Minimal Dominating Sets in a Tree: Counting, Enumeration, and Extremal Results

GÜNTER ROTE, Freie Universität Berlin, Germany

A tree with n vertices has at most $95^{n/13}$ minimal dominating sets. The growth constant $\lambda = \sqrt[13]{95} \approx 1.4194908$ is best possible. It is obtained in a semi-automatic way as a kind of "dominant eigenvalue" of a bilinear operation on sextuples that is derived from the dynamic-programming recursion for computing the number of minimal dominating sets of a tree. This technique is generalizable to other counting problems, and it raises questions about the "growth" of a general bilinear operation. We also derive an output-sensitive algorithm for listing all minimal dominating sets with linear set-up time and linear delay between successive solutions.

Contents

Author's address: Günter Rote, Freie Universität Berlin, Institut für Informatik, Takustraße 9, 14195 Berlin, Germany, rote@inf.fu-berlin.de.

1 INTRODUCTION

Problem Statement. A vertex a in an undirected graph $G = (V, E)$ *dominates* a vertex b if $b = a$ or b is adjacent to a. A *dominating set* is a subset $D \subseteq V$ such that every vertex is dominated by some element of D. In other words, every vertex $a \in V - D$ must have a neighbor in D. D is a *minimal dominating set* if no proper subset of D is a dominating set.

Results. Let M_n denote the maximum number of minimal dominating sets that a tree with n vertices can have. We provide the correct and tight value of the growth constant λ of M_n.

THEOREM 1.1. *Let* $\lambda = \sqrt[13]{95} \approx 1.4194908$.

1) *A tree with n vertices has at most* $2\lambda^{n-2} < 0.992579 \cdot \lambda^n$ *minimal dominating sets.*
2) *For every n, there is a tree with at least* $0.649748 \cdot \lambda^n$ *minimal dominating sets.*
3) *For every n of the form* $n = 13k + 1$, *there is a tree with at least* $95^k > 0.704477 \cdot \lambda^n$ *minimal dominating sets.*

On the algorithmic side, we derive an output-sensitive algorithm for enumerating all solutions:

THEOREM 1.2. *The minimal dominating sets of a tree with n vertices can be enumerated with* $O(n)$ *setup time and with* $O(n)$ *delay between successive solutions.*

Previous Results. Marcin Krzywkowski [2013] gave an algorithm for listing all minimal dominating sets of a tree of order n in time $O(1.4656^n)$, thus proving that every tree has at most 1.4656^n minimal dominating sets. Golovach, Heggernes, Kanté, Kratsch and Villanger [2017] recently improved this upper bound to $3^{n/3} \approx 1.4422^n$.

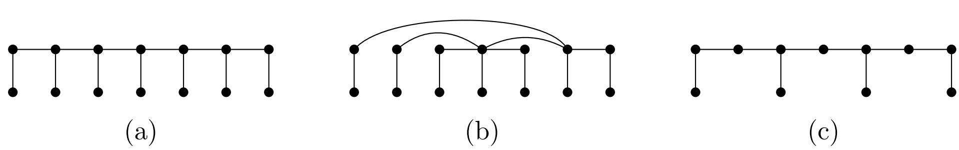


Fig. 1. (a) the comb graph with 7 teeth, (b) a generalized comb, (c) an extended comb (used in Section 5.4).

Small examples indicate that the class of *comb graphs* of Figure 1a with an even number n of vertices and $n/2$ teeth might have the largest number of minimal dominating sets. They have $2^{n/2} \approx 1.4142^n$ minimal dominating sets, because one can independently choose a vertex out of every tooth (see Observation 1(1) below). The class of graphs with so many minimal dominating sets is in fact very large: One can take *any* tree on $n/2$ vertices and append a leaf to each vertex, as in Figure 1b. The trees with odd n seem to have much fewer than 1.4142^n minimal dominating sets. It turns out that these observations are indeed true for $n \leq 18$, but they fail for larger n, see Figure 15 and Table 3 in Section 6.3.

The best lower bound on the growth constant λ that has been known so far is $\sqrt[27]{12161} \approx 1.416756$, due to Krzywkowski [2013]. Krzywkowski constructed a tree with 27 vertices and 12161 minimal dominating sets. Since the sequence M_n is supermultiplicative (Observation 1(4) below), this establishes $\sqrt[27]{12161}$ as a lower bound on λ.

It occurs frequently in combinatorics that a lower bound is established through a particular example, from which the asymptotic growth is derived with the help of supermultiplicativity. However, in our case, this method is bound to fail in finding the true lower bound: By Part 1 of Theorem 1.1, a tree with n vertices that would have λ^n minimal dominating sets does not exist. By contrast, our lower bound $\lim \sqrt[n]{M_n} \geq \lambda$ will be established by an infinite family of trees (Section 3).

The question can of course be asked for other graph classes than trees, and there is an extensive literature, see [Couturier et al. 2013] for an overview. On general graphs, the best upper bound is 1.7159^n, and no graph with n vertices and more than 1.5705^n minimal dominating sets is known.

Techniques. While we settle the question of the growth constant for trees, we believe that the techniques that have lead to this result are more interesting than the result itself.

We start with a standard dynamic-programming algorithm for counting the number of minimal dominating sets of a *particular* tree (Section 4). The algorithm operates on sextuples of numbers, because there are six classes of partial solutions that must be distinguished. We then abstract the calculation from a particular tree, and deduce an algorithm for finding all sextuples that can arise for a fixed number n of vertices. From this, it is easy to calculate M_n.

Finally, we will try to enclose the set of sextuples in a six-dimensional geometric body. If we succeed to find an appropriate shape with certain properties, which depend on some putative value of λ, we have established λ as an upper bound of the growth constant (Proposition 6.3 in Section 6.4). This suggests a semi-automatic computer-assisted method for searching for the correct growth constant (Section 6.5).

As a side result, our dynamic-programming setup can be adapted to an efficient enumeration algorithm for listing all minimal dominating sets of a tree (Theorem 1.2) with linear delay, see Section 5. Previous algorithms [Krzywkowski 2013; Golovach et al. 2017] were not even output-sensitive in the sense of being polynomial in the combined size of the input and output.

These results were presented in preliminary form at the ACM–SIAM Symposium on Discrete Algorithms (SODA19) in San Diego in January 2019 [Rote 2019a].

2 PRELIMINARIES

A more concrete characterization of minimal dominating sets is a follows. A dominating set D is a minimal dominating set if and only if every vertex $a \in D$ has a *private neighbor*: a vertex b that dominated by a but by no other vertex in D. (The private "neighbor" can be the vertex a itself.)

It is useful to rephrase these conditions: We call a vertex $a \in V$ *legal* if

(a) $a \in D$ and a has a private neighbor, or
(b) $a \notin D$ and a is dominated, i.e., it has some neighbor in D.

Thus, D is a minimal dominating set if and only if all vertices of the graph are legal.

We will now establish some basic facts about minimal dominating sets, culminating in the well-known fact that the numbers M_n are supermultiplicative.

