DP-Counter Analytics

Guido Moerkotte

Technical Report TR-2006-002
University of Mannheim
February 9, 2006

Abstract

In the literature mainly two variants of dynamic programming for constructing join trees are described. We show analytically and experimentally that the runtime behaviors of those two variants differ vastly for different query graphs. The query graphs we consider are chain, cycle, star, and clique. More specifically, one of the variants is highly superior for chain and cycle queries whereas the other is highly superior for star and cliques queries. This motivates us to derive an optimal algorithm which is — apart from a small overhead — superior to all algorithms in all cases.

1 Introduction

The problem of generating optimal join trees has been extensively studied[1]. Since there are many different join ordering problems, let us state precisely which one we investigate here: We consider the generation of optimal bushy join trees not containing cross products. Note that this implies that the join graph is connected. The only approach we consider is dynamic programming.

Already starting with the seminal paper by Selinger et al. [7], there seems to be a habit to hide the actual code: only few papers present (pseudo-) code (e.g. [3]). This code reveals one variant of dynamic programming. It generates join trees by increasing size. We call this variant size-chained (`SzCh` or `DPsize` for short).

An alternative to dynamic programming is memoization which generates plans top-down instead of bottom-up. A very efficient approach has been proposed in the seminal paper written by Vance and Maier [11]. The core of their algorithm is a code snippet that allows for very efficient generation of subsets of a set represented as a bitvector. It is an easy exercise to derive a dynamic programming variant from their memoization algorithm. In fact, two variants with slight differences can be derived, which we denote by `Sub` and `SubAlt`. The latter is also called `DPsub`.

The first question we want to answer in this paper is which of the two variants (presented in detail in Section 2) is the better one. We could run

[1]For a complete coverage see manuscript by Moerkotte (http://pi3.informatik.uni-mannheim.de/moer/querycompiler.pdf)

experiments, of course, but we thought that analytical results would give us more insight. Scanning the literature revealed that only a single paper presents analytical results. It is the paper written by Ono and Lohman [5]. For three different kinds of query graphs (chains, stars, and cliques), they analyze the number of rule applications (or calls to cost functions). These are important results. However, these numbers are independent of the variant of dynamic programming used and give lower complexity bounds as pointed out by Vance [10]. As a consequence, the variants remain indistinguishable with this analysis. Hence, though fundamental, the results by Ono and Lohman do not help to answer our question.

The analytical approach we take in order to answer the question works as follows. We place a counter in the innermost loop of the variants and analytically derive the value of the counter after termination (see Section 3). That is, we count the number of executions of the innermost loop of the variants. This reveals the following:

1. In all cases but one, the value of the counter is much higher than the number of rule applications which is the lower bound for the inner counter.
2. For chain and cycle queries, the SzCh variant exhibits a vastly lower counter value than the Sub and SubAlt variants.
3. For star and clique queries, the Sub and SubAlt variants exhibit a vastly lower counter value than the SzCh variant.

These findings immediately spawn a second question: Is it possible to derive a dynamic programming variant whose inner counter is equal to the lower bound derived by Ono and Lohman? If so, this variant would beat the others. The positive answer to this question is presented in Section 4. Since the Sub and SubAlt alternatives use the very efficient subset generation code of Vance and Maier, the new variant exhibits a higher overhead which eludes a theoretical analysis. Therefore, and because we wanted to demonstrate the practical impact of choosing a certain variant, we experimentally evaluated the variants (Section 5).

2 The Existing Algorithms

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. The next subsections give implementation details. We start by discussing the common infrastructure used by all our algorithms (including our new one). Subsection 2.1 sketches the most prominent size-chained variant. Subsection 2.2 presents two variants based on fast subset generation [10, 11].

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

```
DPsize
Input:  a connected query graph with relations $R_0, \ldots, R_{n-1}$
Output:  an optimal bushy join tree without cross products
for (i = 0; i < n; ++i) {
  BestPlan(1 $<<$ $i$) = $R_i$;
}
for ($s = 2$; $s \leq n$; ++$s$) // size of plan to be constructed
for ($s_1 = 1$; $s_1 < s$; ++$s$) { // size of left subplan
  $s_2 = s - s_1$; // size of right subplan
  for (all plans $P_1$ containing $s_1$ relations)
  for (all plans $P_2$ containing $s_2$ relations) {
    ++InnerCounter;
    if ($\emptyset$ != $P_1 \cap P_2$) continue;
    if (not $P_1$ connected to $P_2$) continue;
    ++OnoLohmanCounter;
    CurrPlan = CreateJoinTree($P_1$, $P_2$);
    $S$ = relations contained in CurrPlan;
    if (BestPlan($S$) == NULL ||
        cost(BestPlan($S$)) $>$ cost(CurrPlan)) {
      BestPlan($S$) = CurrPlan;
    }
  }
}
return BestPlan($\{R_0, \ldots, R_{n-1}\}$);
```

Figure 1: Algorithm DPsize (SzCh)

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 ready 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 $s_1 = s_2$. In this case, the complexity can be decreased from $s_1 * s_2$ to $s_1 * s_2/2$ (see below for details). The following formulas are valid only for the variant of DPsize where this optimization has been incorporated.

2.2 Counter Driven Enumeration with Fast Subset Generation

Figure 2 contains the pseudocode for the two algorithms DPsub (Sub for short) and DPsubalt (SubAlt for short). Differing from DPsize, these two variants do not leave the implementor a choice of how to represent sets. Instead, bitvectors are used to encode sets. In a bitvector representing a set, the i-th bit is equal to '1' if R_i is contained in the set. As a consequence, the table `BestPlan` is now indexed by bitvectors.

Let us comment on the pseudocode given in Fig. 2. 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 subsets are successively represented by the bitvector S. It not only represents a subset of $\{R_0, \ldots, R_{n-1}\}$, but is also interpreted as a positive integer. 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

```
DPsub/DPsubalt
**Input:** a connected query graph with relations $R_0, \ldots, R_{n-1}$
**Output:** an optimal bushy join tree
for (i = 0; i < n; ++i) {
  BestPlan(1 $<<$ $i$) = $R_i$;
}
for (S = 1; S $<$ $2^n$-1; ++S) {
  if (not connected $S$) continue; // delete this line for DPsub
  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;
    CurrPlan = CreateJoinTree(BestPlan($S_1$), BestPlan($S_2$));
    if (BestPlan($S$) == NULL || cost(BestPlan($S$)) $>$ cost(CurrPlan)) {
      BestPlan($S$) = CurrPlan;
    }
  }
}
return BestPlan($2^n - 1$);
```

Figure 2: Algorithms DPsub (Sub) and DPsubalt (SubAlt)

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 [10, 11]. 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 must both 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`.

2.3 Implementation Details

2.3.1 Common Infrastructure

Figure 3 contains the most important parts of the common infrastructure. Plans are grouped into *plan classes*. Each plan class contains all plans with the same logical properties. The most interesting logical properties are the set of relations a plan joins (`_contained`) and its result cardinality (`_cardinality`). Other logical properties like result tuple width exist but have been excluded from the figure. Instead, we have introduced the set of relations to which the relations joined in a plan connect via join predicates (`_connectedTo`). We chose to represent all sets as bitvectors. Given two plan classes c_1 and c_2 whose plans join two disjoint sets of relations and whose corresponding sets of relations are connected by some join predicates, we can calculate the logical properties of the plan class resulting from joining plans in c_1 and c_2 as follows:

```
_contained   = c1._contained | c2._contained
_connectedTo = (c1._connectedTo | c2._connectedto) & ~_contained
```

The output cardinality can be calculated by iterating over the relations contained in

```
lLinkRelations   = c1._connectedTo & c2._contained
```

and considering for each relation the selectivity of the predicates connecting it to the relations in c_2.`_contained`.

The plans belonging to the same plan class are linked via the `_next` member of plan nodes. Further, each plan class contains a link to the first of its plans (`_plan`). Plan classes hold the logical properties of their plans. For the size-chained variant of dynamic programming, we additionally need a list of all plans of equal size (i.e. containing the same number of relations). For this purpose, each plan class contains a `_next` pointer. This pointer is not used by the other variants.

Plan nodes contain the physical properties of a plan. The most important physical property is the cost of a plan. Other physical properties like sortedness and groupedness of a plan's result are not shown. The member function `dominates` of physical properties is used to prune dominated plans.

Each plan node contains a back pointer to its class. Though this pointer can be avoided, it is very convenient. The members `_left` and `_right` are pointers to the arguments of a plan. If the algebraic operator is unary (e.g. a selection), the `_right` pointer is unused. The last member to be explained is `_rule`. It contains a pointer to the rule that constructed this plan. There are different rules to introduce different algebraic operators. This pointer allows to reconstruct the generated plan. The rule engine is similar to the one described in a paper by Lohman [4]. It is shared by all dynamic programming variants.

The last common infrastructure class is the memotable. Its main purpose is to provide a mapping from a set of relations to the plan class representing all plans which comprise exactly this set of relations. For simplicity, we implemented the memotable as a simple array with 2^n entries for a query involving n relations.

2.3.2 DP-Variant: Chaining by Size

For the rest of the paper we assume that n is the number of relations to be joined. The code of the first dynamic programming variant is shown in Figure 4. It generates plans with an increasing size by first joining two subplans each containing a single relation. Then, the subplan sizes are increased until all n relations have been joined. Therefore, the sizes of the left and the right argument of the next joins to be performed are increased in two nested loops. Given the size of the left and the right argument, a loop is performed over all plan classes representing plans with exactly these numbers of joins (`sizeLoop`). Then, a further loop iterates over all plans in these classes. This loop is implemented in a member function `planLoop` of the abstract base class of all dynamic programming variants.

Some optimization potential exists if the left and the right argument are of the same size. Then, instead of traversing the single list of plan classes in two nested loops both considering the list from its beginning to its end, we have to consider the whole list for the left argument and only the rest of the list (starting from the successor of left argument) for the right argument. Therefore, this case is distinguished in the main `optimize` procedure in Figure 4. The implementations of the overloaded member functions `sizeLoop` are shown in Figure 5.

Note that the work performed by iterating over the plans of two classes is the same for all dynamic programming variants. Moreover, for our experiments, we excluded all physical properties except for the costs. This has the nice effect that only one plan per plan class is retained. Differing from the analytical approach by Ono and Lohman [5], our rules do not consider commutativity. Hence, our numbers will be twice as large as theirs for counting the number of `planLoop` calls.

Within the member function `sizeLoop` the following situations can occur:

1. the sets of relations of the two argument classes are not connected by a join predicate, and/or
2. the sets of relations of the two argument classes are not disjoint.

Since we do not consider cross products and do not want to join a relation

```
class PlanPropertyLog {
    Bitvector _contained;
    Bitvector _connectedTo;
    card_t    _cardinality;
    ...
};

class PlanPropertyPhys {
    int dominates(PlanPropertyPhys& x); // best: -1 this, 0 uncomparable, 1 x
    cost_t _cost;
    ...
};

class PlanClass {
    PlanPropertyLog& prop() { return _prop; }
    PlanNode*        plan() { return _plan; }
    PlanClass*       next() { return _next; }
  public:
    PlanPropertyLog  _prop;
    PlanNode*        _plan;
    PlanClass*       _next; // for size chain
};

class PlanNode {
    PlanClass*  _class;
    RuleBase*   _rule;
    PlanNode*   _left;
    PlanNode*   _right;
    PlanNode*   _next;
    PlanPropertyPhys _prop;
};

class MemoTable {
    PlanClass&  find(Bitvector& aContained);
};
```

Figure 3: Common infrastructure

```
PlanGeneratorSizeChained::optimize() {
  for(int lSizeLeft = 1; lSizeLeft < n; ++lSizeLeft) {
    int lSizeRightLimit = 0;
    lSizeRightLimit = (lSizeLeft <= n - lSizeLeft) ? lSizeLeft : n - lSizeLeft;
    for(int lSizeRight = 1; lSizeRight <= lSizeRightLimit; ++lSizeRight) {
      if(lSizeLeft != lSizeRight) {
         sizeLoop(lSizeLeft, lSizeRight);
      } else {
         sizeLoop(lSizeLeft); // special treatment
      }
    }
  }
}
```

Figure 4: Dynamic programming variant DPcsg

more than once, two tests have been introduced. Just before the two tests, we introduce the inner counter.

2.3.3 DP-Variant: Subset

Figure 6 shows the code of the dynamic programming variants `Sub` and `SubAlt`. The core idea of the algorithm is the following. For every subset S of the relations to be joined, it generates all subsets S_1 and S_2 of S. The foundation for this is the fast subset enumeration algorithm of Vance and Maier [10, 11] to enumerate the subsets S_1 of S. Then, S_2 is easily determined by $S_2 = S \setminus S_1$. However, one of the following situations may arise:

1. The subset S_1 does not induce a connected subgraph of the query graph,
2. the complement S_2 of the subset S_1 does not induce a connected subgraph of the query graph, and/or
3. the subset S_1 and its complement S_2 are not connected by join predicates.

In any of these cases, the pair (S_1, S_2) must be condemned. Hence, tests have to be introduced before passing the plan classes to `loopPlan`. Again, the inner counter is placed before the necessary tests.

