Consider the graph **??**. Assume `EnumerateCmp` is called with $S_1 = \{R_1\}$. Then $X = \{R_0, R_1\}$. $N = \{R_0, R_4\} \setminus \{R_0, R_1\} = \{R_4\}$. Now we get a 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 $V' \subseteq V$, s.t. $G' = (V', E|_{V'})$ is a connected subgraph of G, where $E|_{V'} = \{(v, v') \in E \mid v, v' \in V'\}$. We denote the direct neighbors of a node v by

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

Indirect neighbors are collected into sets $\mathcal{N}_i(v)$:

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

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., $\exists v \in V : (v, v) \in E$. Furthermore, 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 $\left[\sum_{i=0}^{k-1} |\mathcal{N}_i(v_0)|, \sum_{i=0}^{k} |\mathcal{N}_i(v_0)|\right]$

3.4.2 Correctness of `EnumerateCsg`

Lemma 3.1. *Algorithm `EnumerateCsg` terminates if G is a finite graph*

Lemma 3.2. *Algorithm `EnumerateCsg` enumerates only connected components*

Proof. Induction on the recursion depth n.

$n = 0$: Singleton is a connected component.

Induction: `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. Any subset of N can be added to S to form a connected component □

Lemma 3.3. *Given a connected undirected graph $G = (V, E)$, a vertex $v \in V$, a natural number $n \geq 0$, and $V'_n = \bigcup_{i=0}^{n} \mathcal{N}_i(v)$. Then $(V'_n, E|_{V'_n})$ is a connected component.*

Lemma 3.4. *Given a connected, undirected graph $G = (V, E)$ and a vertex $v \in V$. Then $\exists n \geq 0$ s.t. $\forall_{0 \leq i \leq n} \mathcal{N}_i(v) \neq \emptyset$ and $\forall_{i > n} \mathcal{N}_i(v) = \emptyset$*

Lemma 3.5. *Given a connected, undirect graph $G = (V, E)$, $|V| > 1$ and a set of vertices $V' \subseteq V$ s.t. $(V', E|_{V'})$ is a connected component. Then $\exists v \in V'$ s.t. $(V' \setminus \{v\}, E|_{V' \setminus \{v\}})$ is a connected component.*

Proof. Let $G' = (V', E|_{V'})$ be a connected undirected graph and the base to compute $\mathcal{N}(v)$ and $\mathcal{N}_i(v)$. Choose arbitrary $v_0 \in V'$ and a natural number n s.t. $\mathcal{N}_n(v_0) \neq \emptyset \wedge \mathcal{N}_{n+1}(v_0) = \emptyset$. Note that $n > 0$ as $|V'| > 1$ and that $\bigcup_{0 \leq i \leq n} \mathcal{N}_i(v_0) = V'$. Now any $v \in \mathcal{N}_n(v_0)$ can be removed. □

Lemma 3.6. *When `EnumerateCsgRec` is called with additional vertices, it enumerates at least the same components as without the vertices. More formally:*

$$\{V \cup A \mid (V, E) \text{ enumerated by } \texttt{EnumerateCsgRec}(G, S, X)\} \subseteq$$
$$\{V \mid (V, E) \text{ enumerated by } \texttt{EnumerateCsgRec}(G, S \cup A, X)\}$$

Lemma 3.7. *Algorithm `EnumerateCsg` enumerates all connected components consisting of a single vertex*

Lemma 3.8. *Algorithm `EnumerateCsg` enumerates all connected components*

Proof. By contradiction. We assume that not all connected components are enumerated. Thus $\exists V' \subseteq V \wedge V \neq \emptyset$ s.t. $(V', E|_{V'})$ is a connected component and V' is not enumerated. If several such V' exists, we choose V' s.t. $|V'|$ is minimal. Then $|V'| > 1$ by Lemma 3.7 and we can delete a vertice v' from V' by Lemma 3.3 and get a new connected component which is enumerated.