Observation 1. *1) If a is leaf and b its neighbor, then every minimal dominating set D contains exactly one of a and b. Moreover, a can always be chosen as the private neighbor of this vertex.*
2) If $a_1, \ldots, a_k$ are leaves with a common neighbor b, then either all vertices $a_1, \ldots, a_k$ belong to D or none of them belongs to D. (We will call two leaves that have a common neighbor twins.*)*
3) If T_1 and T_2 are two trees with $M(T_1)$ and $M(T_2)$ minimal dominating sets, there is a way to insert an edge between T_1 and T_2 such that the resulting tree has exactly $M(T_1)M(T_2)$ minimal dominating sets, except when T_1 and T_2 are two singleton trees.
4) The function M_n is supermultiplicative:

$$M_{i+j} \geq M_i M_j$$

for $i, j \geq 1$.

Proof. Statement 1 is easy to see, and Statement 2 follows directly from it.

For the third claim, consider first the case that both T_1 and T_2 have at least 2 vertices. Let a_i be a leaf in T_i and b_i be its neighbor. Then we connect the trees by the edge $b_1 b_2$. We argue that the presence of this edge makes no difference for the minimal dominating sets in the union of the two trees. An edge b_1b_2 could in principle affect the legality of b_1 or b_2 or a neighbor of b_1 or b_2. However, (i) b_1 is always dominated either by a_1 or by b_1, no matter whether the edge b_1b_2 is present. (ii) Independently of whether we choose a_1 or b_1 as an element of D or not, we can always choose a_1 as a private neighbor for it; the edge b_1b_2 is not required to find a private neighbor. (iii) b_1 can never be used as a private neighbor of another vertex than a_1 or b_1 because it is already dominated by a or b. Thus the presence or removal of b_1b_2 will neither help nor prevent any vertex to find a private neighbor.

When one of the trees, say T_1, is a singleton tree, we connect it to a neighbor b_2 of a leaf a_2 in T_2. In the resulting tree, a_2 has a new twin, and thus $M(T_2)$ is unchanged. In view of $M(T_1) = 1$, this is what we need.

Supermultiplicativity in the fourth claim follows from Statement 3. The exceptional case $i = j = 1$, when T_1 and T_2 are two singleton trees, can be checked directly. □

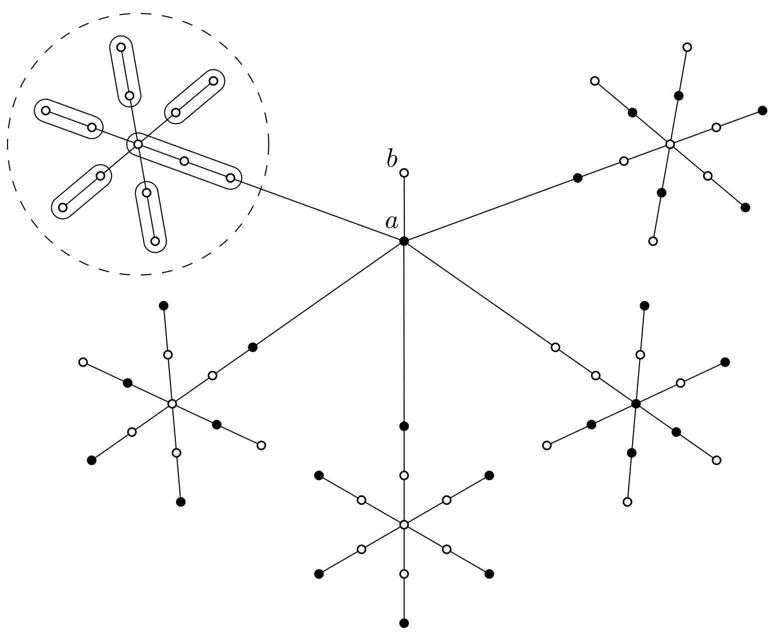


Fig. 2. A star of 5 snowflakes. The vertices of D are black.

3 LOWER BOUND EXAMPLE: THE STAR OF SNOWFLAKES

The lower bound on the constant λ is proved by the *star of snowflakes* (Figure 2), a family of examples with $13k + 2$ vertices and at least 95^k minimal dominating sets, for $k \geq 1$. Through the analysis of this example, we hope that the reader may get familiar with minimal dominating sets. A single snowflake has 13 vertices and consists of 6 paths of two edges each, attached to a central vertex. We take the union of k snowflakes and a separate *root vertex a*, and we connect a to a leaf of each snowflake. In addition, a gets another leaf b as a neighbor, for a total of $13k + 2$ vertices. Let us count the minimal dominating sets containing a. We will first check that 95 possibilities can be independently chosen in each snowflake: We partition each snowflake into five groups of size 2 and one group of size 3, as shown in the snowflake at the top left of Figure 2. It is now straightforward to check that a minimal dominating set must contain exactly one vertex from each group. (For the five groups of size 2, this follows directly from Observation 1(1).) Out of these $3 \cdot 2^5 = 96$ possibilities, one possibility is forbidden, namely the choice of all six outermost vertices (shown in the bottom snowflake of the figure), because this would leave the central vertex undominated. The other 95 possibilities lead to valid minimal dominating sets. Thus the star of k snowflakes has at least 95^k minimal dominating sets, as claimed, and the growth constant λ cannot be smaller than $\lim_{k\to\infty}(95^k)^{1/(13k+2)} = \sqrt[13]{95}$. We have ignored the minimal dominating sets that don't contain a, but their number is negligible: it is 64^k.

A tree that approaches the upper bound more tightly is obtained by omitting the vertex b, but it is not so straightforward to analyze. Such a tree has $13k+1$ vertices and $95^k - 63^k + 64^k + k \cdot 32^{k-1} \geq 95^k$ minimal dominating sets. Let us at least confirm the leading term: The 95^k sets are the same ones as before. If we subtract the 63^k cases where *every* star has a neighbor or a distance-2 neighbor of a in D, we are sure that the vertex $a \in D$ can choose a private neighbor. This establishes the lower bound $95^k - 63^k = 95^k(1 - o(1))$ on the asymptotic growth for these trees. The last two terms of the formula are for the cases where $a \in D$ chooses itself as a private neighbor or a does not belong to D.

This family of trees gives asymptotically the largest number of minimal dominating sets that we know. It approaches the bound λ^n with a multiplicative error that goes to $1/\lambda \approx 0.704$ as $k \to \infty$, and this proves part 3 of Theorem 1.1. We call these trees our *record trees* and denote them by RT_{13k+1}.

We remark that, in the original star of snowflakes, the 95^k minimal dominating sets containing the vertex a are in fact *minimum* dominating sets: dominating sets of smallest size. Since they are always a subset of the minimal dominating sets, the asymptotic growth constant λ is valid also for *minimum dominating sets* in trees.

4 COUNTING MINIMAL DOMINATING SETS OF A PARTICULAR TREE BY DYNAMIC PROGRAMMING

4.1 Combining rooted trees

It is not difficult to compute the number of minimal dominating sets of a tree by dynamic programming, and there are different ways to organize the computation. For inductively building up a tree from smaller trees, it is convenient to mark an arbitrary vertex as the *root* of the tree. We combine trees with the following *composition* operation: We take two rooted trees A and B and add an edge between the roots. The root of A is kept as the root of the result. The basic building block for the construction is the singleton tree. There are many ways in which a given tree T can be built up through a sequence of compositions: After selecting an arbitrary root vertex r for T, one picks an edge rs incident to r and removes it. This results in two trees with roots r and s, and these two trees are further decomposed recursively. In the following, we will specify a subtree by its vertex set $A \subseteq V$, often without explicitly mentioning its root.