If S is not connected, at least one of the above situations arises. Thus, it does make a lot of sense to test S for connectedness before deriving S_1 and S_2. Hence, we considered a variant (called `SubAlt`) where this is checked and a variant (called `Sub`) that does not contain this test. It should be obvious that including this test leads to a better performance as long as the query graph is not (close to) a clique.

```
PlanGeneratorSizeChained::sizeLoop(int aSizeLeft, int aSizeRight) {
  PlanClass* lClassLeftStart  = _sizeArray[aSizeLeft];
  PlanClass* lClassRightStart = _sizeArray[aSizeRight];
  PlanClass* lClassLeft = 0;
  PlanClass* lClassRight = 0;
  for(lClassLeft = lClassLeftStart; lClassLeft != 0; lClassLeft = lClassLeft->next()) {
    for(lClassRight = lClassRightStart; lClassRight != 0;
        lClassRight = lClassRight->next()) {
      Bitvector lContained = lClassLeft->prop().contained();
      lContained |= lClassRight->prop().contained();
      ++(_stat._countInner);  // counter for innermost loop
      // problems: 1. left/right may overlap, 2. left/right may not be connected
      if((lClassLeft->prop().contained().overlap(lClassRight->prop().contained())) ||
         (lClassLeft->prop().contained().disjoint(lClassRight->prop().connected()))) {
        continue;
      }
      PlanClass& lClassResult = getPlanClass(lContained);
      loopPlan(*lClassLeft, *lClassRight, lClassResult);
      loopPlan(*lClassRight, *lClassLeft, lClassResult);
    }
  }
}

PlanGeneratorSizeChained::sizeLoop(int aSize) {
  PlanClass* lClassLeftStart  = _sizeArray[aSize];
  PlanClass* lClassLeft = 0;
  PlanClass* lClassRight = 0;
  for(lClassLeft = lClassLeftStart; lClassLeft != 0; lClassLeft = lClassLeft->next()) {
    for(lClassRight = lClassLeft->next(); lClassRight != 0;
        lClassRight = lClassRight->next()) {
      Bitvector lContained = lClassLeft->prop().contained();
      lContained |= lClassRight->prop().contained();
      ++(_stat._countInner);  // counter for innermost loop
      // problems: 1. left/right may overlap, 2. left/right maybe not connected
      if((lClassLeft->prop().contained().overlap(lClassRight->prop().contained())) ||
         (lClassLeft->prop().contained().disjoint(lClassRight->prop().connected()))) {
        continue;
      }
      PlanClass& lClassResult = getPlanClass(lContained);
      loopPlan(*lClassLeft, *lClassRight, lClassResult);
      loopPlan(*lClassRight, *lClassLeft, lClassResult);
    }
  }
}
```

Figure 5: Dynamic programming variant DPsize `sizeLoop`

```
PlanGeneratorSubset::optimize() {
  for(Bitvector s = 1; s <= ((1 << n) - 1); ++s) {
    if(!isConnectedSubgraph(s)) { continue; } // only for SubAlt
    const Bitvector lBvCurrent(s);
    PlanClass&      lClassCurrent = memo().find(s);
    bool first = true;
    for(Bitvector::IteratorSubset iter = lBvCurrent.beginSub();
                                  iter != lBvCurrent.endSub(); ++iter) {
      Bitvector lBvLeft  = (*iter);
      Bitvector lBvRight(lBvCurrent ^ lBvLeft);
      ++(_stat._countInner);  // counter for innermost loop
      PlanClass& lClassLeft  = memo().find(lBvLeft);
      PlanClass& lClassRight = memo().find(lBvRight);
      // check whether left and right induce connected subgraph
      if((0 == lClassLeft.plan()) ||
         (0 == lClassRight.plan())) {
        continue;
      }
      // check for predicates connecting left and right
      if(lClassLeft.prop().contained().disjoint(lClassRight.prop().connected())) {
        continue;
      }
      loopPlan(lClassLeft, lClassRight, lClassCurrent);
    }
  }
}
```

Figure 6: Dynamic programming variant DPsub

3 Analysis

3.1 Algorithm-Independent Results

Every subset of nodes (relations) of the join graph induces a subgraph. Since we do not consider cross products, we are interested in connected subgraphs. For a given graph G with n nodes, we denote by $\#\texttt{csg}(n)$ the number of non-empty connected subgraphs and by $\#\texttt{csg}(n, k)$ the number of non-empty connected subgraphs with k ($k > 0$) nodes. Sometimes, for small n up to 2 or 3 some of the following formulas do not hold. Hence, we assume $n > 3$. The concrete values for these n and up to 20 are contained in the next subsection. We do not give any proofs in this section. All proofs are contained in the appendices.

For chains, cycles, stars, and cliques with n nodes we have

$$\#\texttt{csg}^{\text{chain}}(n) = \frac{n(n+1)}{2} \quad (1)$$

$$\#\texttt{csg}^{\text{cycle}}(n) = n^2 - n + 1 \quad (2)$$

$$\#\texttt{csg}^{\text{star}}(n) = 2^{n-1} + n - 1 \quad (3)$$

$$\#\texttt{csg}^{\text{clique}}(n) = 2^n - 1 \quad (4)$$

It would be very nice to have an algorithm which takes a query graph and returns the number of node-induced connected subgraphs. This algorithm could then be used to estimate the runtime of the plan generator and to decide which data structure to use for the memotable. Further, we could deduce from this information the expected memory consumption for the memotable. Unfortunately, no efficient such algorithm exist. Moreover, we cannot expect to find one, since the problem of calculating the number of connected subgraphs of a given graph is $\#P$-hard [9].

The above equations can be derived from the following by summing over k:

$$\#\texttt{csg}^{\text{chain}}(n, k) = (n - k + 1) \quad (5)$$

$$\#\texttt{csg}^{\text{cycle}}(n, k) = \begin{cases} 1 & n = k \\ n & \text{else} \end{cases} \quad (6)$$

$$\#\texttt{csg}^{\text{star}}(n, k) = \begin{cases} n & k = 1 \\ \binom{n-1}{k-1} & k > 1 \end{cases} \quad (7)$$

$$\#\texttt{csg}^{\text{clique}}(n, k) = \binom{n}{k} \quad (8)$$

$$(9)$$

These results are also used to derive the values for the inner counter for the size-chained variant.

For the number of rule applications we have

$$\#\texttt{rap}^{\text{chain}}(n) = \frac{1}{3}((n+1)^3 - (n+1)^2 + 2 * (n+1)) \quad (10)$$

$$\#\texttt{rap}^{\text{cycle}}(n) = n^3 - 2n^2 + n \quad (11)$$

$$\#\texttt{rap}^{\text{star}}(n) = (n-1)2^{n-2} \quad (12)$$

$$\#\texttt{rap}^{\text{clique}}(n) = 3^n - 2^{n+1} + 1 \quad (13)$$

The results for chain, star, and clique are due to Ono and Lohman [5]. However, there is a minor difference: we count calls to `CreateJoinTree(`T_1`, `T_2`)` and its counterpart `CreateJoinTree(`T_2`, `T_1`)` as two calls, whereas Ono and Lohman count these commutative calls only once. Thus, the above formulas have to be divided by two in order to give Ono and Lohman's formulas. Further note that #`rap` is a lower bound on the work to be performed by any dynamic programming variant. It corresponds to the number of calls to `loopPlan` in all our dynamic programming variants.

3.2 Algorithm-Dependent Results

The value of the inner counter of the DP-Variant `SzCh` can be calculated as follows. For chains, we have:

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

For cycles, we have:

$$I^{\text{cycle}}_{\text{SzCh}}(n) = \begin{cases} \frac{1}{4}(n^4 - n^3 - n^2) & n \text{ even} \\ \frac{1}{4}(n^4 - n^3 - n^2 + n) & n \text{ odd} \end{cases} \tag{15}$$

For stars, we have:

$$I^{\text{star}}_{\text{SzCh}}(n) = \begin{cases} 2^{2n-4} - 1/4\binom{2(n-1)}{n-1}) + q(n) & n \text{ even} \\ 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + 1/4\binom{n-1}{(n-1)/2} + q(n) & n \text{ odd} \end{cases} \tag{16}$$

with $q(n) = n2^{n-1} - 5 * 2^{n-3} + 1/2(n^2 - 5n + 4)$.

For cliques, we have:

$$I^{\text{clique}}_{\text{SzCh}}(n) = \begin{cases} 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} - 1/4\binom{n}{n/2} + 1 & n \text{ even} \\ 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} + 1 & n \text{ odd} \end{cases} \tag{17}$$

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

The *Inner Counter* of the DP-Variant `Sub` is independent of the query graph and can be calculated as follows:

$$I_{\text{Sub}} = 3^n - 2^{n+1} + 1 \tag{18}$$

The value of the inner counter for the DP-Variant `SubAlt` can be calculated as follows:

$$\begin{aligned} I^{\text{chain}}_{\text{SubAlt}}(n) &= 2^{n+2} - n^n - 3n - 4 & (19) \\ I^{\text{cycle}}_{\text{SubAlt}}(n) &= n2^n + 2^n - 2n^2 - 2 & (20) \\ I^{\text{star}}_{\text{SubAlt}}(n) &= 23^{n-1} - 2^n & (21) \\ I^{\text{clique}}_{\text{SubAlt}}(n) &= 3^n - 2^{n+1} + 1 & (22) \end{aligned}$$

	Chain		Cycle		Star		Clique	
n	Csg	Rap	Csg	Rap	Csg	Rap	Csg	Rap
2	3	2	3	2	3	2	3	2
3	6	8	7	12	6	8	7	12
4	10	20	13	36	11	24	15	50
5	15	40	21	80	20	64	31	180
6	21	70	31	150	37	160	63	602
7	28	112	43	252	70	384	127	1932
8	36	168	57	392	135	896	255	6050
9	45	240	73	576	264	2048	511	18660
10	55	330	91	810	521	4608	1023	57002
11	66	440	111	1100	1034	10240	2047	173052
12	78	572	133	1452	2059	22528	4095	523250
13	91	728	157	1872	4108	49152	8191	1577940
14	105	910	183	2366	8205	106496	16383	4750202
15	120	1120	211	2940	16398	229376	32767	14283372
16	136	1360	241	3600	32783	491520	65535	42915650
17	153	1632	273	4352	65552	1048576	131071	128878020
18	171	1938	307	5202	131089	2228224	262143	386896202
19	190	2280	343	6156	262162	4718592	524287	1161212892
20	210	2660	381	7220	524307	9961472	1048575	3484687250

Table 1: Sample values for #`csg` and #`rap`

3.3 Sample Numbers

Tables 1 and 3.3 provide concrete values for join graph sizes between 2 and 20. For different query graphs, Table 1 gives the number of connected subgraphs and the number of rule applications. Table 3.3 shows the values for the inner counter for different variants of dynamic programming. We already list the inner counter values for the new variant `DPcsg`, which will be presented in the next section. Since for the DP-variant `Sub` the value of its inner counter is independent of the query graph, there is only one column for it.

We observe the following:

- As expected, `SubAlt` is clearly superior to `Sub`.
- For chain and cycle queries, the `SizeChained` variant behaves much better than the `Sub` and `SubAlt` variant,
- For star and clique queries, the `Sub` and `SubAlt` variants behaves much better than the *SizeChained* variant.
- By comparison with the numbers given in Table 1 it becomes clear that except for cliques, the number of rule applications is far less than the value of *InnerCounter* for all DP-variants.

These observations motivate us to derive a new algorithm whose *InnerCounter* value is equal to the number of rule applications.

	Chain			Cycle			Star			Clique			All Graphs
n	SubAlt	SzCh	CSG	SubAlt	SzCh	CSG	SubAlt	SzCh	CSG	SubAlt	SzCh	CSG	Sub
2	2	1	1	2	1	1	2	1	1	2	1	1	2
3	10	9	4	12	12	6	10	9	4	12	12	6	12
4	32	29	10	46	44	18	38	33	12	50	61	25	50
5	84	73	20	140	120	40	130	110	32	180	280	90	180
6	198	150	35	374	261	75	422	350	80	602	1171	301	602
7	438	278	56	924	504	126	1330	1175	192	1932	4795	966	1932
8	932	470	84	2174	880	196	4118	4116	448	6050	19265	3025	6050
9	1936	750	120	4956	1440	288	12610	15188	1024	18660	77052	9330	18660
10	3962	1135	165	11062	2225	405	38342	57888	2304	57002	306991	28501	57002
11	8034	1655	220	24332	3300	550	116050	226037	5120	173052	1222375	86526	173052
12	16200	2331	286	52958	4716	726	350198	894278	11264	523250	4864993	261625	523250
13	32556	3199	364	114348	6552	936	1054690	3566678	24576	1577940	19367127	788970	1577940
14	65294	4284	455	245366	8869	1183	3172262	14281579	53248	4750202	77116677	2375101	4750202
15	130798	5628	560	523836	11760	1470	9533170	57305929	114688	14283372	307173877	7141686	14283372
16	261836	7260	680	1113598	15296	1800	28632278	230139494	245760	42915650	1223926785	21457825	42915650
17	523944	9228	816	2358716	19584	2176	85962370	924507240	524288	128878020	4878205012	64439010	128878020
18	1048194	11565	969	4980086	24705	2601	258018182	3713761316	1114112	386896202	19448313175	193448101	386896202
19	2096730	14325	1140	10485036	30780	3078	774316690	14915750705	2359296	1161212892	77555137327	580606446	1161212892
20	4193840	17545	1330	22019294	37900	3610	2323474358	59892991338	4980736	3484687250	309338182241	1742343625	3484687250

Table 3.3: Sample Values for Inner Count

4 The New Algorithm DPcsg

4.1 Problem Statement

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 connected.

The goal of this new algorithm is to enumerate

- non-empty subsets S_1 of $\{R_0, \ldots, R_{n-1}\}$, and
- non-empty subsets S_2 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.

Let us call a pair fulfilling all these conditions a csg-cmp-pair.

We want to enumerate only 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 a way 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 pair of that kind must be constant or at most linear. This condition is necessary in order to beat DPsize and DPsub.

If we meet all these requirements, the algorithm `DPcsg` is easily specified: iterate over all csg-cmp-pairs (S_1, S_2) and consider joining the best plans associated with them. Figure 7 shows the pseudocode. 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.

The rest of this section is organized as follows. The next subsection discusses an algorithm enumerating non-empty connected subsets S_1 of $\{R_0, \ldots, R_{n-1}\}$. Subsection 4.3 then shows how to enumerate the complements S_2 such that (S_1, S_2) is a csg-cmp-pair.

4.2 Enumerating Connected Subsets

Scanning the literature for algorithms enumerating 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]. We could not use the whole algorithm directly for a number of reasons: the algorithm was flawed; it maintained a

```
DPcsg
Input:  a connected query graph with relations $R_0, \ldots, R_{n-1}$
Output:  an optimal bushy join tree
for (i = 0; i < n; ++i) {
  BestPlan({$R_i$}) = $R_i$;
}
forall csg-cmp-pairs ($S_1, S_2$) {
  ++InnerCounter;
  CurrPlan = CreateJoinTree(BestPlan($S_1$), BestPlan($S_2$));
  if (BestPlan($S$) == NULL || cost(BestPlan($S$)) > cost(CurrPlan)) {
    BestPlan($S$) = CurrPlan;
  }
  CurrPlan = CreateJoinTree(BestPlan($S_2$), BestPlan($S_1$));
  if (cost(BestPlan($S$)) > cost(CurrPlan)) {
    BestPlan($S$) = CurrPlan;
  }
}
return BestPlan({$R_0, \ldots, R_{n-1}$});
```

Figure 7: Algorithm DPcsg

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.

We could generate all connected subsets as follows. For every node $v_i \in V$ we proceed as follows. First, we emit $\{v_i\}$ as a connected subset. Then, we call a routine that extends a given connected set to bigger connected sets. We call this routine for every set $\{v_i\}$. 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 only 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 [1]. The idea of the following algorithm is as follows. In order to avoid duplicates, it 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 a breadth-first search
**Output:** emits all subsets of $V$ inducing a connected sugraph of $G$
**for** ($i = n - 1$; $i \geq 0$; --$i$) {
  **emit** $\{v_i\}$;
  EnumerateCsgRec($G$, $\{v_i\}$, $\mathcal{B}_i$);
}

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

4.3 Enumerating Complements of Connected Subgraphs

In order to generate all csg-cmp-pairs, we procede as follows. We first generate all connected subsets S_1 in an outer loop using the procedure described in the previous subsection. Then, for each such S_1, we generate all its complements S_2. Again, the challenge is to generate each csg-cmp-pair only once.

To achieve this, we have to restrict the subsets generated for a given connected subset S. We only consider complements S_2 of S_1 (with (S_1, S_2) being a csg-cmp-pair) such that S_2 only contains v_j with j larger than any i for which $v_i \in S$. This avoids the generation of duplicates.

We need some definitions to state the actual algorithm. Let $S \subseteq V$ be a non-empty subset of V. Then, we define $\min(S) := \min(\{i | v_i \in S\})$. This is used to extract the starting node from which S was constructed. 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 looks as follows:

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

```
for all ($v_i \in N$ by descending $i$) {
    emit $\{v_i\}$;
    EnumerateCsgRec($G$, $\{v_i\}$, $X \cup \mathcal{B}_i(N)$);
}
```

4.4 Implementation Details

It is straight-forward to implementat the pseudocode given in Figure 7. Figure 8 gives the resulting code for the dynamic programming variant using an iterator to enumerate connected components and their complements.