Case 1: v' appeared in N during the enumeration of $V' \setminus \{v\}$, then V' would be enumerated by Lemma **??**

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., $L(v') < \min(\{L(v) \mid v \in V' \setminus \{v'\}\})$. Then `EnumerateCsg` will enumerate V' when selecting v' as the start vertex □

Lemma 3.9. *If V' and V'' are both enumerated and $\min(\{L(v) \mid v \in V'\}) = \min(\{L(v) \mid v \in V''\})$, V' and V'' are enumerated using the same start vertex*

Lemma 3.10. *Algorithm `EnumerateCsg` enumerates all connected components only once*

Proof. By contradiction. Choose $V' \subseteq V$ that is enumerated at least twice and is of minimal cardinality.

Case 1: $|V'| = 1$

Case 2: $|V'| > 1$. By Lemma 3.9, all enumerations of V' started with the same vertex and X.

A single invocation of `EnumerateCsgRec` (without the recursive call) does not produce duplicates. V' cannot be enumerated by two different calls to `EnumerateCsgRec` with the same parameters, as $|V'|$ is minimal. Thus there exists $S_1, S_2, X_1, X_2 \subseteq V$ s.t. $S_1 \neq S_2$, 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, S_2, X_2) enumerate V'. Hence

$$(V' \setminus S_1) \cap X_1 = \emptyset \wedge (V' \setminus S_2) \cap X_2 = \emptyset$$

As $S_1 \neq S_2$, there exists a invocation of `EnumerateCsgRec`, that recursively calls `EnumerateCsgRec` with S_1' and S_2', which finally lead to S_1 and S_2. Let Y be the corresponding exclusion filter in line 6. Then

$$\exists v \in (S_1' \cup S_2') : v \notin (S_1' \cap S_2') \wedge v \in Y$$
$$((v \in S_1 \wedge v \notin S_2) \vee (v \notin S_1 \wedge v \in S_2)) \wedge (v \in Y)$$
$$v \in V' \wedge v \notin V'$$

Essentially, $S_1 \neq S_2$ but $X_1 = X_2$. Now $v \in X_2 = X_1$, $v \notin S_2$ and therefore $v \notin V$. But $v \in S_1 \subseteq V'$. □

Lemma 3.11. *If $V' \subset V''$, $n = |V''| - |V'| - 1$ and both $(V', E|_{V'})$ and $(V'', E|_{V''})$ are connected components, then $\exists V_1 \dots V_n$ s.t. $V' \subset V_1$, $V_i \subset V_{i+1}$, $V_n \subset V''$ and $(V_i, E|_{V_i})$ is a connected component for all $1 \leq i \leq n$*

Lemma 3.12. *If $V' \subset V''$ and both $(V', E|_{V'})$ and $(V'', E|_{V''})$ are connected components, `EnumerateCsg` enumerates $(V', E|_{V'})$ before $(V'', E|_{V''})$*

Theorem 3.13. *Algorithm `EnumerateCsg` is correct*

Proof. 3.1, 3.2, **??**, 3.10, 3.12

- Terminate
- Only list connected components
- enumerates all connected components
- enumerates all connected components only once
- enumerates in a order that is suitable for dynamic programming

□

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 $(V' \subseteq V)$ s.t. $(V', E|_{V'})$ is a connected component, enumerate all $V'' \subseteq V \setminus V'$ s.t. $(V'', E|_{V''})$ and $(V' \cup V'', E|_{V' \cup V''})$ are connected components.

The algorithm 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 disjoin connected components that matches the enumeration order used in `EnumeratedCsg`:

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

Using this ordering, we only enumerate V'' for V' if $V' < V''$.

3.4.4 Proofs

Lemma 3.14. *Algorithm `EnumerateCmp` terminates if G is a finite graph*

Lemma 3.15. *Algorithm `EnumerateCmp` enumerates all connected components consisting of a single vertex*

Lemma 3.16. *Algorithm `EnumerateCmp` enumerates all adjacent connected components in the complement (that satisfy the ordering)*

Lemma 3.17. *Algorithm `EnumerateCmp` enumerates connected components only once*

Theorem 3.18. *Algorithm `EnumerateCmp` is correct*

4 Problems

5 References