We want count minimal dominating sets bottom-up, following the composition. In this process, we have to count *partial solutions*, i.e., subsets $D \subseteq A$ that have the potential to become a minimal dominating set when more components are connected to the root r. In Section 2 we have characterized minimal dominating sets by requiring that every vertex is *legal*. The subtree A is connected to the rest of the tree by edges incident to r; therefore, r itself need not be legal in a partial solution. Every vertex $a \neq r$, however, must be legal: It is dominated, and if it belongs to D, then it has a private neighbor.

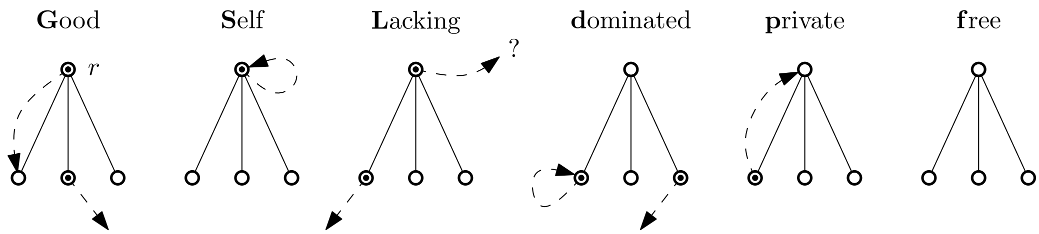
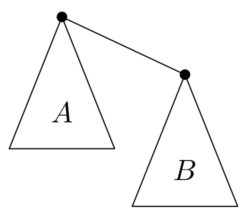


Fig. 3. Six types of partial solutions for a rooted tree. We show the root r and its neighbors in some typical configuration. The vertices belonging to D are marked. The dotted arrow indicates the private neighbor for a vertex.



		B					
		G	**S**	**L**	**d**	**p**	**f**
A	**G**	**G**	–	–	**G**	–	**G**
	S	**L**	–	–	**S**	–	**G**
	L	**L**	–	–	**L**	–	**G**
	d	**d**	**d**	–	**d**	**d**	–
	p	–	–	–	**p**	**p**	–
	f	**d**	**d**	**p**	**f**	**f**	–

Table 1. The category when a tree of type B is attached as a child to a tree of type A. The symbol "–" indicates that the result is not valid.

4.2 Combining partial solutions

By sitting down and thinking how to compose partial solutions, one will discover that six types of partial solutions must be distinguished, see Figure 3: When the root belongs to D, there are three categories, which we denote with capital letters:

- **G**ood. The root r has a private neighbor among its neighbors.
- **S**elf. The only private neighbor of the root r is r itself.
- **L**acking. The root r does not yet have a private neighbor. The private neighbor needs to be found among the neighbors that will still be attached to r.

When the root is not part of D, there are three more categories, indicated by small letters:

- **d**ominated. The root r is dominated by some neighbor in D, and each vertex in D has a private neighbor different from r.
- **p**rivate. There is vertex in D whose only private neighbor is the root.
- **f**ree. The root has no neighbor in D. A neighbor that will dominate r needs to be found in the components that will still be attached to r.

Table 1 shows the resulting category of a composite tree depending on the category of the components. Let us give an example: When composing a partial solution of type **L** for a tree A with root r and a partial solution of type **f** for a tree B, the root s of B can be used as the private neighbor for r, and at the same time, s has found a dominating vertex, namely r. The result will be of type **G**. Some compositions are not valid: For example, when B is of type **p**, the root s of B is the only private neighbor of some vertex below it. When this is combined with a tree A of type **G**, **S**, or **L**, s can no longer function as a private neighbor, because it is adjacent to the root of A, which belongs to D. The other entries of the table can be worked out similarly.

4.3 Characteristic vectors

For a rooted tree, we record the number of partial solutions of each type in a 6-vector $v = (G, S, L, d, p, f)$. Table 1 can be directly translated into the formula for the vector obtained by combining two subtrees T_1 and T_2 (written as column vectors):

$$\begin{pmatrix} G_1 \\ S_1 \\ L_1 \\ d_1 \\ p_1 \\ f_1 \end{pmatrix} \star \begin{pmatrix} G_2 \\ S_2 \\ L_2 \\ d_2 \\ p_2 \\ f_2 \end{pmatrix} := \begin{pmatrix} G_1G_2 + G_1d_2 + G_1f_2 + S_1f_2 + L_1f_2 \\ S_1d_2 \\ S_1G_2 + L_1G_2 + L_1d_2 \\ d_1G_2 + d_1S_2 + d_1d_2 + d_1p_2 + f_1G_2 + f_1S_2 \\ p_1d_2 + p_1p_2 + f_1L_2 \\ f_1d_2 + f_1p_2 \end{pmatrix} \tag{1}$$

The *final categories* are those partial solutions that can stand alone as a minimal dominating set: **G**, **S**, **d**, and **p**. Therefore, the total number $M(T)$ of minimal dominating sets of a tree T with vector (G, S, L, d, p, f) is calculated by the linear function

$$\bar{M}(G, S, L, d, p, f) := G + S + d + p. \tag{2}$$

A single-vertex tree has category **S** when the vertex belongs to D, and category **f** if $D = \emptyset$. Thus, a single-vertex tree has the vector

$$v_0 := (0, 1, 0, 0, 0, 1). \tag{3}$$

This provides the starting condition for the recursion.

We have now all ingredients for a straightforward counting algorithm for the minimal dominating sets of a tree: choose a root, recursively decompose the tree into smaller parts, compute the vectors for all parts in a bottom-up way, and apply the operation $\bar{M}$ from (2) to the result vector. Figure 4 shows a partially worked example.

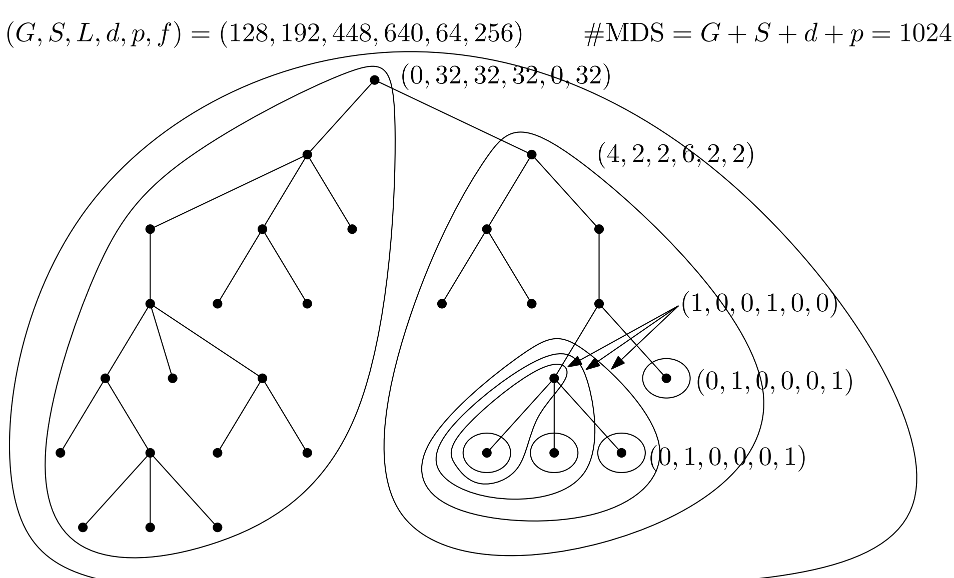


Fig. 4. Calculating the number of minimal dominating sets of a tree bottom-up

All the knowledge about the possible number of minimal dominating sets that a tree with n vertices can have is actually embodied in these formulas: the starting vector (3), the composition operation (1) in terms of the bilinear operation $\star$, and the terminal formula (2).