The procedure `EnumerateCsg` could also easily be implemented. However, for our dynamic programming routine we need an iterator based implementationt of it. First note that `EnumerateCsg` calls a recursive subroutine. Evaluating this requires a stack. Hence, the class representing the iterator for connected subgraphs has a stack as its member. On this stack, we keep the parameters of `EnumerateCsgRec`. Note that `EnumerateCsgRec` has two loops: one for generating subsets of a neighborhood and one for calling itself recursively on these subsets. Generating subsets can be done highly efficiently by a code fragment introduced by Vance and Maier [10, 11]: This code fragment generates all subsets of a given set S including the empty set and S itself. The enumeration order suits our purpose except that the before-last element generated is the empty set, which we do not need to consider, and the last element is the full set S, which we do have to consider. Thus, we stop enumeration as soon as we see the empty set and consider the full set S separately. These considerations lead to the following states (steps) for our iterator implementing `EnumerateCsg` and `EnumerateCsgRec`.

1. iterator not yet initialized
2. iterator initialized, emit results corresponding to subsets S' with $\emptyset \subset S' \subset S$
3. emit result corresponding to S
4. subset iterator needs to be reinitialized
5. consider subsets S' with $\emptyset \subset S' \subset S$
6. consider S

After the last step, we have to emulate the return from `EnumerateCsgRec` and hence pop the stack. These different activities necessary in different states can be implemented using an array of function pointers or using a `switch` statement. Figure 9 shows the implementation using the latter. The procedures implementing the stack operations are given in Figure 10.

The iterator `IteratorCsgAndComplement`, which steps through csg-cmp-pairs consists of two different csg-iterators. The first one generates the first component of a csg-cmp-pair, while the second steps through the second components. Its initialization looks as follows:

```
void
PlanGeneratorCsg::optimize() {
  IteratorCsgAndComplement lIterCsgCmp(qg(), memo());
  Bitvector lBvLeft;
  Bitvector lBvRight;
  Bitvector lBvCurrent;

  for(lIterCsgCmp.init(); lIterCsgCmp.isValid(); ++lIterCsgCmp) {
    lIterCsgCmp.getCombination(lBvLeft, lBvRight);
    lBvCurrent = unionOf(lBvLeft, lBvRight);
    PlanClass& lClassLeft = memo().find(lBvLeft);
    PlanClass& lClassRight = memo().find(lBvRight);
    PlanClass& lClassCurrent = memo().find(lBvCurrent);
    ++(_stat._countInner);
    // combine
    //
    loopPlan(lClassLeft, lClassRight, lClassCurrent);
    loopPlan(lClassRight, lClassLeft, lClassCurrent);
  }
}
```

Figure 8: Dynamic programming variant DPcsg

```
void
PlanGeneratorCsg::IteratorCsgAndComplement::init() {
  for(_bfsCsgCounter = (qgraph().noNodes() - 1);
      0 <= _bfsCsgCounter;
      --_bfsCsgCounter) {
    _bfsXclCsg.set_first_n(_bfsCsgCounter);
    for(_iterCsg.init( (1 << _bfsCsgCounter), _bfsXclCsg.bitvector());
        _iterCsg.isValid(); ++_iterCsg) {
      _iterCsg.collectBfsNeibor((*_iterCsg), _csgConnection);
      _csgConnection.set_difference((*_iterCsg).bitvector() | _bfsXclCsg.bitvector());
      while(_csgConnection.is_not_empty()) {
        const Bitvector lCsgConnectionLowestBit = _csgConnection.lowest_bit();
        _csgConnection.set_difference(lCsgConnectionLowestBit);
        Bitvector lComplementExclude = ( (*_iterCsg).bitvector()
                                       | _bfsXclCsg.bitvector()
                                       | _csgConnection.bitvector() );
        _iterCmp.init(lCsgConnectionLowestBit.bitvector(),
                      lComplementExclude.bitvector());
        if(_iterCmp.isValid()) {
          _valid = true;
          return;
        }
      }
    }
  }
  _valid = false;
```

```
bool PlanGeneratorCsg::IteratorCsg::nextSwitch() {
  StackEntry& lX = *(top());
  switch(lX._round) {
    case 1: lX._neiborSubsetIter = lX._bfsNeibor.beginSub();
            lX._round = 2;
            if(lX._neiborSubsetIter.isValid()) {
              _current = ( (*(lX._neiborSubsetIter)) | lX._visited );
              break;
            }
            goto L017;
    case 2: ++(lX._neiborSubsetIter);
            if(lX._neiborSubsetIter.isValid()) {
              _current = ( (*(lX._neiborSubsetIter)) | lX._visited );
              break;
            }
    L017:   _current = (lX._bfsNeibor | lX._visited);
            lX._round = 4;
            break;
    case 4: bool lFound = false;
            if(_allNodes == (lX.exclude() | lX._bfsNeibor)) {
              nextUpSwitch(lX);
              break;
            }
            lX._neiborSubsetIter = lX._bfsNeibor.beginSub();
            lX._round = 5;
            while((lX._neiborSubsetIter.isValid())
                  && (!(lFound = nextDownSwitch(lX, *(lX._neiborSubsetIter))))) {
                ++(lX._neiborSubsetIter);
            }
            if(lFound) { break; }
    case 5: bool lFound = false;
            ++(lX._neiborSubsetIter);
            while((lX._neiborSubsetIter.isValid())
                  && (!(lFound = nextDownSwitch(lX, *(lX._neiborSubsetIter))))) {
                ++(lX._neiborSubsetIter);
            }
            if(lFound) { break; }
    case 6: lX._round = 7;
            if(nextDownSwitch(lX, lX._bfsNeibor)) {
              break;
            }
    case 7: nextUpSwitch(lX);
            break;
  }
  return _valid;
}
```

Figure 9: Iterator for CSGs

```
bool
PlanGeneratorCsg::IteratorCsg::nextDownSwitch(StackEntry& aFormer,
                                              Bitvector& aNeiborSet) {
  StackEntry& lX = *(++_stackTop);

  lX._visited  = (aFormer._visited | aNeiborSet);

  collectBfsNeibor(aNeiborSet, lX._bfsNeibor);
  lX._bfsNeibor.set_difference(aFormer._toExclude);

  lX._toExclude = (aFormer._toExclude | lX._bfsNeibor);

  if(lX._bfsNeibor.is_empty()) {
    --_stackTop;
    return false;
  }

  lX._round = 1;
  _valid = true;
  return nextSwitch();
}

bool
PlanGeneratorCsg::IteratorCsg::nextUpSwitch(StackEntry& aStackEntry) {
  if(bottom() == top()) { return (_valid = false); }
  --_stackTop;
  return nextSwitch();
}
```

Figure 10: Subroutines for iterator for CSGs

```
}
```

In order to implement the `next` routine on this iterator, the `init` code must be unwinded to provide all necessary entry points. Since this is an easy exercise and the result is quite lengthy, we restrain from giving the code here. It can be obtained from the author.

5 Evaluation

The next four tables contain the cpu times for chain, cycle, star, and clique graphs. We first observe that except for cliques, `SubAlt` is always superior to `Sub`. Further, even for cliques `SubAlt` exhibits only a negligible overhead over `Sub`. This essentially means that `Sub` is no candidate for an efficient implementation of a join ordering algorithm.

As indicated by the theoretical investigations of Section 3, `SzCh` and `DPcsg` are superior to `Sub` and `SubAlt` for chain and cycle queries. For star and clique queries, `Sub`, `SubAlt`, and `DPcsg` are superior to `SzCh`.

Also, `DPcsg` is highly superior to `SubAlg` for star queries. Only for clique queries the overhead imposed by the iterator for csg-cmp-pairs shows. But it is always below 30%. Furthermore, since star queries are of high practical importance in data warehouses and clique queries do not have any practical value, `DPcsg` is the algorithm of choice.

Chain

n	Sub	SubAlt	SzCh	Csg
2	1.47977e-06	1.55976e-06	1.54977e-06	1.78973e-06
3	3.04953e-06	3.41948e-06	2.56961e-06	3.39948e-06
4	5.24921e-06	5.8891e-06	4.96925e-06	6.06907e-06
5	8.75866e-06	9.72852e-06	7.76882e-06	9.2586e-06
6	1.69074e-05	1.71074e-05	1.17182e-05	1.39179e-05
7	3.56546e-05	2.98255e-05	1.73974e-05	2.04169e-05
8	8.22775e-05	5.2802e-05	2.53262e-05	2.97355e-05
9	0.000205189	9.62854e-05	3.88441e-05	4.41733e-05
10	0.000545147	0.000178973	5.86511e-05	6.44802e-05
11	0.00151177	0.000335949	9.02463e-05	9.72752e-05
12	0.00442833	0.000644902	0.000144318	0.000151837
13	0.012878	0.00138679	0.000371944	0.000379443
14	0.0382092	0.00270759	0.000666899	0.000675397
15	0.115032	0.00561715	0.0013093	0.0013173
16	0.363045	0.0124281	0.00278708	0.00279858
17	1.24114	0.026146	0.00604208	0.00601359
18	4.6063	0.0535769	0.0121392	0.0119812
19	17.0454	0.107784	0.0240364	0.0240313
20	61.9326	0.218167	0.0482477	0.0479477

Cycle

n	Sub	SubAlt	SzCh	Csg
2	1.47978e-06	1.55976e-06	1.54976e-06	1.78973e-06
3	3.50947e-06	3.87941e-06	2.95955e-06	3.83942e-06
4	6.73897e-06	7.58885e-06	6.63899e-06	7.91879e-06
5	1.27881e-05	1.45578e-05	1.11683e-05	1.37979e-05
6	2.44463e-05	2.73058e-05	1.86272e-05	2.33065e-05
7	5.01724e-05	5.06523e-05	2.95355e-05	3.68544e-05
8	0.000107983	9.2386e-05	4.66129e-05	5.59615e-05
9	0.000250762	0.000168114	6.97594e-05	8.30774e-05
10	0.000629404	0.000313972	0.000104074	0.000118402
11	0.00166225	0.00059291	0.000152297	0.000172644
12	0.0045933	0.00113383	0.000229605	0.000251152
13	0.0135329	0.00237764	0.000484516	0.000508422
14	0.0390991	0.00470878	0.000824575	0.000845371
15	0.116532	0.0102414	0.00153227	0.00154027
16	0.366194	0.0246912	0.00308203	0.00308203
17	1.26314	0.0557415	0.00627705	0.00626105
18	4.6093	0.114633	0.0125911	0.0125991
19	17.0384	0.235614	0.0248662	0.0247012
20	61.7496	0.478277	0.0486126	0.0484176

Star

n	Sub	SubAlt	SzCh	Csg
2	1.46978e-06	1.53976e-06	1.52977e-06	1.76973e-06
3	3.14952e-06	3.43948e-06	2.6196e-06	3.24951e-06
4	5.71913e-06	6.20906e-06	5.27919e-06	6.04908e-06
5	1.08684e-05	1.19282e-05	9.80851e-06	1.04984e-05
6	2.47662e-05	2.6336e-05	2.03769e-05	2.05869e-05
7	5.62614e-05	5.8971e-05	4.43333e-05	4.24736e-05
8	0.000138979	0.000139519	0.000103494	9.51955e-05
9	0.000345447	0.00032909	0.000257311	0.000208969
10	0.000871927	0.000797379	0.000689396	0.000447512
11	0.00227665	0.0019847	0.00214817	0.000999848
12	0.00614607	0.00503024	0.00876367	0.00218617
13	0.0168974	0.013378	0.0369244	0.00481527
14	0.0476228	0.0360745	0.157026	0.0103774
15	0.137129	0.100385	0.706993	0.0223616
16	0.417587	0.298955	3.54679	0.0495175
17	1.37879	0.931158	34.8567	0.102884
18	4.89892	3.21418	233.397	0.219917
19	18.1885	11.526	1133.95	0.468629
20	67.9017	42.6665	4791.74	0.998448

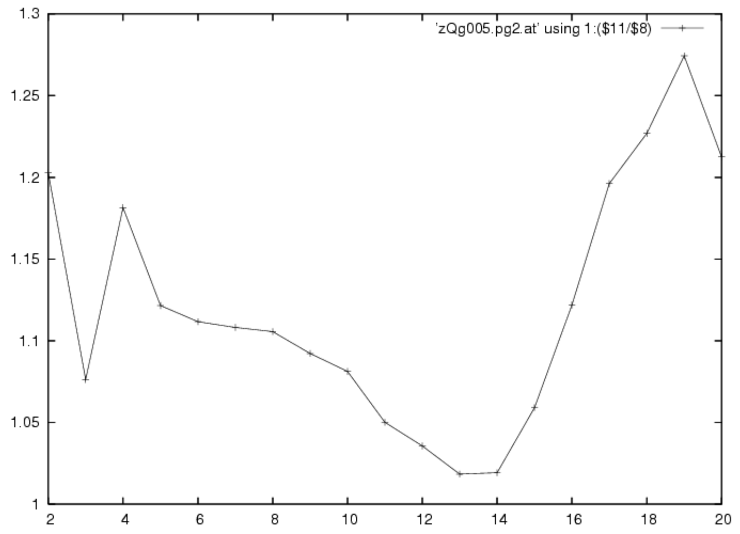

Figure 11: Overhead of CSG compared to SubAlt in case of Cliques

Clique

n	Sub	SubAlt	SzCh	Csg
2	1.47978e-06	1.55976e-06	1.52977e-06	1.77973e-06
3	3.53946e-06	3.9394e-06	2.90956e-06	3.80942e-06
4	8.09877e-06	9.10861e-06	7.80882e-06	9.56854e-06
5	2.14767e-05	2.38664e-05	2.09368e-05	2.40863e-05
6	6.06208e-05	6.67899e-05	6.08008e-05	6.73897e-05
7	0.000176473	0.000191041	0.000186802	0.00019555
8	0.000516851	0.000550417	0.000570683	0.000571413
9	0.00155076	0.00162725	0.00179373	0.00169374
10	0.0046063	0.00482727	0.00578412	0.00498024
11	0.0139779	0.0144028	0.0206769	0.0146778
12	0.0421186	0.0431984	0.0768883	0.0436184
13	0.127881	0.12933	0.288706	0.13023
14	0.387991	0.388841	1.11483	0.39544
15	1.17982	1.18649	4.61097	1.24948
16	3.69777	3.70244	30.5883	4.14837
17	11.829	11.8877	214.408	14.1496
18	39.746	40.0739	1059.55	48.7646
19	129.182	135.581	4706.31	164.616
20	436.875	439.507	21294.2	529.699

Figure 11 analyses the overhead more carefully. The overhead clearly differs with n. It decreases until $n = 14$. This is due to the case that the initialization overhead for the iterator for the complement amortizes. After that, this effect is overrun by an increasing number of cache misses of the algorithm `DPcsg`. After $n = 19$ the cache misses for `SubAlt` also increase and thus the relative overhead decreases again.

Since star queries are of high practical relevance, we evaluated the scalability of all algorithms for star queries beyond $n = 20$. The results are shown in Figure 12.