Before we embark on studying these formulas from a quantitative viewpoint, we will use them for designing an enumeration algorithm.

5 LISTING ALL MINIMAL DOMINATING SETS OF A TREE

In the previous section, the composition rules in Table 1 have been used to design a dynamic-programming algorithm for *counting* minimal dominating sets, based on the recursion (1) for the *number* of partial solutions of each category. We can reinterpret (1) as an implicit representation of the *set* of partial solutions. For instance, Table 1 tells us that each solution of category **S** for a subtree A and each solution of category **G** for B, when taken together, give rise to a solution of category **L** for the combined tree. Accordingly, we find the term S_1G_2 in (1), but we now interpret the multiplication as a sort of Cartesian product operation, combining all solutions of one set with all solutions from another set. The + operation is interpreted as set union.

We will first model the dynamic-programming recursion as a directed acyclic graph. Based on this implicit representation of the solutions, we will then develop an output-sensitive algorithm for listing all solutions.

5.1 The expression DAG

The directed acyclic graph (DAG) for representing all solutions in a tree T has three kinds of nodes: *basis nodes*, *product nodes*, and *union nodes*. Each node K is *associated* to some subtree A of G and it implicitly *represents* a some class $R(K) \subseteq 2^A$ of vertex subsets of A, namely the partial solutions of a certain category.

A *basis node* K has no outgoing arcs, and it is associated to a singleton subtree $A = \{a\}$. Its role is to declare that the vertex a is in D or does not belong to D. Accordingly, it represents the set $D = A = \{a\}$ itself ($R(K) = \{\{a\}\}$) or the empty set ($R(K) = \{\emptyset\}$). For uniformity, we also allow a basis node to represent no set ($R(K) = \{\}$), but we will eventually get rid of such nodes.

A *product node* K has two outgoing arcs to neighbors K_1 and K_2 that are associated to disjoint subtrees A_1 and A_2. The product node is then associated to $A_1 \cup A_2$, and it represents the vertex subsets obtained by combining each subset of A_1 represented by K_1 with each subset of A_2 represented by K_2:

$$R(K) = \{\, D_1 \cup D_2 \mid D_1 \in R(K_1), D_2 \in R(K_2)\,\}$$

A *union node* K has two outgoing arcs to neighbors K_1, K_2 that are associated to the same subtree A. The union node is then also associated to A, and it represents the *disjoint union* of its successor nodes:

$$R(K) = R(K_1) \cup R(K_2)$$

One node of the DAG is designated as the *target node* that represents the final solution set. It has no incoming arcs, and it is associated to the vertex set V of the whole tree. We draw the arcs from top to bottom, with the target node topmost and the basis nodes at the bottom.

With these types of nodes, it is straightforward to build an *expression DAG* $\mathcal{X}$ that represents the minimal dominating sets of a tree T. $\mathcal{X}$ has a node for each subtree that occurs in the composition sequence and for each category. Additional nodes are necessary for intermediate results when forming multiple unions. Figure 5 illustrates the construction with an example of the node $(C, \mathbf{L})$ for a rooted subtree C that is composed of two subtrees A and B. This node represents all partial solution of category **L** in the subtree C.

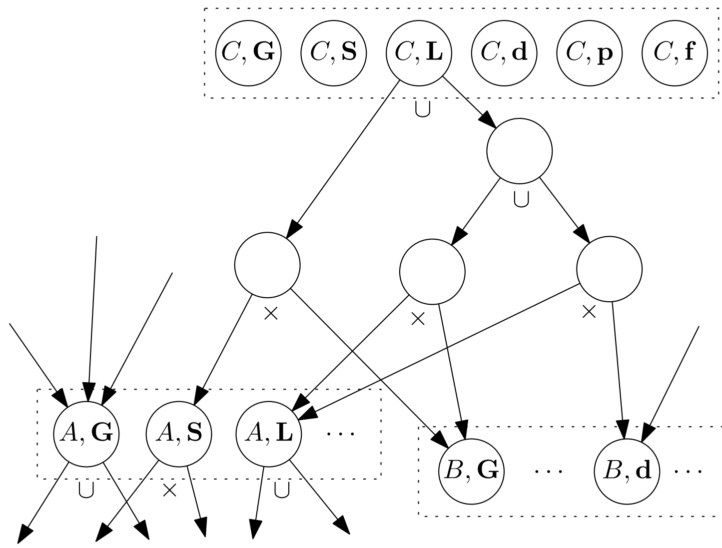


Fig. 5. A part of the DAG $\mathcal{X}$ corresponding to the third entry $S_1G_2 + L_1G_2 + L_1d_2$ in (1). Union and product nodes are marked by ∪ and ×.

The whole construction has $6n + 34(n-1) + 3$ nodes. 6 nodes are used to represent each singleton tree: One node represents the singleton set $\{\{a\}\}$, of category **S**, another one represents the empty set $\{\emptyset\}$, of category **f**, and the four others represent no set. There are $n - 1$ composition steps, one for each edge of T, and for each composition we need 34 nodes: $34 = 14 + 20$ is the number of additions and multiplications on the right-hand side of (1). Finally, we need 3 union nodes to compute the union of the categories **G**, **S**, **d**, and **p** for the whole tree, corresponding to the total sum $\bar{M} = G + S + d + p$. It is important to note that all union nodes in this construction represent disjoint unions, as every partial solution belongs to a unique category. Another important property of the tree is that a path can go through at most 8 consecutive union nodes: The largest number of additions for a single entry of (1) if 5; we have to add 3 for evaluating $\bar{M}$. The bound of 8 can be reduced to 4 if we care to balance the network of union nodes.

We can reinterpret $\mathcal{X}$ as an arithmetic circuit, by viewing union and product nodes as addition and multiplication gates, and basis nodes as inputs with values 0 or 1. Then the value computed in each node equals the number of subsets represented by that node, and the computation modeled by this circuit is nothing but our counting algorithm of Section 4.

5.2 Pruning of nodes

We now get rid of unnecessary nodes. In a first sweep we proceed upward from the basis nodes towards the target and eliminate all nodes representing the empty set. (They correspond to the gates that have value 0.) These are first of all the basis nodes of categories **G**, **L**, **d**, and **p**. Continuing towards the target node, we eliminate all union nodes without successor, and all product nodes that have lost at least one successor.

In a second, downward, sweep from the target towards basis nodes, we delete all nodes that do not contribute towards the result. These are all nodes without predecessor, except for the target node. In particular, intermediate results that would only be multiplied by 0 are discarded.

In a final clean-up step, we eliminate each union node K with a single successor K' and introduce shortcut arcs from the predecessors of K to K'.

Every node of the resulting DAG is now "useful": it represents a nonempty set, and it is computed through a nontrivial operation from its children. When the DAG is viewed as an arithmetic circuit, it starts with ones and performs multiplications and additions of positive numbers that will eventually contribute to the total number of minimal dominating sets. Thus, we need not worry about computing with excessively big numbers while the eventual result is small. For any tree T of size n we can evaluate the number $M(T)$ with $O(n)$ additions and multiplications of numbers that are bounded by $M(T)$, with $O(n)$ overhead. It is likely that even a straightforward application of the composition rules (1) without pruning never involves numbers that substantially exceed $M(T)$, but we have not tried to show this. In any case, the numbers are trivially bounded by 2^n, and thus, n bits are sufficient.

5.3 The enumeration algorithm ENUM1

The idea of the algorithm is clear: to enumerate the solutions represented by a union node, we have to enumerate solutions for the two successor nodes in sequence. For product nodes, the results of the successor nodes must be combined in all possible ways, by cycling through them in two nested loops. The real "work" is done only in the basis nodes: deciding whether a particular node belongs to the minimal dominating set D or not. We arbitrarily order the two successors of union and product nodes, so that we can speak of the first and second *child*. (We use the term "child" although $\mathcal{X}$ is not a tree.)

The program is easiest to write in a language like Python that supports generator functions, see Figure 6. Each node of $\mathcal{X}$ is represented by a Python object. The different node types are subclasses

```
class Basis_node_S(Node):
   def enumerate_solutions(self):
      a = self.vertex
      yield [a]   # category S
class Basis_node_f(Node):
   def enumerate_solutions(self):
      yield []    # category f, the only solution is the empty list
class Union_node(Node):
   def enumerate_solutions(self):
      for D in self.child1.enumerate_solutions():
         yield D
      for D in self.child2.enumerate_solutions():
         yield D
class Product_node(Node):
   def enumerate_solutions(self):
      for D1 in self.child1.enumerate_solutions():
         for D2 in self.child2.enumerate_solutions():
            yield D1+D2 # concatenation of lists D1 and D2
# main call:
for D in target_node.enumerate_solutions():
   print D # or otherwise process D
```

Fig. 6. Recursive enumeration algorithm in Python

of a common superclass `Node` whose definition is not shown. What is also omitted is the code to generate the graph and to set the `vertex` or the `child1` and `child2` attributes of the nodes.

The `yield` statement of Python suspends the execution of the current function until the next generated element is requested in the `for`-loop in which the function is called. Different generator functions and different nested loops are simultaneously active, and they interact like coroutines. The first parameter `self` of the functions is just Python's convention to refer to the object to which a method is attached.

The Python library actually provides standard functions for achieving precisely the effect of the enumeration procedures in the union and product nodes: the functions `itertools.chain` and `itertools.product` from the `itertools` package. For clarity, we wrote the loops explicitly instead of using these functions.

As currently written in Figure 6, the generation takes more than linear time per solution, because each solution is built up by concatenating shorter lists `D1` and `D2` into longer lists `D1+D2`, which is not a constant-time operation in Python. This has been done to make the program clear, but it is easy to fix: We can either use linked lists, or we just let each basis node set or clear a bit in a bit-vector representation of the solution. In the last variant, the program for a basis node of category **S** would be as follows:

```
i = self.vertex_number
D[i] = True  # category S
yield None
```

and accordingly with `False` for category **f**. The solution is maintained in the global variable `D`, which is a list of Boolean values. No partial solutions are ever returned to the calling subroutine, and the combination of the solutions can be bypassed. All `yield` statements of the program are changed so that they just produce the dummy element `None`. We will refer to this version as algorithm ENUM1. If desired, the solution can be constructed in any suitable form at the target node from the bit vector `D` in linear time.

The enumeration works as follows: When a new solution is needed, a call `enumerate_solutions` is initiated at the target node and proceeds towards the basis nodes. For a union node, one child is entered, and for a product node, the algorithm enters both children or only the second child, in case we are in the inner loop and the solution `D1` of the first child remains fixed. Eventually, at most one basis node is entered for each vertex, and there it is decided whether this vertex belongs to the solution D or not. The visited nodes form a subtree of $\mathcal{X}$ with at most n leaves. As we have observed, there can be at most 8 consecutive levels of union nodes where the tree does not branch. From this, one can conclude that the subtree of visited nodes has linear size.

However, the way how generators are handled in Python makes this argument invalid: When a loop like

for x in ⟨*generator-function*⟩: ...

loops over k successive elements x, the *generator-function* is actually called $k+1$ times. In the $(k+1)$-st iteration, it will raise the `StopIteration` exception to signal that there are no more items. Thus, in a union node, for example, the algorithm does not always descend into just *one* of the two children in the clean way as we supposed in our description. It might call `self.child1.enumerate_solutions()`, only to receive a `StopIteration` exception and subsequently call `self.child2.enumerate_solutions()`.

Despite this behavior, the runtime between successive solutions is still only $O(n)$. This fact requires a more elaborate analysis, which we will give in Section 5.7. Here it will be important that the number k of elements generated by every generator function is positive, due to the preparatory pruning of the expression DAG. Before that, in Section 5.5, we will describe and analyze a different process, ENUM2, for which the above argument goes through in a clean way. The analysis of

ENUM1 in Section 5.7 builds on these results. In the next section, we will first discuss a possibility for optimizing the *total* generation time.

5.4 Optimizing the overall runtime by reordering the children

As we have argued, and as we will show in Section 5.7, the algorithm takes $O(n)$ time per solution. In a setting where we want to examine each solution explicitly, this is optimal and leaves no room for improvement, at least if the size of a typical solution D is not much smaller than n.

Algorithm ENUM1 does not treat the children of a product node equally: While the solutions for child 1 are only enumerated once, the solutions for child 2 are enumerated again and again as part of the inner loop. One may try to optimize the running time by choosing the best order. Potentially, one may even achieve sublinear average time per solution.

In fact, in most enumeration tasks, an explicit list that can be stored is not what is actually needed, but one wants to run through all solutions, for example with the objective to evaluate them and choose the best one. Often, such an evaluation can be maintained incrementally: It is cheaper to *update* the objective function of D when a vertex is inserted or deleted instead of computing it from scratch. In such a setting, if makes sense to strive for sublinear average time. Since the basic operation of our enumeration algorithm is the insertion or deletion of single elements, the runtime of Algorithm ENUM1 gives an appropriate model for such an application case.