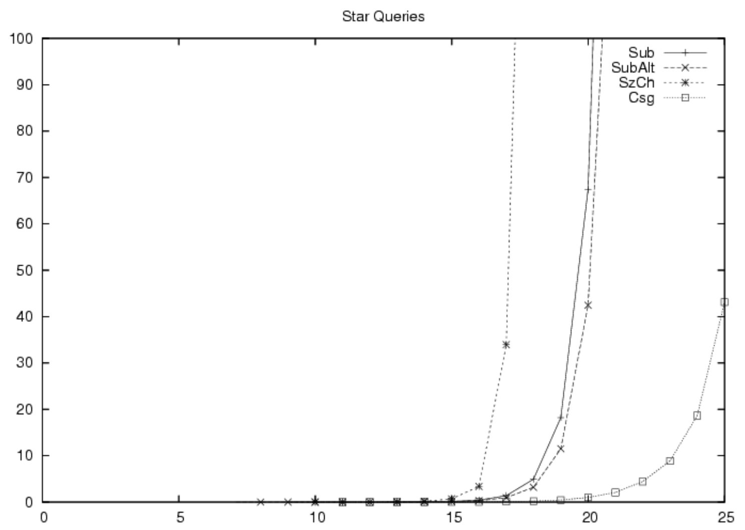

Figure 12: Scalability of the variants

6 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 `DPcsg`, which is highly superior to `DPsize` and `DPsub`. `DPcsg` should thus be the algorithm of choice when implementing a plan generator.

A DP-Variant Independent Results

This section analyses the join ordering problem for chain, cycle, and clique queries. For each kind of query graph, we calculate the number of connected subgraphs (#`csg`, Csg) and the number of rule applications (#`rap`,Rap). The number of connected subgraphs and the number of rule applications are independent of the dynamic programming algorithm used. Further, we analyse the value of *InnerCounter* of the DP-Variantes after termination. For the *SizeChained* variant, this value is dependent on the join graph. For the *Subsets* variant the value of *InnerCounter* after termination is independent of the query graph.

A.1 Chain Queries

Consider a chain of length n. Then, the number of connected subgraphs of size $k > 0$ is

$$\#\mathtt{csg}(n,k) = (n-k+1)$$

and, hence, the total number of non-empty connected subgraphs becomes

$$\begin{aligned} \#\mathtt{csg}(n) &= \sum_{k=1}^{n}(n-k+1) \\ &= \frac{n(n+1)}{2} \end{aligned} \tag{23}$$

To calculate the number of rule applications consider the following chain of size n:

$$R_1 \dots R_i \dots R_j \dots R_n$$

and assume that the subchain $R_i \dots R_j$ consists of k relations. This subchain can be joined with

- R_{i-1}, $R_{i-2}R_{i-1}$, ..., $R_1 \dots R_{i-1}$ and with
- R_{j+1}, $R_{j+1}R_{j+2}$, ..., $R_{j+1} \dots R_n$.

These are exactly $n-k$ subchains. Hence, the number of rule applications can be calculated as

$$\begin{aligned} \#\mathtt{rap} &= \sum_{k=1}^{n-1}(n-k+1)(n-k) \\ &= \sum_{k=1}^{n-1} k(k+1) \\ &= 2\binom{n+1}{3} \\ &= \frac{1}{3}((n+1)^3 - (n+1)^2 + 2 * (n+1)) \end{aligned} \tag{24}$$

A.2 Cycle Queries

The number of connected subgraphs of size $k > 0$ of a cyle of size n is

$$\#\mathtt{csg}(n,k) = \begin{cases} 1 & n = k \\ n & \text{else} \end{cases} \tag{25}$$

The number of non-empty connected subgraphs of a cylce of size n is easy to calculate. For every relation R_i, we have $n-1$ connected subgraphs. However, we have to be careful not to count the subgraph of size n, i.e. the whole graph, multiple times as it occurs only once. Hence, the number of non-empty connected subgraphs of a cycle of size n is

$$\#\mathtt{csg}(n) = n^2 - n + 1 \tag{26}$$

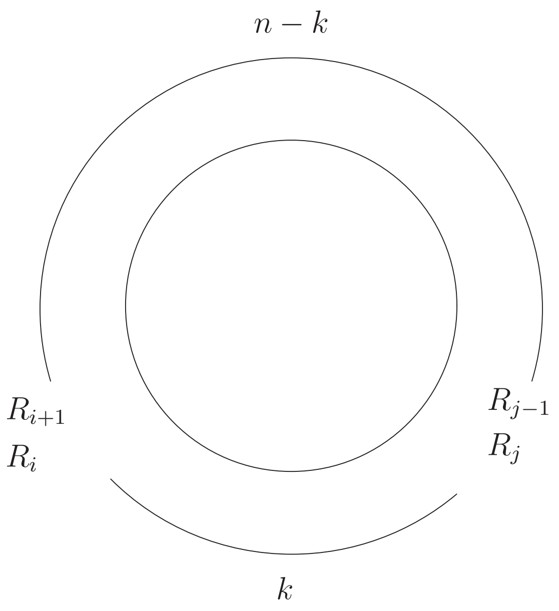

Figure 13: A cyclic join query

To calculate the number of rule applications, consider Figure 13. Given a connected subgraph of size k, there remain $n-k$ relations. Every relation gives rise to two join partners. However, we have to be careful to not count the join-partner consisting of all $n-k$ relations twice. Hence, the number of rule applications is

$$\begin{aligned}\#\mathtt{rap} &= \sum_{k=1}^{n-1} n(2(n-k)-1) \\ &= n^3 - 2n^2 + n \qquad (27)\end{aligned}$$

A.3 Star Queries

The number of connected subgraphs of size $k > 0$ of a cyle of size n is

$$\#\mathtt{csg}(n,k) = \begin{cases} n & k = 1 \\ \binom{n-1}{k-1} & k > 1 \end{cases}$$

The total number of non-empty connected subgraphs is

$$\begin{aligned}\#\mathtt{csg}(n) &= \sum_{k=1}^{n} \#\mathtt{csg}(n,k) \\ &= n + \sum_{k=2}^{n} \binom{n-1}{k-1} \\ &= n + \sum_{k=1}^{n-1} \binom{n-1}{k} \\ &= 2^{n-1} + n - 1\end{aligned}$$

The number of rule applications is

$$\#\texttt{rap} = (n-1)2^{n-2} \tag{28}$$

This result is due to Ono and Lohman [5]

A.4 Clique Queries

The number of connected subgraphs of size $k > 0$ of a clique of size n is

$$\#\texttt{csg}(n,k) = \binom{n}{k} \tag{29}$$

Number of non-empty connected subgraphs of a clique of size n:

$$\#\texttt{csg}(n) = 2^n - 1 \tag{30}$$

For a clique every subset of vertices is a connected subgraph. Hence, the total number of rule applications can be calculated as follows:

$$\begin{aligned}\#\texttt{rap} &= \sum_{k=1}^{n}\binom{n}{k}\sum_{i=1}^{k-1}\binom{k}{i} \\ &= 3^n - 2^{n+1} + 1 \end{aligned} \tag{31}$$

The second equality follow from

$$\begin{aligned}\sum_{k=1}^{n}\binom{n}{k}\sum_{i=1}^{k-1}\binom{k}{i} &= \sum_{k=1}^{n}\binom{n}{k}\sum_{i=0}^{k}\binom{k}{i} - 2\sum_{k=1}^{n}\binom{n}{k} \\ &= \sum_{k=0}^{n}\binom{n}{k}\sum_{i=0}^{k}\binom{k}{i} - 1 - 2(2^n-1) \\ &= \sum_{k=0}^{n}2^k\binom{n}{k} - 2^{n+1} + 1 \\ &= 3^n - 2^{n+1} + 1 \end{aligned} \tag{32}$$

where we used Identity 50 in the last step. This result is due to Ono and Lohman [5].

B Dynamic Programming Variant Size-Chained

B.1 Dynamic Programming Variant Size Chained: General Remarks

We first calculate the number of executions of the innermost loop in general terms (#csg). Then, we instanciate the resulting sums for the different query graphs.

Given #`csg`, the number of executions of the innermost loop follows directly from the pseudocode:

$$I_{\mathrm{SzCh}}(n) = \sum_{k=1}^{n-1} \#\mathtt{csg}(n,k) \sum_{i=1}^{\min(k,n-k)} f(n,k,i) \tag{33}$$

where

$$f(n,k,i) = \begin{cases} (\#\mathtt{csg}(n,i)-1)/2 & (i=k) \\ \#\mathtt{csg}(n,i) & (i \neq k) \end{cases}$$

The following table illustrates the situation for even n:

$k\backslash i$	0	1	2	...	$n/2-2$	$n/2-1$	$n/2$	$n/2+1$	$n/2+2$	...	$n-2$	$n-1$	n
0	$*$	–	–	–	–	–	–	–	–	–	–	–	–
1	–	$1*$											–
2	–	1	$1*$										–
...	–	1	1	$1*$									–
$n/2-2$	–	1	1	1	$1*$								–
$n/2-1$	–	1	1	1	1	$1*$							–
$n/2$	–	2	2	2	2	2	$2*$						–
$n/2+1$	–	2	2	2	2	2		$*$					–
$n/2+2$	–	2	2	2	2				$*$				–
...	–	2	2	2						$*$			–
$n-2$	–	2	2								$*$		–
$n-1$	–	2										$*$	–
n	–	–	–	–	–	–	–	–	–	–	–	–	–

In the matrix, k and i take values for 0 and n which our original sum does not. Therefore, the according fields are marked by $-$. The diagonal is labeled with $*$'s. This is where for $f(n,k,i)$, we have to take the case $i=k$. As long as $k < n/2$, we have $\min(k, n-k) = k$. We have

$$\min(k,n-k) = \begin{cases} k & k < n/2 \\ n/2 & k = n/2 \\ n-k & k > n/2 \end{cases}$$

We will split the sum into two sums: the first sum will cover the part where $k < n/2$ (labelled with 1 in the matrix), the second sum the rest (labelled with 2). Most of the diagonal will be covered by the first sum; only the case $k = n/2 \wedge i = n/2$ is covered by the second sum.

The two cases in f's definition and the minimum in the upper bound of the second sum disturb a direct summation. In a first step, we get rid of them. Let us call the above sum $I(n)$. We first split the sum to cover the diagonal case.

$$I_{\mathrm{SzCh}}(n) = \sum_{k=1}^{n-1} \sum_{i=1}^{\min(k,n-k)} f_1(n,k,i) + \sum_{k=1}^{\lfloor n/2 \rfloor} \sum_{i=1}^{\min(k,n-k)} f_2(n,k,i)$$

where

$$f_1(n,k,i) = \begin{cases} 0 & (i=k) \\ \#\mathtt{csg}(n,k)\#\mathtt{csg}(n,i) & (i \neq k) \end{cases}$$

and

$$f_2(n,k,i) = \begin{cases} \#\texttt{csg}(n,k)(\#\texttt{csg}(n,i)-1)/2 & (i=k) \\ 0 & (i \neq k) \end{cases}$$

In the next step, we split the first sum in order to get rid of the minimum calculation later on. This results in

$$I_{\mathrm{SzCh}}(n) = \sum_{k=1}^{\lfloor n/2 \rfloor -1} \sum_{i=1}^{\min(k,n-k)} f_1(n,k,i) + \sum_{k=\lfloor n/2 \rfloor}^{n-1} \sum_{i=1}^{\min(k,n-k)} f_1(n,k,i) + \sum_{k=1}^{\lfloor n/2 \rfloor} \sum_{i=1}^{\min(k,n-k)} f_2(n,k,i)$$

In order to calculate the third sum, we must distinguish whether n is odd or even. We add a correcting summand in case n is even. Additionally, we remove the minimum calcuations. This results in

$$I_{\mathrm{SzCh}}(n) = \sum_{k=1}^{\lfloor (n+1)/2 \rfloor -1} \sum_{i=1}^{k-1} f_1(n,k,i) + \sum_{k=\lfloor (n+1)/2 \rfloor}^{n-1} \sum_{i=1}^{n-k} f_1(n,k,i) + \sum_{k=1}^{\lfloor n/2 \rfloor} f_2(n,k,k) - f_{\mathrm{even}}(n)$$

where

$$f_{\mathrm{even}}(n) = \begin{cases} \#\texttt{csg}(n,n/2)\#\texttt{csg}(n,n/2) & (n \mod 2 \equiv 0) \\ 0 & \text{else} \end{cases}$$

With

$$\begin{aligned} S_1(n) &= \sum_{k=1}^{\lfloor (n+1)/2 \rfloor -1} \sum_{i=1}^{k-1} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) \\ S_2(n) &= \sum_{k=\lfloor (n+1)/2 \rfloor}^{n-1} \sum_{i=1}^{n-k} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) \\ S_3(n) &= \sum_{k=1}^{\lfloor n/2 \rfloor} \#\texttt{csg}(n,k)(\#\texttt{csg}(n,k)-1)/2 \end{aligned}$$

we now have

$$I_{\mathrm{SzCh}}(n) = S_1(n) + S_2(n) + S_3(n) - f_{\mathrm{even}}(n)$$

For n even, we have:

$$\begin{aligned} I_{\mathrm{SzCh}}(n) &= \sum_{k=1}^{n/2-1} \sum_{i=1}^{k-1} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) + \sum_{k=n/2}^{n-1} \sum_{i=1}^{n-k} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) \\ &\quad + \sum_{k=1}^{(n/2)} \#\texttt{csg}(n,k)(\#\texttt{csg}(n,k)-1)/2 - \#\texttt{csg}(n,n/2)\#\texttt{csg}(n,n/2) \end{aligned}$$

and for n odd, we have

$$\begin{aligned} I_{\mathrm{SzCh}}(n) &= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) + \sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) \\ &\quad + \sum_{k=1}^{(n-1)/2} \#\texttt{csg}(n,k)(\#\texttt{csg}(n,k)-1)/2 \end{aligned}$$

B.2 Dynamic Programming Variant Size Chained: Chain

We have $\#\texttt{csg}(n,k) = (n-k+1)$ for all $k < n$.

B.2.1 Case I: n even

For n even, we have:

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) + \sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) \\
&\quad + \sum_{k=1}^{(n/2)} \#\texttt{csg}(n,k)(\#\texttt{csg}(n,k)-1)/2 - \#\texttt{csg}(n,n/2)\#\texttt{csg}(n,n/2) \\
&= \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1}(n-k+1)(n-i+1) + \sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k}(n-k+1)(n-i+1) \\
&\quad + \sum_{k=1}^{(n/2)} (n-k+1)((n-k+1)-1)/2 \quad - (n-(n/2)+1)^2 \\
&= \sum_{k=1}^{n/2-1} (n-k+1)\sum_{i=1}^{k-1}(n-i+1) + \sum_{k=n/2}^{n-1} (n-k+1)\sum_{i=1}^{n-k}(n-i+1) \\
&\quad + \sum_{k=1}^{(n/2)} (n-k+1)(n-k)/2 \quad - ((n/2)+1)^2
\end{aligned}$$

First, we consider each sum separately.