Let us therefore analyze the runtime for some product node K. Assume that child i represents C_i solutions, and t_i is the average time per solution, i. e., it takes time $t_i C_i$ to enumerated all solutions. Then, up to constant factors, the total time for node K is

$$C_1 C_2 + C_1 t_1 + C_1 C_2 t_2.$$

Here, the first term $C_1 C_2$ accounts for the time spent internally in the enumeration procedure for node K (putting together the solutions, passing them to the parent node, etc.), without the recursive calls. For this analysis, the extra `StopIteration` call at the end of the loop does not hurt us, because it would only change $C_1 C_2$ to $C_1 C_2 + 1$, and thus it would increase the overall runtime at most by a constant factor.

The resulting average time per solution is

$$t = 1 + t_1/C_2 + t_2.$$

This has to be compared against $t' = 1 + t_1 + t_2/C_1$. The typical case is when the numbers C_i are large; then the term that is divided by C_i becomes negligible, and the optimal choice gives

$$t \approx 1 + \min\{t_1, t_2\}. \tag{4}$$

For a union node, we have total time of

$$C_1 + C_2 + C_1 t_1 + C_2 t_2 = C_1(t_1 + 1) + C_2(t_2 + 1).$$

Thus, a union node effectively adds a constant overhead to each solution. One can optimize the structure of a tree of union nodes into a Huffman tree. However, since the number of consecutive levels of union nodes is already bounded by 8, this will improve the runtime at most by a constant factor.

For a given expression DAG, it is straightforward to compute the required quantities bottom-up and to reorder the children appropriately. Moreover, a given tree T has many recursive decompositions into subtrees, and it might be interesting to choose a best one. Formula (4) suggests that the runtime should depend on the shortest path from the root to a leaf (basis node). More precisely, such a short path should exist from every product node that is reachable from the target node through a sequence of union nodes. On the other hand, a short path to a leaf indicates a small subtree, and for small subtrees, the assumption under which the approximate formula (4) was derived, namely that the number of solutions is large, is not satisfied. We leave it as an open problem to find the right balance and to analyze the speedup that can be achieved in general with these ideas.

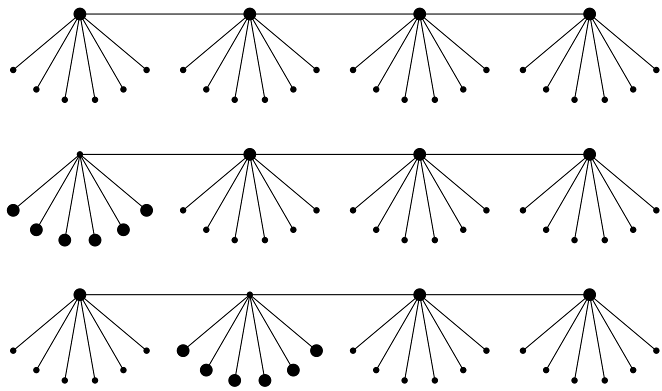


Fig. 7. Minimal dominating sets in a chain of stars

However, there is a limit on the speedup that one can hope for: The tree in Figure 7 consists of many clusters of leaves that are adjacent to a common vertex like in a star. By Observation 1(2), all these twins must belong to a minimal dominating set together. Thus, to go from one minimal dominating set to another, one has to completely swap at least one such cluster into or out of the solution. With k stars of size n/k, there are 2^k solutions, and it takes at least n/k time just to swap nodes in and out of any solution. Taking $k \approx a \log_2 n$ for some constant a produces an example with $\Theta(n^a)$ solutions and a total running time $\Omega(n^a \times n/\log n)$. This rules out a speed-up by more than a logarithmic factor, even if we allow arbitrary polynomial-time preprocessing.

In view of this example, it makes sense to lump clusters of twins together as a preprocessing step. From each cluster of twin leaves, one representative is chosen, and the other vertices go along with that representative. Essentially, this means that we delete all leaves except one representative from each cluster, or in other words, we consider only graphs without twins.

It seems that such graphs always have an exponential number of minimal dominating sets. We found empirically that, for $2 \le n \le 70$, the number of solutions is at least $2^{n/3}$. We calculated this by adapting the algorithm from Section 6 below to the *minimization* of the number of solutions. It turned out that when n is of the form $3k - 1$, the tree without twins that has the smallest number of minimal dominating sets is the extended comb with k teeth shown in Figure 1c. It consists of $2k - 1$ vertices on a path, with a leaf added to every other vertex. From each of the k teeth, one can independently choose one of the two vertices. Such a selection can be completed into a unique minimal dominating set by adding an appropriate subset of the $k - 1$ intermediate vertices between the teeth; thus, there are exactly $2^k = 2^{(n+1)/3}$ minimal dominating sets in this example. The best tree with $n = 3k - 2$ vertices has the same number 2^k of solutions, and it is obtained by removing the leftmost or rightmost leaf from the comb. For $n = 3k \ge 6$, the best tree has $\frac{7}{4} \cdot 2^k$ solutions. These statements are not proved to hold in general. The proof technique of Section 6.4 should be applicable, but we did not try.

The exponential number of solutions for trees without twins gives hope that one might be able to enumerate the minimal dominating sets in substantially sublinear average time, because occasional expensive updates can be amortized over a large number of outputs.

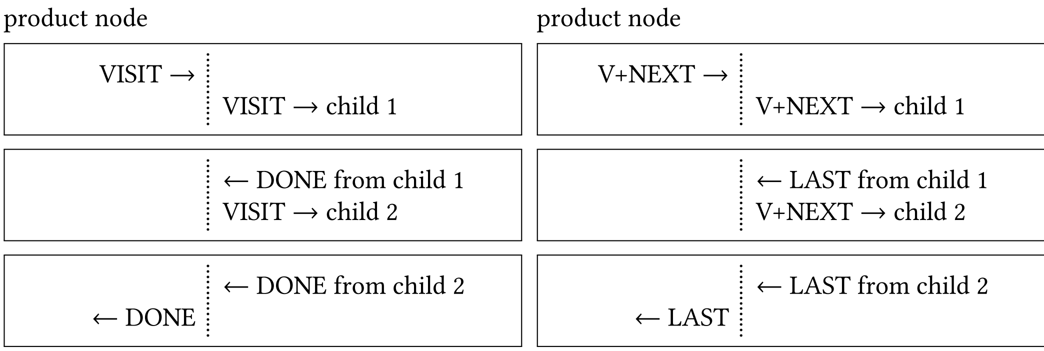


Fig. 8. Program for a product node

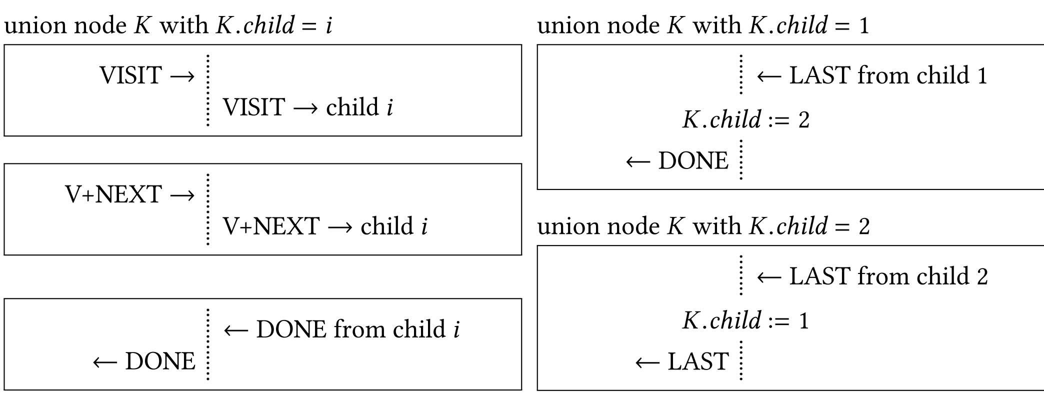


Fig. 9. Program for a union node

5.5 Implementation by message passing: Algorithm ENUM2

We give now a more explicit description of the enumeration procedure as a message-passing algorithm, without relying on the generator framework o PYTHON. At any time, there is one active node of the DAG. This node sends a message to one of its neighbors, and the action passes to that neighbor. The nodes maintain private state variables.

There are two types of *request messages*, which always flow downward in the network: VISIT and V+NEXT. There are two types of *reply messages*, which flow upward in response to the request messages: DONE and LAST.

The interaction follows a structured protocol: When a node K sends a message to one of its children K' for the first time, a bidirectional *channel* between K and K' is established, and K becomes the *parent* of K', for the time being. Over this channel, the flow of messages is a strict alternation between downward requests and upward replies:

$$\begin{array}{l} \rightarrow \text{V+NEXT} \\ \leftarrow \text{DONE} \\ \rightarrow \text{V+NEXT} \\ \leftarrow \text{DONE} \\ \dots \\ \rightarrow \text{V+NEXT} \\ \leftarrow \text{LAST} \end{array} \tag{5}$$

basis node K for vertex a, representing $\{\{a\}\}$	basis node K for vertex a, representing $\{\emptyset\}$
VISIT $\longrightarrow$ ⋮ report "$a \in D$" $\longleftarrow$ DONE ⋮	VISIT $\longrightarrow$ ⋮ report "$a \notin D$" $\longleftarrow$ DONE ⋮
V+NEXT $\longrightarrow$ ⋮ report "$a \in D$" $\longleftarrow$ LAST ⋮	V+NEXT $\longrightarrow$ ⋮ report "$a \notin D$" $\longleftarrow$ LAST ⋮

Fig. 10. Program for a basis node

master node	master node
START ⋮ V+NEXT $\longrightarrow$ target node	⋮ $\longleftarrow$ DONE from target node process the solution D ⋮ V+NEXT $\longrightarrow$ target node
⋮ $\longleftarrow$ LAST from target node process the solution D STOP	

Fig. 11. Program for the master node

The meaning of this exchange is as follows: V+NEXT stands for "VISIT and ADVANCE TO NEXT SOLUTION". It instructs the child node to "visit" one solution in the subtree of T for which it is responsible, and to advance the internal variables in the nodes of the DAG so that the next visit will produce the next solution. Successful completion is signaled by the DONE message. The LAST message signals in addition that the enumeration is completed and no more additional solutions are available. If K' represents m solutions, this dialogue will finish after $2m$ messages. The node K is then no longer the parent of K', and K' is ready to receive another V+NEXT instruction from a new parent. The state variables have been reset in such a way that the enumeration will then resume with the first solution.

The above dialogue can be interspersed with any number of VISIT/DONE pairs of the following type:

$$\begin{array}{l} \longrightarrow \text{VISIT} \\ \longleftarrow \text{DONE} \end{array} \tag{6}$$

This will just visit the current solution but not advance the pointers, so that the next VISIT or V+NEXT request will revisit the same solution.

To record the current status of the enumeration, every union node K has an attribute $K.child$ which is either 1 or 2. At the beginning, all *child* attributes are initialized to 1. These are the only pointers that need to be explicitly maintained. A union node K will have an open channel to at most one of its children at a time, as selected by $K.child$. A product node opens channels to both children simultaneously.

We present the program in Figures 8–11 in terms of simple *patterns*: For each node type and for each message that it potentially receives, there is one pattern. The pattern prescribes some actions or some variable change, and it terminates with sending a message. The message exchange with the parent is written on the left of the dotted line, the exchange with the children occurs on the right side. For example, the first box in Figure 8 says: If a product node receives a VISIT request (from its parent), it sends a VISIT request to its first child.

We add a *master node* with a single outgoing arc leading to the target node (Figure 11). Its only job is to send V+NEXT requests until the solutions are exhausted.

The program is very simple, but it is not immediate obvious from the patterns why it works. To gain some understanding, we will first analyze the set of nodes that are visited when generating one solution.

A subgraph E of the expression DAG is called a *well-structured enumeration tree* if it contains both children of every product node in E and exactly one child of every union node in E. The following lemma states some good properties of these graphs, justifying their name "*well-structured* enumeration *trees*".

Lemma 5.1. *1) A well-structured enumeration tree is a rooted directed tree, and its leaves are basis nodes.*
2) If the root of a well-structured enumeration tree is associated to the vertex set A, then its leaves are in one-to-one correspondence with the vertices of A,
3) A well-structured enumeration tree contains $\Theta(|A|)$ *nodes in total.*

Proof. 1) By definition, a well-structured enumeration tree E can branch only at product nodes. Since the two children of such a node are associated to disjoint subtrees of V, the two branches cannot meet, and therefore E is a tree. (This justifies the terminology of children and parents that we are using.) By definition, the leaves of the tree can only be basis nodes.

2) This follows from the properties of the expression DAG: When the tree branches at a product node, the associated set $A \subseteq V$ is split, and at a union node, which has only one child, the associated set is preserved.

3) By statement 2, the tree has $|A|$ leaves. As was argued towards the end of Section 5.1 on p. 9, a chain of non-branching union nodes has length at most 8. It follows that the tree has $\Theta(|A|)$ nodes. □

We apply this lemma to bound the number of nodes visited by the algorithm:

Lemma 5.2. *Let K be a node that is associated to a subtree A. We consider the period from the time when K receives a message from its parent to the first time when it returns a message to its parent.*
1) If K receives a VISIT *message, the visited nodes form a well-structured enumeration tree with root K. This tree is traversed in depth-first order. No variables are changed, and the node will return a* DONE *message to its parent after visiting* $\Theta(|A|)$ *nodes.*
2) Consequently, if the node K repeatedly receives VISIT *messages, the algorithm will revisit the same sequence of nodes again.*
3) If K receives a V+NEXT *message, the algorithm will visit the same sequence of nodes as if a* VISIT *message had been received. However, some variables may be changed, and the node may return a* DONE *or a* LAST *message to its parent.*

Proof. 1) It is easy to check that a VISIT message leads only to VISIT and DONE messages. The union and product nodes behave as shown in Figure 12. For a union node, the program goes to exactly one of the children, and for a product node, it recursively visits each child. Thus, the visited nodes form a well-structured enumeration tree. The running time follows from Lemma 5.1.