Consider the first sum:

$$\begin{aligned}
S_1 &= \sum_{k=1}^{n/2-1}(n-k+1)\sum_{i=1}^{k-1}(n-i+1)\\
&= \sum_{k=1}^{n/2-1}(n-k+1)(n(k-1)-\frac{1}{2}k(k-1)+(k-1))\\
&= \sum_{k=1}^{n/2-1}(n-(k-1))(k-1)(n-\frac{1}{2}k+1)\\
&= \sum_{k=1}^{n/2-1}n(k-1)(n+1-\frac{1}{2}k)-\sum_{k=1}^{n/2-1}(k-1)(k-1)(n+1-\frac{1}{2}k)\\
&= n(n+1)\sum_{k=1}^{n/2-1}(k-1)-\frac{1}{2}n\sum_{k=1}^{n/2-1}(k-1)k\\
&\quad -(n+1)\sum_{k=1}^{n/2-1}(k-1)(k-1)+\frac{1}{2}\sum_{k=1}^{n/2-1}(k-1)(k-1)k\\
&= n(n+1)\sum_{k=1}^{n/2-2}k-\frac{1}{2}n\sum_{k=1}^{n/2-1}(k-1)k\\
&\quad -(n+1)\sum_{k=1}^{n/2-2}k^2+\frac{1}{2}\sum_{k=1}^{n/2-1}(k-1)(k-1)k\\
&= n(n+1)\binom{n/2-1}{2}-\frac{1}{2}n2\binom{n/2}{3}\\
&\quad -(n+1)(2\binom{n/2}{3}-\binom{n/2-1}{2})+\frac{1}{2}(6\binom{n/2}{4}+2\binom{n/2}{3})\\
&= n(n+1)\binom{n/2-1}{2}-n\binom{n/2}{3}\\
&\quad -(n+1)(2\binom{n/2}{3}-\binom{n/2-1}{2})+(3\binom{n/2}{4}+\binom{n/2}{3})\\
&= n(n+1)\binom{n/2-1}{2}+(n+1)\binom{n/2-1}{2}-n\binom{n/2}{3}\\
&\quad -(n+1)2\binom{n/2}{3}+\binom{n/2}{3}+3\binom{n/2}{4}\\
&= (n+1)(n+1)\binom{n/2-1}{2}+3\binom{n/2}{4}-(3n+1)\binom{n/2}{3}
\end{aligned}$$

Consider the second sum:

$$\begin{aligned}
S_2 &= \sum_{k=n/2}^{n-1}(n-k+1)\sum_{i=1}^{n-k}(n-i+1) \\
&= \sum_{k=n/2}^{n-1}(n-k+1)\sum_{i=1}^{n-k}(k+i) \\
&= \sum_{k=n/2}^{n-1}(n-k+1)(k(n-k)+\binom{n-k+1}{2}) \\
&= \sum_{k=1}^{n/2}(n-(n-k)+1)(((n-k)(n-(n-k))+\binom{n-(n-k)+1}{2})) \\
&= \sum_{k=1}^{n/2}(k+1)((n-k)k+\binom{k+1}{2}) \\
&= n\sum_{k=1}^{n/2}k(k+1)-\sum_{k=1}^{n/2}k^2(k+1)+\sum_{k=1}^{n/2}(k+1)\binom{k+1}{2}) \\
&= 2n\binom{n/2+2}{3} \\
&\quad -(6\binom{n/2+2}{4}+\binom{n/2+1}{2}+2\binom{n/2+2}{3}-\binom{n/2+1}{2}) \\
&\quad +3\binom{n/2+3}{4}-1\binom{n/2+2}{3} \\
&= 2n\binom{n/2+2}{3}-3\binom{n/2+2}{3}-6\binom{n/2+2}{4}+3\binom{n/2+3}{4} \\
&= 2n\binom{n/2+2}{3}-3\binom{n/2+2}{3}-3\binom{n/2+2}{4}+3(\binom{n/2+3}{4}-\binom{n/2+2}{4}) \\
&= 2n\binom{n/2+2}{3}-3\binom{n/2+2}{3}-3\binom{n/2+2}{4}+3\binom{n/2+2}{3} \\
&= 2n\binom{n/2+2}{3}-3\binom{n/2+2}{4}
\end{aligned}$$

Consider the third sum.

$$\begin{aligned}
S_3 &= \sum_{k=1}^{(n/2)}(n-k+1)(n-k) \\
&= \sum_{k=1}^{n}k(k-1)-\sum_{k=1}^{n/2}k(k-1) \\
&= 2(\binom{n+1}{3}-\binom{n/2+1}{3})
\end{aligned}$$

Altogether we now have

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= (n+1)(n+1)\binom{n/2-1}{2} + 3\binom{n/2}{4} - (3n+1)\binom{n/2}{3} \\
&\quad +2n\binom{n/2+2}{3} - 3\binom{n/2+2}{4} \\
&\quad +\binom{n+1}{3} - \binom{n/2+1}{3} \\
&\quad -((n/2)+1)^2
\end{aligned}$$

Let us simplify this expression by first considering

$$\begin{aligned}
3\binom{n/2+2}{4} - 3\binom{n/2}{4} &= \frac{3}{4!}[(n/2+2)(n/2+1)n/2(n/2-1) - n/2(n/2-1)(n/2-2)(n/2-3)] \\
&= \frac{3}{4!}[n/2(n/2-1)((n/2+2)(n/2+1) - (n/2-2)(n/2-3))] \\
&= \frac{3}{3*4}[((n/2)^2 + 3n/2 + 2) - ((n/2)^2 - 5n/2 + 6)]\binom{n/2}{2} \\
&= \frac{1}{4}[8n/2 - 4]\binom{n/2}{2} \\
&= (n-1)\binom{n/2}{2}
\end{aligned}$$

Using this leaves us with

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= \binom{n+1}{3} + (n+1)(n+1)\binom{n/2-1}{2} - (n-1)\binom{n/2}{2} - (3n+1)\binom{n/2}{3} \\
&\quad +2n\binom{n/2+2}{3} - \binom{n/2+1}{3} - ((n/2)+1)^2
\end{aligned}$$

Using

$$\begin{aligned}
2n\binom{n/2+2}{3} - \binom{n/2+1}{3} &= 2n(\binom{n/2+2}{3} - \binom{n/2+1}{3}) + (2n-1)\binom{n/2+1}{3} \\
&= 2n\binom{n/2+1}{2} + (2n-1)\binom{n/2+1}{3}
\end{aligned}$$

yields

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= \binom{n+1}{3} + (n+1)(n+1)\binom{n/2-1}{2} + 2n\binom{n/2+1}{2} \\
&\quad -(n-1)\binom{n/2}{2} + (2n-1)\binom{n/2+1}{3} - (3n+1)\binom{n/2}{3} - ((n/2)+1)^2 \\
&= \binom{n+1}{3} + (n+1)(n+1)\binom{n/2-1}{2} + 2n\binom{n/2+1}{2} \\
&\quad +n\binom{n/2}{2} - (n+2)\binom{n/2}{3} - ((n/2)+1)^2
\end{aligned}$$

by using

$$\begin{aligned}
(2n-1)\binom{n/2+1}{3} - (3n+1)\binom{n/2}{3} &= (2n-1)(\binom{n/2+1}{3} - \binom{n/2}{3}) - (n+2)\binom{n/2}{3} \\
&= (2n-1)\binom{n/2}{2} - (n+2)\binom{n/2}{3}
\end{aligned}$$

Finally, adding up

$$\begin{aligned}
(n+1)(n+1)\binom{n/2-1}{2} &= 1/4(n^4/2 - 2n^3 - 3n^2/2 + 5n + 4) \\
-((n/2)+1)^2 &= -1/4(n^2 - 4n - 4) \\
2n\binom{n/2+1}{2} + n\binom{n/2}{2} &= 1/4(3 * n^3/2 + n^2)
\end{aligned}$$

yields

$$1/4(n^4/2 - n^3/2 - 3n^2/2 + n)$$

which leaves us with

$$I(n) = \binom{n+1}{3} - (n+2)\binom{n/2}{3} + 1/4(n^4/2 - n^3/2 - 3n^2/2 + n) \tag{34}$$

We continue with observing that

$$\begin{aligned}
\binom{n+1}{3} &= 1/6(n+1)n(n-1) \\
&= 1/6(n^3 - n) \\
(n+2)\binom{n/2}{3} &= 1/6(n+2)(n/2(n/2-1)(n/2-2)) \\
&= 1/6(n+2)n/2(n^2/4 - 3n/2 + 2) \\
&= 1/6(n+1)(n^3/8 - 3n^2/4 + n) \\
&= 1/6(n^4/8 - 3n^3/4 + n^2 + n^3/4 - 3n^2/2 + 2n) \\
&= 1/6(n^4/8 - n^3/2 - n^2/2 + 2n)
\end{aligned}$$

Subtracting these two items and adding the polynomial from $I(n)$ gives

$$\begin{aligned}
I_{\text{SzCh}}(n) &= 1/6(n^3 - n) - 1/6(n^4/8 - n^3/2 - n^2/2 + 2n) \\
&\quad +1/4(n^4/2 - n^3/2 - 3n^2/2 + n) \\
&= 1/12(-n^4/4 + 6n^3/2 + 2n^2/2 - 6n) \\
&\quad +1/12(3n^4/2 - 3n^3/2 - 9n^2/2 + 3n) \\
&= 1/12(5n^4/4 + 3n^3/2 - 7n^2/2 - 3n) \\
&= 1/48(5n^4 + 6n^3 - 14n^2 - 12n)
\end{aligned}$$

B.2.2 Case II: n odd

For n odd, we have:

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= \sum_{k=1}^{(n+1)/2-1}\sum_{i=1}^{k-1}\#\mathtt{csg}(n,k)\#\mathtt{csg}(n,i) + \sum_{k=(n+1)/2}^{n-1}\sum_{i=1}^{n-k}\#\mathtt{csg}(n,k)\#\mathtt{csg}(n,i) \\
&\quad + \sum_{k=1}^{(n-1)/2}\#\mathtt{csg}(n,k)(\#\mathtt{csg}(n,k)-1)/2 \\
&= \sum_{k=1}^{(n+1)/2-1}\sum_{i=1}^{k-1}(n-k+1)(n-i+1) + \sum_{k=(n+1)/2}^{n-1}\sum_{i=1}^{n-k}(n-k+1)(n-i+1) \\
&\quad + \sum_{k=1}^{(n-1)/2}(n-k+1)((n-k+1)-1)/2 \\
&= \sum_{k=1}^{\lfloor n/2\rfloor}(n-k+1)\sum_{i=1}^{k-1}(n-i+1) + \sum_{k=\lceil n/2\rceil}^{n-1}(n-k+1)\sum_{i=1}^{n-k}(n-i+1) \\
&\quad + \sum_{k=1}^{\lfloor n/2\rfloor}(n-k+1)((n-k+1)-1)/2
\end{aligned}$$

First, we consider each sum separately.

Consider the first sum:

$$
\begin{aligned}
S_1 &= \sum_{k=1}^{(n+1)/2-1} (n-k+1)\sum_{i=1}^{k-1}(n-i+1) \\
&= \sum_{k=1}^{\lfloor n/2 \rfloor} (n-k+1)(n(k-1) - \frac{1}{2}k(k-1) + (k-1)) \\
&= \sum_{k=1}^{\lfloor n/2 \rfloor} (n-(k-1))(k-1)(n - \frac{1}{2}k + 1) \\
&= \sum_{k=1}^{\lfloor n/2 \rfloor} n(k-1)(n+1-\frac{1}{2}k) - \sum_{k=1}^{\lfloor n/2 \rfloor} (k-1)(k-1)(n+1-\frac{1}{2}k) \\
&= n(n+1)\sum_{k=1}^{\lfloor n/2 \rfloor}(k-1) - \frac{1}{2}n\sum_{k=1}^{\lfloor n/2 \rfloor}(k-1)k \\
&\quad -(n+1)\sum_{k=1}^{\lfloor n/2 \rfloor}(k-1)(k-1) + \frac{1}{2}\sum_{k=1}^{\lfloor n/2 \rfloor}(k-1)(k-1)k \\
&= n(n+1)\sum_{k=1}^{\lfloor n/2-1 \rfloor} k - \frac{1}{2}n\sum_{k=1}^{\lfloor n/2 \rfloor}(k-1)k \\
&\quad -(n+1)\sum_{k=1}^{\lfloor n/2-1 \rfloor} k^2 + \frac{1}{2}\sum_{k=1}^{\lfloor n/2 \rfloor}(k-1)(k-1)k \\
&= n(n+1)\binom{\lfloor n/2 \rfloor}{2} - \frac{1}{2}n2\binom{\lfloor n/2 \rfloor + 1}{3} \\
&\quad -(n+1)(2\binom{\lfloor n/2 \rfloor + 1}{3} - \binom{\lfloor n/2 \rfloor}{2}) + \frac{1}{2}(6\binom{\lfloor n/2 \rfloor + 1}{4} + 2\binom{\lfloor n/2 \rfloor + 1}{3}) \\
&= 3\binom{\lfloor n/2 \rfloor + 1}{4} + (n+1)(n+1)\binom{\lfloor n/2 \rfloor}{2} - (3n+1)\binom{\lfloor n/2 \rfloor + 1}{3}
\end{aligned}
$$

Consider the second sum:

$$\begin{aligned}
S_2 &= \sum_{k=(n+1)/2}^{n-1} (n-k+1) \sum_{i=1}^{n-k} (n-i+1) \\
&= \sum_{k=\lceil n/2 \rceil}^{n-1} (n-k+1) \sum_{i=1}^{n-k} (k+i) \\
&= \sum_{k=\lceil n/2 \rceil}^{n-1} (n-k+1)(k(n-k) + \binom{n-k+1}{2}) \\
&= \sum_{k=1}^{\lfloor n/2 \rfloor} (n-(n-k)+1)(((n-k)(n-(n-k)) + \binom{n-(n-k)+1}{2})) \\
&= \sum_{k=1}^{\lfloor n/2 \rfloor} (k+1)((n-k)k + \binom{k+1}{2}) \\
&= n \sum_{k=1}^{\lfloor n/2 \rfloor} k(k+1) - \sum_{k=1}^{\lfloor n/2 \rfloor} k^2(k+1) + \sum_{k=1}^{\lfloor n/2 \rfloor} (k+1)\binom{k+1}{2}) \\
&= 2n\binom{\lfloor n/2 \rfloor + 2}{3} \\
&\quad -(6\binom{\lfloor n/2 \rfloor + 2}{4} + \binom{\lfloor n/2 \rfloor + 1}{2} + 2\binom{\lfloor n/2 \rfloor + 2}{3} - \binom{\lfloor n/2 \rfloor + 1}{2}) \\
&\quad +3\binom{\lfloor n/2 \rfloor + 3}{4} - \binom{\lfloor n/2 \rfloor + 2}{3} \\
&= 2n\binom{\lfloor n/2 \rfloor + 2}{3} - 3\binom{\lfloor n/2 \rfloor + 2}{3} - 6\binom{\lfloor n/2 \rfloor + 2}{4} + 3\binom{\lfloor n/2 \rfloor + 3}{4} \\
&= 2n\binom{\lfloor n/2 \rfloor + 2}{3} + 3(\binom{\lfloor n/2 \rfloor + 3}{4} - \binom{\lfloor n/2 \rfloor + 2}{3}) - 6\binom{\lfloor n/2 \rfloor + 2}{4} \\
&= 2n\binom{\lfloor n/2 \rfloor + 2}{3} - 3\binom{\lfloor n/2 \rfloor + 2}{4}
\end{aligned}$$

Consider the third sum.

$$\begin{aligned}
S_3 &= 1/2 \sum_{k=1}^{\lfloor n/2 \rfloor} (n-k+1)(n-k) \\
&= 1/2(\sum_{k=1}^{n} k(k-1) - \sum_{k=1}^{\lfloor n/2 \rfloor + 1} k(k-1)) \\
&= \binom{n+1}{3} - \binom{\lfloor n/2 \rfloor + 2}{3}
\end{aligned}$$

Altogether we now have

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= 3\binom{\lfloor n/2\rfloor+1}{4} + (n+1)^2\binom{\lfloor n/2\rfloor}{2} - (3n+1)\binom{\lfloor n/2\rfloor+1}{3} \\
&\quad +2n\binom{\lfloor n/2\rfloor+2}{3} - 3\binom{\lfloor n/2\rfloor+2}{4} \\
&\quad +\binom{n+1}{3} - \binom{\lfloor n/2\rfloor+2}{3} \\
&= \binom{n+1}{3} + (n+1)^2\binom{\lfloor n/2\rfloor}{2} - (3n+4)\binom{\lfloor n/2\rfloor+1}{3} + (2n-1)\binom{\lfloor n/2\rfloor+2}{3} \\
&= \binom{n+1}{3} + (n+1)^2\binom{\lfloor n/2\rfloor}{2} + (2n-1)(\binom{\lfloor n/2\rfloor+2}{3} - \binom{\lfloor n/2\rfloor+1}{3}) \\
&\quad -(n+5)\binom{\lfloor n/2\rfloor+1}{3} \\
&= \binom{n+1}{3} + (n+1)^2\binom{\lfloor n/2\rfloor}{2} + (2n-1)\binom{\lfloor n/2\rfloor+1}{2} - (n+5)\binom{\lfloor n/2\rfloor+1}{3}
\end{aligned}$$

Considering each summand separately

$$\begin{aligned}
\binom{n+1}{3} &= 1/6(n+1)n(n-1) \\
&= 1/6(n^3-n) \\
&= 1/6n^3 - 1/6n \\
(n+1)^2\binom{\lfloor n/2\rfloor}{2} &= 1/2(n+1)^2\lfloor n/2\rfloor(\lfloor n/2\rfloor-1) \\
&= 3/6(n+1)^2\lfloor n/2\rfloor^2 - 1/6(3n^2+6n+3)\lfloor n/2\rfloor \\
(2n-1)\binom{\lfloor n/2\rfloor+1}{2} &= 1/2(2n-1)\lfloor n/2\rfloor(\lfloor n/2\rfloor+1) \\
&= 1/6(6n-3)\lfloor n/2\rfloor^2 + 1/6(6n-3)\lfloor n/2\rfloor \\
-(n+5)\binom{\lfloor n/2\rfloor+1}{3} &= 1/6(5-n)\lfloor n/2\rfloor^3 - 1/6(5-n)\lfloor n/2\rfloor
\end{aligned}$$

and summing up again yields

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= 1/6(n^3 - n + (n+1)^2\lfloor n/2\rfloor^2 - (3n^2+6n+3)\lfloor n/2\rfloor \\
&\quad +(6n-3)\lfloor n/2\rfloor^2 + (6n-3)\lfloor n/2\rfloor - n(\lfloor n/2\rfloor^3 - \lfloor n/2\rfloor) - 5(\lfloor n/2\rfloor^3 - \lfloor n/2\rfloor)) \\
&= 1/48(8n^3 - 8n + 24(n+1)^2\lfloor n/2\rfloor^2 - 8(3n^2+6n+3)\lfloor n/2\rfloor \\
&\quad +8(6n-3)\lfloor n/2\rfloor^2 + 8(6n-3)\lfloor n/2\rfloor - 8n(\lfloor n/2\rfloor^3 - \lfloor n/2\rfloor) - 40(\lfloor n/2\rfloor^3 - \lfloor n/2\rfloor)) \\
&= 1/48(8n^3 - 8n + 6(n+1)^2(n-1)^2 - 12n^3 - 24n^2 - 12n + 12n^2 + 24n + 12 \\
&\quad +(12n-6)*(n^2-2n+1) + (24n-12)*(n-1) \\
&\quad -8n\lfloor n/2\rfloor^3 + 4n(n-1) - 40\lfloor n/2\rfloor^3 + 20(n-1)) \\
&= 1/48(+8n^3 - 8n + 6n^4 - 12n^2 + 6 - 12n^3 - 24n^2 - 12n + 12^2n + 24n + 12 \\
&\quad +12n^3 - 24n^2 + 12n - 6n^2 + 12n - 6 + 24n^2 - 24n - 12n + 12 \\
&\quad -n^4 + 3n^3 - 3n^2 + n + 4n^2 - 4n - 5n^3 + 15n^2 - 15n + 15 + 20n - 20) \\
&= 1/48(5n^4 + 6n^3 - 14n^2 - 6n + 11)
\end{aligned}$$

B.3 Dynamic Programming Variant Size Chained: Cycle

We have $\#\texttt{csg}(n,k) = n$ for all $k < n$.

For n even, we have:

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1} n^2 + \sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k} n^2 + \sum_{k=1}^{(n/2)} n(n-1)/2 - n^2 \\
&= n^2(\sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1} 1) + n^2(\sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k} 1) + n(n-1)/2 \sum_{k=1}^{(n/2)} 1 - n^2 \\
&= n^2(\sum_{k=1}^{n/2-1} (k-1)) + n^2(\sum_{k=n/2}^{n-1} (n-k)) + n(n-1)/2(n/2) - n^2 \\
&= n^2(((n/2)*((n/2)-1)/2) - ((n/2)-1)) \\
&\quad + n^2(((n/2)*(n/2+1))/2) + n^2((n-1)/4) - n^2 \\
&= \frac{1}{8}n^2[n*(n-2) - (4n-8) + n*(n+2) + (2n-2) - 8] \\
&= \frac{1}{8}n^2[n^2 - 2n - 4n + 8 + n^2 + 2n + 2n - 2 - 8] \\
&= \frac{1}{8}n^2[2n^2 - 2n - 2] \\
&= \frac{1}{4}n^2[n^2 - n - 1] \\
&= \frac{1}{4}(n^4 - n^3 - n^2)
\end{aligned}$$

For n odd, we have:

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} n^2 + \sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} n^2 + \sum_{k=1}^{(n-1)/2} n(n-1)/2 \\
&= n^2(\sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} 1) + n^2(\sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} 1) + n(n-1)/2(\sum_{k=1}^{(n-1)/2} 1) \\
&= n^2(\sum_{k=1}^{(n+1)/2-1} (k-1)) + n^2(\sum_{k=(n+1)/2}^{n-1} (n-k)) + (n(n-1)/2)((n-1)/2) \\
&= n^2(((((n+1)/2)-2) * (((n+1)/2)-1))/2) \\
&\quad + n^2((((n-1)/2) * (((n-1)/2)+1))/2) \\
&\quad + \frac{1}{4} n(n-1)^2 \\
&= \frac{1}{8} n^2[((n+1)-4) * ((n+1)-2) + (n-1) * ((n-1)+2)] \\
&\quad + \frac{1}{4} n(n-1)^2 \\
&= \frac{1}{8} n^2[(n-3) * (n-1) + (n-1) * (n+1)] + \frac{1}{4} n(n-1)^2 \\
&= \frac{1}{8} n^2 (n-1)[(n-3) + (n+1)] + \frac{1}{4} n(n-1)^2 \\
&= \frac{1}{4} n^2 (n-1)[n-1] + \frac{1}{4} n(n-1)^2 \\
&= \frac{1}{4} n^2 (n-1)^2 + \frac{1}{4} n(n-1)^2 \\
&= \frac{1}{4} (n-1)^2 n(n+1) \\
&= \frac{1}{4} (n^4 - n^3 - n^2 + n)
\end{aligned}$$

B.4 Matrix of Binomial Coefficients

For analyzing the behavior of the size-chained variant for stars and cliques, we have to consider a matrix M whose values are the result of products of binomial coefficients. The sums S_1 and S_2 then cover a certain area in the matrix. A direct summation of the values in this area is not possible. Hence, we take a direct approach by calculating the sum of all values in the matrix and subtracting those parts that are not covered by S_1 or S_2.

Define $M(n)$ to be the $(n \times n)$-matrix with

$$M_{k,i} = \binom{n}{k}\binom{n}{i}$$

in cell (k, i). The matrix is highly symmetrical, as the following holds: $M_{k,i} = M_{i,k} = M_{n-k,i} = M_{k,n-i} = M_{n-k,n-i}$ and so on.

The sum over all values stored in $M(n)$ is

$$\sum_{k=0}^{n}\sum_{i=0}^{n}\binom{n}{k}\binom{n}{i} = 2^{2n} \tag{35}$$

We will be interested in the sum of those values of M which are contained in some regular fragment of M. The first fragment we are interested in excludes the values on the diagonals. The second fragment excludes the diagonals, the borders, and, in case n is even, the middle column and row. Since the calculation of the first part follows from the second fragment, we start with the second fragment and only state the result for the first afterwards. The appendix contains some useful information for subsequent calculations.

To proceed any further, we have to distinguish whether n is even or odd. We start with the case where n is even. For the first subset $G_M(n)$, we wish to exclude

1. the sum or the values on the four borderlines ($k = 0$ or $i = 0$) four times
$$\sum_{j=0}^{n}\binom{n}{j} = 2^n$$
2. the sum of the values in the diagonal
$$\sum_{k=0}^{n}\binom{n}{k}\binom{n}{k} = \binom{2n}{n}$$
3. the sum of the values in the secondary diagonal ($k + i = n$)
$$\sum_{k=0}^{n}\binom{n}{k}\binom{n}{n-k} = \binom{2n}{n}$$
4. the sum of the values in the middle column and row ($k = n/2$ or $i = n/2$). That is, two times
$$\sum_{k=0}^{n}\binom{n}{n/2}\binom{n}{k} = 2^n\binom{n}{n/2}$$

Let us denote by G_M the sum of the remaining values of M. We can calculate the value G_M as follows

$$G_M \quad = \quad 2^{2n} - 4 * 2^n - 2\binom{2n}{n} - 2 * 2^n\binom{n}{n/2} + 8 + 4\binom{n}{n/2} + 3\binom{n}{n/2}\binom{n}{n/2}$$

We had to be careful not to subtract the cells where the excluded subsets overlap more than once. The cells are:

1. the corner cells (($k = 0$ or $k = n$) and ($i = 0$ or $i = n$))
2. the border middle cells (($k = n/2$ or $i = n/2$) and ($k = 0$, $k = n$, $i = 0$, or $i = n$))

3. the middle cell ($k = n/2$, $i = n/2$)

For n odd, the values we have to subtract to yield $G_M(n)$ are

1. the sum or the values on the four borderlines ($k = 0$ or $i = 0$) four times
$$\sum_{j=0}^{n} \binom{n}{j} = 2^n$$
2. the sum of the values in the diagonal
$$\sum_{k=0}^{n} \binom{n}{k}\binom{n}{k} = \binom{2n}{n}$$
3. the sum of the values in the secondary diagonal ($k + i = n$)
$$\sum_{k=0}^{n} \binom{n}{k}\binom{n}{n-k} = \binom{2n}{n}$$

For n odd, we can calculate the value G_M as follows

$$G_M(n) \quad = \quad 2^{2n} - 4 * 2^n - 2\binom{2n}{n} + 8$$

Some cells in the matrix are subtracted several times. This has to be corrected. The cells are:

- the corner cells (($k = 0$ or $k = n$) and ($i = 0$ or $i = n$))

Summarizing, we have

$$G_M(n) = \begin{cases} 2^{2n} - 4 * 2^n - 2\binom{2n}{n} + 8 - 2 * 2^n\binom{n}{n/2} + 4\binom{n}{n/2} + 3\binom{n}{n/2}\binom{n}{n/2} & n \text{ even} \\ 2^{2n} - 4 * 2^n - 2\binom{2n}{n} + 8 & n \text{ odd} \end{cases} \tag{36}$$

The second subset we are interested in only excludes the values in the diagonals. We denote the result by A_M and find

$$A_M(n) = \begin{cases} 2^{2n} - 2\binom{2n}{n} + \binom{n}{n/2}\binom{n}{n/2} & n \text{ even} \\ 2^{2n} - 2\binom{2n}{n} & n \text{ odd} \end{cases} \tag{37}$$

B.5 Dynamic Programming Variant Size Chained: Clique

B.5.1 Case I: n even

Again, consider the matrix from the beginning. Call M the $(n \times n)$-matrix that contains the value of $\binom{n}{k}\binom{n}{i}$ in cell $M_{k,i}$. Then, we can calculate the value of $I(n)$ as follows. Take G_M. One fourth of it covers most of the values we have to sum up for $I(n)$, except for

$$\begin{array}{cccccccc}
\binom{n}{0}\binom{n}{0} & \binom{n}{0}\binom{n}{1} & \binom{n}{0}\binom{n}{2} & \binom{n}{0}\binom{n}{3} & \cdots & \binom{n}{0}\binom{n}{n/2-3} & \binom{n}{0}\binom{n}{n/2-2} & \binom{n}{0}\binom{n}{n/2-1} \\
\binom{n}{1}\binom{n}{0} & \binom{n}{1}\binom{n}{1} & \binom{n}{1}\binom{n}{2} & \binom{n}{1}\binom{n}{3} & \cdots & \binom{n}{1}\binom{n}{n/2-3} & \binom{n}{1}\binom{n}{n/2-2} & \binom{n}{1}\binom{n}{n/2-1} \\
\binom{n}{2}\binom{n}{0} & \binom{n}{2}\binom{n}{1} & \binom{n}{2}\binom{n}{2} & \binom{n}{2}\binom{n}{3} & \cdots & \binom{n}{2}\binom{n}{n/2-3} & \binom{n}{2}\binom{n}{n/2-2} & \binom{n}{2}\binom{n}{n/2-1} \\
\binom{n}{3}\binom{n}{0} & \binom{n}{3}\binom{n}{1} & \binom{n}{3}\binom{n}{2} & \binom{n}{3}\binom{n}{3} & \cdots & \binom{n}{3}\binom{n}{n/2-3} & \binom{n}{3}\binom{n}{n/2-2} & \binom{n}{3}\binom{n}{n/2-1} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\binom{n}{n/2-3}\binom{n}{0} & \binom{n}{n/2-3}\binom{n}{1} & \binom{n}{n/2-3}\binom{n}{2} & \binom{n}{n/2-3}\binom{n}{3} & \cdots & \binom{n}{n/2-3}\binom{n}{n/2-3} & \binom{n}{n/2-3}\binom{n}{n/2-2} & \binom{n}{n/2-3}\binom{n}{n/2-1} \\
\binom{n}{n/2-2}\binom{n}{0} & \binom{n}{n/2-2}\binom{n}{1} & \binom{n}{n/2-2}\binom{n}{2} & \binom{n}{n/2-2}\binom{n}{3} & \cdots & \binom{n}{n/2-2}\binom{n}{n/2-3} & \binom{n}{n/2-2}\binom{n}{n/2-2} & \binom{n}{n/2-2}\binom{n}{n/2-1} \\
\binom{n}{n/2-1}\binom{n}{0} & \binom{n}{n/2-1}\binom{n}{1} & \binom{n}{n/2-1}\binom{n}{2} & \binom{n}{n/2-1}\binom{n}{3} & \cdots & \binom{n}{n/2-1}\binom{n}{n/2-3} & \binom{n}{n/2-1}\binom{n}{n/2-2} & \binom{n}{n/2-1}\binom{n}{n/2-1} \\
\binom{n}{n/2}\binom{n}{0} & \binom{n}{n/2}\binom{n}{1} & \binom{n}{n/2}\binom{n}{2} & \binom{n}{n/2}\binom{n}{3} & \cdots & \binom{n}{n/2}\binom{n}{n/2-3} & \binom{n}{n/2}\binom{n}{n/2-2} & \binom{n}{n/2}\binom{n}{n/2-1}
\end{array}$$

Figure 14: Q1 of M (clique)

1. the values on the diagonal from $k = 1$ to $k = n/2$. The sum of these values is equal to D_M where

$$\begin{aligned}
D_M &= \sum_{k=1}^{n/2} 1/2\binom{n}{k}(\binom{n}{k} - 1) \\
&= \sum_{k=0}^{n/2} 1/2\binom{n}{k}(\binom{n}{k} - 1) \\
&= 1/2(\sum_{k=0}^{n/2}\binom{n}{k}\binom{n}{k} - \sum_{k=0}^{n/2}\binom{n}{k}) \\
&= 1/4(\binom{2n}{n} - \binom{n}{n/2}\binom{n}{n/2} - 2^n - \binom{n}{n/2})) \\
&= 1/4\binom{2n}{n} + 1/4\binom{n}{n/2}\binom{n}{n/2} - 2^{n-2} - 1/4\binom{n}{n/2}
\end{aligned}$$

2. the middle row $k = n/2$ for $i = 1$ to $k = n/2-1$. The sum of these values is equal to $R_M - \binom{n}{n/2}\binom{n}{n/2}$ where

$$\begin{aligned}
R_M &= \sum_{i=1}^{n/2}\binom{n}{n/2}\binom{n}{i} \\
&= \binom{n}{n/2}(2^{n-1} + 1/2\binom{n}{n/2} - 1) \\
&= 2^{n-1}\binom{n}{n/2} + 1/2\binom{n}{n/2}\binom{n}{n/2} - \binom{n}{n/2}
\end{aligned}$$

3. the secondary diagonal for $k = n-1$ down to $k = n/2+1$ and $i = n-k$

The sum of these values is equal to $S_M - \binom{n}{n/2}\binom{n}{n/2}$ where

$$\begin{aligned} S_M &= \sum_{k=n/2}^{n-1} \binom{n}{k}\binom{n}{n-k} \\ &= \sum_{k=1}^{n/2-1} \binom{n}{k}\binom{n}{k} \\ &= 1/2\binom{2n}{n} + 1/2\binom{n}{n/2}\binom{n}{n/2} - 1 \end{aligned}$$

Summarizing, we have

$$\begin{aligned} I_{\text{SzCh}}(n) &= G_M/4 + D_M + R_M + S_M - 2\binom{n}{n/2}\binom{n}{n/2} \\ &= 2^{2n-2} - 2^n - 1/2\binom{2n}{n} - 2^{n-1}\binom{n}{n/2} + 2 + \binom{n}{n/2} + 3/4\binom{n}{n/2}\binom{n}{n/2} \\ &\quad +1/4\binom{2n}{n} - 1/4\binom{n}{n/2}\binom{n}{n/2} - 2^{n-2} - 1/4\binom{n}{n/2} \\ &\quad +2^{n-1}\binom{n}{n/2} + 1/2\binom{n}{n/2}\binom{n}{n/2} - \binom{n}{n/2} \\ &\quad +1/2\binom{2n}{n} + 1/2\binom{n}{n/2}\binom{n}{n/2} - 1 \\ &\quad -2\binom{n}{n/2}\binom{n}{n/2} \\ &= 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} - 1/4\binom{n}{n/2} + 1 \end{aligned}$$

B.5.2 Case II: n odd

Consider the matrix from the beginning again. We can calculate the value of $I_{\text{SzCh}}(n)$ for odd n as follows. Let us start with G_M. One fourth of it covers most of the values we have to sum up for $I_{\text{SzCh}}(n)$, except for

1. the values on the diagonal from $k = 1$ to $k = \lfloor n/2 \rfloor$. The sum of these values is equal to D_M where

$$\begin{aligned} D_M &= \sum_{k=1}^{\lfloor n/2 \rfloor} 1/2\binom{n}{k}(\binom{n}{k} - 1) \\ &= \sum_{k=0}^{\lfloor n/2 \rfloor} 1/2\binom{n}{k}(\binom{n}{k} - 1) \\ &= 1/2(\sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{k}\binom{n}{k} - \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{k}) \\ &= 1/4(\binom{2n}{n} - 2^n \end{aligned}$$

2. the secondary diagonal for $k = n - 1$ down to $k = \lfloor n/2 \rfloor + 1$ and $i = n - k$
   The sum of these values is equal to S_M where

$$\begin{aligned} S_M &= \sum_{k=\lceil n/2 \rceil}^{n-1} \binom{n}{k}\binom{n}{n-k} \\ &= \sum_{k=1}^{\lfloor n/2 \rfloor - 1} \binom{n}{k}\binom{n}{k} \\ &= 1/2\binom{2n}{n} - 1 \end{aligned}$$

For odd n we thus have

$$\begin{aligned} I_{\text{SzCh}}(n) &= G_M/4 + D_M + R_M + S_M \\ &= 2^{2n-2} - 2^n - 1/2\binom{2n}{n} + 2 \\ &\quad +1/4\binom{2n}{n} - 2^{n-2} \\ &\quad +1/2\binom{2n}{n} - 1 \\ &= 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} + 1 \end{aligned}$$

B.6 Dynamic Programming Variant Size Chained: Star

$$\#\texttt{csg}(n,k) = \begin{cases} n & k = 1 \\ \binom{n-1}{k-1} & k > 1 \end{cases}$$

B.6.1 Case I: n even

Consider the first sum ($k > 2$):

$$\begin{aligned}
S_1 &= \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1} \#\texttt{csg}(n,k) * \#\texttt{csg}(n,i) \\
&= \sum_{k=1}^{n/2-1}\sum_{i=2}^{k-1}\binom{n-1}{k-1}\binom{n-1}{i-1} + \sum_{k=2}^{n/2-1} n\binom{n-1}{k-1} \\
&= \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1}\binom{n-1}{k-1}\binom{n-1}{i-1} + \sum_{k=2}^{n/2-1} n\binom{n-1}{k-1} - \sum_{k=2}^{n/2-1}\binom{n-1}{k-1} \\
&= \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1}\binom{n-1}{k-1}\binom{n-1}{i-1} + \sum_{k=1}^{n/2-1} n\binom{n-1}{k-1} - \sum_{k=1}^{n/2-1}\binom{n-1}{k-1} - n + 1 \\
&= \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1}\binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1)\sum_{k=1}^{n/2-1}\binom{n-1}{k-1} - n + 1 \\
&= \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1}\binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1)(2^{n-2} - \binom{n-1}{(n/2)-1}) - n + 1
\end{aligned}$$

Define

$$C_1 = \sum_{k=1}^{n/2-1}\sum_{i=1}^{k-1}\binom{n-1}{k-1}\binom{n-1}{i-1} \tag{38}$$

$$K_1 = (n-1)(2^{n-2} - \binom{n-1}{(n/2)-1}) - n + 1 \tag{39}$$

Consider the second sum ($k > 2$):

$$\begin{aligned}
S_2 &= \sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) \\
&= \sum_{k=n/2}^{n-1}\sum_{i=2}^{n-k} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) + \sum_{k=n/2}^{n-1} n\binom{n-1}{k-1} \\
&= \sum_{k=n/2}^{n-1}\sum_{i=2}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + \sum_{k=n/2}^{n-1} n\binom{n-1}{k-1} \\
&= \sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + \sum_{k=n/2}^{n-1} n\binom{n-1}{k-1} - \sum_{k=n/2}^{n-1} \binom{n-1}{k-1} \\
&= \sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1) * \sum_{k=n/2}^{n-1} \binom{n-1}{k-1} \\
&= \sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1) * (2^{n-2} + \binom{n-1}{\lfloor (n-1)/2 \rfloor} - 1)
\end{aligned}$$

Define

$$C_2 = \sum_{k=n/2}^{n-1}\sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} \qquad (40)$$

$$K_1 = (n-1) * (2^{n-2} + \binom{n-1}{\lfloor (n-1)/2 \rfloor} - 1) \qquad (41)$$

Consider the third sum ($k > 2$):

$$\begin{aligned}
S_3 &= \sum_{k=1}^{(n/2)} \#\texttt{csg}(n,k)(\#\texttt{csg}(n,k)-1)/2 \\
&= \sum_{k=2}^{(n/2)} \binom{n-1}{k-1}(\binom{n-1}{k-1} - 1)/2 + n(n-1)/2 \\
&= \sum_{k=1}^{(n/2)} \binom{n-1}{k-1}(\binom{n-1}{k-1} - 1)/2 + n(n-1)/2 \\
&= 1/2 \sum_{k=0}^{(n/2)-1} \binom{n-1}{k}(\binom{n-1}{k} - 1) + 1/2n(n-1) \\
&= 1/2 \sum_{k=0}^{\lfloor (n-1)/2 \rfloor} \binom{n-1}{k}(\binom{n-1}{k} - 1) + 1/2n(n-1) \\
&= 1/4(\binom{2(n-1)}{n-1} - 2^{n-1}) + 1/2n(n-1)
\end{aligned}$$

One fourth of matrix M without diagonals exactly covers C_1 and C_2. Note that the correcting term for the center is excluded by $A_M(n-1)$ and hence does not have to be subtracted. Summarizing, we have

$$\begin{aligned}
I_{\mathrm{SzCh}}(n) &= A_M(n-1)/4 + K_1 + K + 2 + S_3 \\
&= 1/4(2^{2(n-1)} - 2\binom{2(n-1)}{n-1}) + K_1 + K_2 + S_3 \\
&= 1/4(2^{2(n-1)} - 2\binom{2(n-1)}{n-1}) \\
&\quad +(n-1)(2^{n-2} - \binom{n-1}{(n/2)-1}) - n + 1 \\
&\quad +(n-1)(2^{n-2} + \binom{n-1}{(n/2)-1} - 1) \\
&\quad +1/4(\binom{2(n-1)}{n-1} - 2^{n-1}) + 1/2n(n-1) \\
&= 2^{2n-4)} - 1/4\binom{2(n-1)}{n-1}) \\
&\quad +n2^{n-1} - 5 * 2^{n-3} + 1/2(n^2 - 5n + 4)
\end{aligned}$$

B.6.2 Case II: n odd

Let us start with the first sum

$$\begin{aligned}
S_1 &= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=2}^{k-1} \#\texttt{csg}(n,k)\#\texttt{csg}(n,i) + \sum_{k=2}^{(n+1)/2-1} \#\texttt{csg}(n,k)\#\texttt{csg}(n,1) \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=2}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + \sum_{k=2}^{(n+1)/2-1} \binom{n-1}{k-1} n \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + n \sum_{k=2}^{(n+1)/2-1} \binom{n-1}{k-1} - \sum_{k=2}^{(n+1)/2-1} \binom{n-1}{k-1} \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1) \sum_{k=2}^{(n+1)/2-1} \binom{n-1}{k-1} \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1) \sum_{k=1}^{(n+1)/2-2} \binom{n-1}{k} \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1) \sum_{k=1}^{(n-1)/2-1} \binom{n-1}{k} \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1) \sum_{k=1}^{(n-1)/2-1} \binom{n-1}{k} \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1)(\sum_{k=0}^{(n-1)/2} \binom{n-1}{k} - \binom{n-1}{(n-1)/2} - 1) \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1)(2^{n-2} + 1/2\binom{n-1}{(n-1)/2} - \binom{n-1}{(n-1)/2} - 1) \\
&= \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1)(2^{n-2} - 1/2\binom{n-1}{(n-1)/2} - 1)
\end{aligned}$$

Define

$$C_1 = \sum_{k=1}^{(n+1)/2-1} \sum_{i=1}^{k-1} \binom{n-1}{k-1}\binom{n-1}{i-1} \tag{42}$$

$$K_1 = (n-1)(2^{n-2} - 1/2\binom{n-1}{(n-1)/2} - 1) \tag{43}$$

For the second sum we have

$$\begin{aligned}
S_2 &= \sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} \#\mathtt{csg}(n,k)\#\mathtt{csg}(n,i) \\
&= \sum_{k=(n+1)/2}^{n-1} \sum_{i=2}^{n-k} \#\mathtt{csg}(n,k)\#\mathtt{csg}(n,i) + \sum_{k=(n+1)/2}^{n-1} \#\mathtt{csg}(n,k)\#\mathtt{csg}(n,i) \\
&= \sum_{k=(n+1)/2}^{n-1} \sum_{i=2}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + \sum_{k=(n+1)/2}^{n-1} \binom{n-1}{k-1} n \\
&= \sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + n \sum_{k=(n+1)/2}^{n-1} \binom{n-1}{k-1} - \sum_{k=(n+1)/2}^{n-1} \binom{n-1}{k-1} \\
&= \sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1) \sum_{k=(n+1)/2}^{n-1} \binom{n-1}{k-1} \\
&= \sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1) \sum_{k=(n-1)/2}^{n-2} \binom{n-1}{k} \\
&= \sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} + (n-1)(2^{n-2} + 1/2\binom{n-1}{(n-1)/2} - 1)
\end{aligned}$$

Define

$$C_2 = \sum_{k=(n+1)/2}^{n-1} \sum_{i=1}^{n-k} \binom{n-1}{k-1}\binom{n-1}{i-1} \tag{44}$$

$$K_2 = (n-1)(2^{n-2} + 1/2\binom{n-1}{(n-1)/2} - 1) \tag{45}$$

The third sum can be calculated as follows:

$$\begin{aligned}
S_3 &= \sum_{k=1}^{(n-1)/2} \#\texttt{csg}(n,k)(\#\texttt{csg}(n,k)-1)/2 \\
&= \sum_{k=2}^{(n-1)/2} \#\texttt{csg}(n,k)(\#\texttt{csg}(n,k)-1)/2 + \#\texttt{csg}(n,1)(\#\texttt{csg}(n,1)-1)/2 \\
&= 1/2 \sum_{k=2}^{(n-1)/2} \binom{n-1}{k-1}(\binom{n-1}{k-1}-1) + n(n-1)/2 \\
&= 1/2 \sum_{k=1}^{(n-1)/2} \binom{n-1}{k-1}(\binom{n-1}{k-1}-1) + n(n-1)/2 \\
&= 1/2 \sum_{k=0}^{(n-1)/2-1} \binom{n-1}{k}(\binom{n-1}{k}-1) + n(n-1)/2 \\
&= 1/2 \sum_{k=1}^{(n-1)/2} \binom{n-1}{k}(\binom{n-1}{k}-1) - 1/2\binom{n-1}{(n-1)/2}(\binom{n-1}{(n-1)/2}-1) + n(n-1)/2 \\
&= 1/4(\binom{2(n-1)}{n-1} + \binom{n-1}{(n-1)/2}\binom{n-1}{(n-1)/2} - 2^{n-1} - \binom{n-1}{(n-1)/2}) \\
&\quad -1/2\binom{n-1}{(n-1)/2}(\binom{n-1}{(n-1)/2}-1) + n(n-1)/2
\end{aligned}$$

$A_M(n-1)/4$ exactly covers C_1 and C_2. Hence, we have

$$\begin{aligned}
I_{\text{SzCh}}(n) &= A_M(n-1)/4 + K_1 + K_2 + S_3 \\
&= 1/4(2^{2(n-1)} + \binom{n-1}{(n-1)/2}\binom{n-1}{(n-1)/2} - 2\binom{2(n-1)}{n-1}) \\
&\quad +(n-1)(2^{n-2} - 1/2\binom{n-1}{(n-1)/2} - 1) \\
&\quad +(n-1)(2^{n-2} + 1/2\binom{n-1}{(n-1)/2} - 1) \\
&\quad +1/4(\binom{2(n-1)}{n-1} + \binom{n-1}{(n-1)/2}\binom{n-1}{(n-1)/2} - 2^{n-1} - \binom{n-1}{(n-1)/2}) \\
&\quad -1/2\binom{n-1}{(n-1)/2}(\binom{n-1}{(n-1)/2}-1) + n(n-1)/2 \\
&= 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + 1/4\binom{n-1}{(n-1)/2} + n2^{n-1} - 5*2^{n-3} + 1/2(n^2-5n+4)
\end{aligned}$$

C DP-Variants Based on Subset Generation

The *Inner Counter* of the DP-Variant `Sub` is independent of the query graph and can be calculated as follows:

$$\begin{aligned} I_{\mathrm{Sub}} &= \sum_{k=1}^{n} \binom{n}{k} \sum_{i=1}^{k-1} \binom{k}{i} \\ &= 3^n - 2^{n+1} + 1 \end{aligned} \tag{46}$$

Subsequently, we investigate the variant `SubAlt`.

C.1 DP Variant SubAlt: General Remarks

Independent of the query graph, we can derive the following formula for calculating the value of the inner counter after termination of the DP-Variant `SubAlt`:

$$\begin{aligned} I_{\mathrm{SubAlt}}(n) &= \sum_{k=1}^{n} \#\mathtt{csg}(n,k) \sum_{i=1}^{k-1} \binom{k}{i} \\ &= \sum_{k=1}^{n} \#\mathtt{csg}(n,k) \sum_{i=0}^{k} \binom{k}{i} - 2 \sum_{k=1}^{n} \#\mathtt{csg}(n,k) \\ &= \sum_{k=1}^{n} 2^k \#\mathtt{csg}(n,k) - 2 \sum_{k=1}^{n} \#\mathtt{csg}(n,k) \end{aligned}$$

C.2 DP-Variant SubAlt: Chain

Remembering

$$\begin{aligned} \#\mathtt{csg}(n,k) &= (n-k+1) \\ \#\mathtt{csg}(n) &= n(n+1)/2 \end{aligned}$$

we easily derive

$$\begin{aligned} I_{\mathrm{SubAlt}}(n) &= \sum_{k=1}^{n} 2^k \#\mathtt{csg}(n,k) - 2 \sum_{k=1}^{n} \#\mathtt{csg}(n,k) \\ &= \sum_{k=1}^{n} 2^k (n-k+1) - 2(n(n+1)/2) \\ &= n \sum_{k=1}^{n} 2^k - \sum_{k=1}^{n} k2^k + \sum_{k=1}^{n} 2^k - n^n - n) \\ &= n(2^{n+1}-2) - ((n-1)2^{n+1}+2) + 2^{n+1} - 2 - n^n - n \\ &= 2^{n+2} - n^n - 3n - 4 \end{aligned}$$

where we applied Identity 68.

C.3 DP-Variant SubAlt: Cycle

Remembering

$$\#\texttt{csg}(n,k) = \begin{cases} 1 & n = k \\ n & \text{else} \end{cases}$$

and $\#\texttt{csg}(n) = n^2 - n + 1$ we easily derive

$$\begin{aligned} I_{\text{SubAlt}}(n) &= \sum_{k=1}^{n} 2^k \#\texttt{csg}(n,k) - 2\sum_{k=1}^{n} \#\texttt{csg}(n,k) \\ &= \sum_{k=1}^{n-1} 2^k n + 2^n - 2(n^2 - n + 1) \\ &= n(2^n - 2) + 2^n - 2n^2 + 2n - 2 \\ &= n2^n + 2^n - 2n^2 - 2 \end{aligned}$$

C.4 DP-Variant SubAlt: Star

Remembering

$$\#\texttt{csg}(n,k) = \begin{cases} n & k = 1 \\ \binom{n-1}{k-1} & k > 1 \end{cases}$$

and $\#\texttt{csg}(n) = 2^{n-1} + n - 1$, we easily derive

$$\begin{aligned} I_{\text{SubAlt}}(n) &= \sum_{k=1}^{n} 2^k \#\texttt{csg}(n,k) - 2\sum_{k=1}^{n} \#\texttt{csg}(n,k) \\ &= \sum_{k=1}^{n} 2^k \binom{n-1}{k-1} + 2n - 2 - 2(2^{n-1} + n - 1) \\ &= 2\sum_{k=1}^{n-1} 2^{k-1} \binom{n-1}{k-1} + 2n - 2 - 2^n - 2n + 2 \\ &= 23^{n-1} - 2^n \end{aligned}$$

C.5 DP-Variant SubAlt: Clique

For cliques, the additional test always succeeds and hence

$$\begin{aligned} I_{\text{SubAlt}}(n) &= \sum_{k=1}^{n} 2^k \#\texttt{csg}(n,k) - 2\sum_{k=1}^{n} \#\texttt{csg}(n,k) \\ &= 3^n - 2^{n+1} + 1 \end{aligned}$$

D Useful Identities

It is well known that (see [2]):

$$\binom{n}{k} = \binom{n}{n-k} \tag{47}$$

$$\sum_{k=0}^{n}\binom{m+k}{k} = \binom{m+n+1}{m+1} = \binom{m+n+1}{n} \tag{48}$$

Combining these two identities results in

$$\begin{aligned}\sum_{k=0}^{m}\binom{k+r}{r} &= \sum_{k=0}^{m}\binom{k+r}{k}\\ &= \binom{m+r+1}{r+1} \qquad (49)\end{aligned}$$

The following identity will also be used:

$$\sum_{k=0}^{n}2^k\binom{n}{k} = 3^n \tag{50}$$

This identity follows inductively from (case $n=0$: $\surd$)

$$\begin{aligned}\sum_{k=0}^{n+1}2^k\binom{n+1}{k} &= \sum_{k=1}^{n}2^k\binom{n+1}{k} + 2^{n+1}\binom{n+1}{n+1} + 2^0\binom{n+1}{0}\\ &= \sum_{k=1}^{n}2^k(\binom{n}{k} + \binom{n}{k-1}) + 2^{n+1}\binom{n+1}{n+1} + 2^0\binom{n+1}{0}\\ &= \sum_{k=1}^{n}2^k\binom{n}{k} + \sum_{k=1}^{n}2^k\binom{n}{k-1} + 2^{n+1}\binom{n+1}{n+1} + 2^0\binom{n+1}{0}\\ &= \sum_{k=1}^{n}2^k\binom{n}{k} + 2^0\binom{n+1}{0} + 2*\sum_{k=0}^{n-1}2^k\binom{n}{k} + 2*2^n\binom{n}{n}\\ &= \sum_{k=0}^{n}2^k\binom{n}{k} + 2*\sum_{k=0}^{n}2^k\binom{n}{k}\\ &= 3^n + 2*3^n\\ &= 3^{n+1}\end{aligned}$$

From Identity 49 we derive some more useful identities:

$$\begin{aligned}
\sum_{k=1}^{m} k &= \binom{m+1}{2} && (51)\\
\sum_{k=1}^{m} k(k+1) &= 2\binom{m+2}{3} && (52)\\
\sum_{k=1}^{m} k(k-1) &= 2\binom{m+1}{3} && (53)\\
\sum_{k=1}^{m} k^2 &= 2\binom{m+1}{3} + \binom{m+1}{2} && (54)\\
\sum_{k=1}^{m} k^2 &= 2\binom{m+2}{3} - \binom{m+1}{2} && (55)\\
\sum_{k=1}^{m} k(k+1)(k+2) &= 6\binom{m+3}{4} && (56)\\
\sum_{k=1}^{m} k(k+1)(k+1) &= 6\binom{m+3}{4} - 2\binom{m+2}{3} && (57)\\
\sum_{k=1}^{m} k(k-1)(k-2) &= 6\binom{m+1}{4} && (58)\\
\sum_{k=1}^{m} k(k-1)(k-1) &= 6\binom{m+1}{4} + 2\binom{m+1}{3} && (59)\\
\sum_{k=1}^{m} k^3 &= 6\binom{m+2}{4} + \binom{m+1}{2} && (60)
\end{aligned}$$

Identity 51:

$$\begin{aligned}
\sum_{k=1}^{m} k &= \sum_{k=1}^{m} \binom{k}{1}\\
&= \binom{m+1}{2}
\end{aligned}$$

Identity 52:

$$\begin{aligned}
\sum_{k=1}^{m} k(k+1) &= 2\sum_{k=1}^{m} \binom{k+1}{2}\\
&= 2\sum_{k=0}^{m-1} \binom{k+2}{2}\\
&= 2\binom{m+2}{3}
\end{aligned}$$

Identity 53:

$$\begin{aligned}\sum_{k=1}^{m} k(k-1) &= \sum_{k=0}^{m-1} k(k+1)\\ &= 2\binom{m+1}{3}\end{aligned}$$

Identity 54:

$$\begin{aligned}\sum_{k=1}^{m} k^2 &= \sum_{k=1}^{m} k(k-1+1)\\ &= \sum_{k=1}^{m} k(k-1) + \sum_{k=1}^{m} k\\ &= 2\binom{m+1}{3} + \binom{m+1}{2}\end{aligned}$$

Identity 55:

$$\begin{aligned}\sum_{k=1}^{m} k^2 &= \sum_{k=1}^{m} k(k+1-1)\\ &= \sum_{k=1}^{m} k(k-1) - \sum_{k=1}^{m} k\\ &= 2\binom{m+2}{3} - \binom{k+1}{2}\end{aligned}$$

Identity 56:

$$\begin{aligned}\sum_{k=1}^{m} k(k+1)(k+2) &= 6\sum_{k=1}^{m} \binom{k+2}{3}\\ &= 6\sum_{k=0}^{m-1} \binom{k+3}{3}\\ &= 6\binom{m+3}{4}\end{aligned}$$

Identity 57:

$$\begin{aligned}\sum_{k=1}^{m} k(k+1)(k+1) &= \sum_{k=1}^{m} k(k+1)(k+2) - \sum_{k=1}^{m} k(k+1)\\ &= 6\binom{m+3}{4} - 2\binom{m+2}{3}\end{aligned}$$

Identity 58:

$$\begin{aligned}\sum_{k=1}^{m} k(k-1)(k-2) &= \sum_{k=0}^{m-3} k(k+1)(k+2)\\ &= 6\sum_{k=0}^{m-3}\binom{k+3}{3}\\ &= 6\binom{m+1}{4}\end{aligned}$$

Identity 59:

$$\begin{aligned}\sum_{k=1}^{m} k(k-1)(k-1) &= \sum_{k=1}^{m} k(k-1)(k-2+1)\\ &= \sum_{k=1}^{m} k(k-1)(k-2)+\sum_{k=1}^{m} k(k-1)\\ &= 6\binom{m+1}{4}+2\binom{m+1}{3}\end{aligned}$$

Identity 60:

$$\begin{aligned}\sum_{k=1}^{m} k^3 &= \sum_{k=2}^{m+1}(k-1)(k-1)(k-1)\\ &= \sum_{k=2}^{m+1} k(k-1)(k-1)-\sum_{k=2}^{m+1}(k-1)(k-1)\\ &= \sum_{k=1}^{m+1} k(k-1)(k-1)-\sum_{k=1}^{m} k^2\\ &= 6\binom{m+2}{4}+2\binom{m+2}{3}-2\binom{m+1}{3}-\binom{m+1}{2}\\ &= 6\binom{m+2}{4}+\binom{m+2}{3}-2\binom{m+1}{3}+\binom{m+2}{3}-\binom{m+1}{2}\\ &= 6\binom{m+2}{4}+\binom{m+2}{3}-2\binom{m+1}{3}+\binom{m+1}{3}\\ &= 6\binom{m+2}{4}+\binom{m+2}{3}-\binom{m+1}{3}\\ &= 6\binom{m+2}{4}+\binom{m+1}{2}\end{aligned}$$

where we used

$$\binom{n}{k} = \binom{n-1}{k}+\binom{n-1}{k-1} \tag{61}$$

$$\binom{n}{k}-\binom{n-1}{k-1} = \binom{n-1}{k} \tag{62}$$

$$\binom{n}{k}-\binom{n-1}{k} = \binom{n-1}{k-1} \tag{63}$$

From Vandermonde's Convolution we can derive

$$\sum_0^n \binom{n}{k}\binom{n}{k} = \binom{2n}{n} \tag{64}$$

[Note that this is equal to $\frac{1}{n+1}\mathcal{C}(n)$ where $\mathcal{C}(n)$ are the catalan numbers.]

Further, it is helpful to know that

$$\sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{k} = \begin{cases} 2^{n-1} + 1/2\binom{n}{n/2} & n \mod 2 \equiv 0 \\ 2^{n-1} & n \mod 2 \equiv 1 \end{cases} \tag{65}$$

and

$$\sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{k}\binom{n}{k} = \begin{cases} 1/2\binom{2n}{n} + 1/2\binom{n}{n/2}\binom{n}{n/2} & n \mod 2 \equiv 0 \\ 1/2\binom{2n}{n} & n \mod 2 \equiv 1 \end{cases} \tag{66}$$

For a double sum of the product of binomial coefficients, we have

$$\sum_{k=0}^{\lfloor n/2 \rfloor} \sum_{j=0}^{\lfloor n/2 \rfloor} \binom{n}{k}\binom{n}{j} = \begin{cases} (2^{n-1} + 1/2\binom{n}{n/2})^2 & n \mod 2 \equiv 0 \\ (2^{n-1})^2 & n \mod 2 \equiv 1 \end{cases} \tag{67}$$

A last identity used is

$$\sum_{k=1}^{n} k2^k = (n-1)2^{n+1} + 2 \tag{68}$$

This identity follows directly by induction ($n = 1$: $\surd$) from:

$$\begin{aligned} \sum_{k=1}^{n+1} k2^k &= \sum_{k=1}^{n} k2^k + (n+1)2^{n+1} \\ &= (n-1)2^{n+1} + 2 + (n+1)2^{n+1} \\ &= n2^{n+2} + 2 \end{aligned}$$

References

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

[2] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2002.

[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. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, 1988.

[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] K. Sutner, A. Satyanarayana, and C. Suffel. The complexity of the residual node connectedness reliability problem. *SIAM J. Comp.*, 20(1):149–155, 1991.

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

[11] 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.