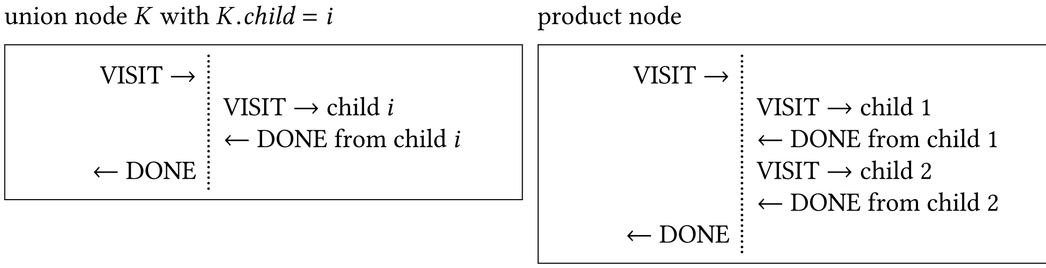


Fig. 12. The VISIT operation from the viewpoint of a union and a product node

2) This is an immediate consequence of the first statement.

3) One can easily check this by looking at the programs. The only difference to a VISIT is that some DONE replies may be changed to LAST, and the *child* attribute of some union nodes may change. □

If we apply the lemma to the target node, this shows that Algorithm ENUM2 has only a linear delay between successive solutions.

5.6 Correctness

To understand why the program is correct, we will focus on the messages sent and received from a single node. We have seen that VISIT messages are harmless, so let us restrict our attention to V+NEXT messages. We prove by induction that every node, when receiving a sequence of V+NEXT messages from a parent, will follow the protocol (5): Before each reply to the parent, it will set up a solution in its associated subtree, and it will cycle through all solutions and send back a LAST reply when it is done.

This is obvious for the basis nodes. For the union and product nodes, we assume inductively that each child follows the established protocol (5) from the first V+NEXT request to the LAST reply, and we get the program flow in Figure 13. It is a matter of comparing the charts with the programs of Figures 8 and 9 to check that they represent the true flow of actions. The left part of Figure 13 shows the process from the point of view of a union node K. We clearly see the two successive loops over the results of the two children. When the process terminates, $K.child$ is reset to 1. In this way, the node is reinitialized for the next loop.

The right part shows a product node. Every iteration descends first to child 1 and then to child 2 before responding to the parent node. We see two nested loops, but it does not look like the most natural implementation of loops: Since the advancement to the next solution (the "+NEXT" part) has to be requested when *entering* the child node, the advancement of the outer loop is done as part of the last iteration of the inner loop. As a consequence, the loop over child 1 is nested within the loop over child 2 (in contrast to the program ENUM1 of Figure 6). This allows the termination of the inner loop to be detected at the LAST visit of the first child and the appropriate action (V+NEXT instead of the default VISIT operation) to be taken for the second child.

In both types of nodes, the results are reported back to the parent in a cycle ending with a LAST message. The program is indeed a low-level implementation of the same loop structures for the recursive enumeration as in the program ENUM1, apart from the nesting order of the loops. We have thus shown that the algorithm correctly generates all solutions. There is a linear delay between consecutive solutions. The expression DAG in the preprocessing phase can be constructed also in linear time, thus establishing Theorem 1.2: The minimal dominating sets of a tree with n vertices can be enumerated with $O(n)$ setup time and with $O(n)$ delay between successive solutions.

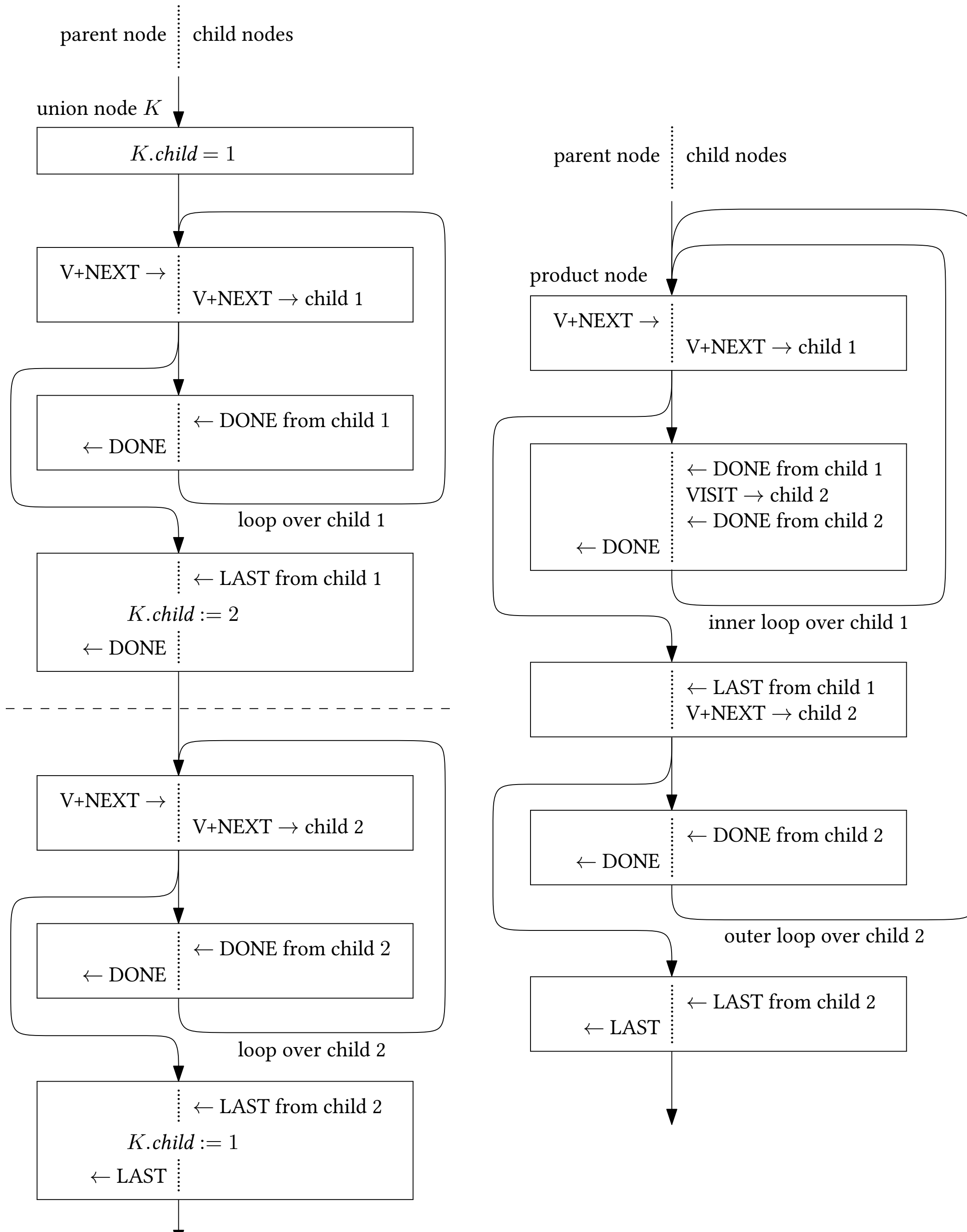


Fig. 13. Correctness is seen by observing the message flow from the viewpoint of a union node K (left) and from the viewpoint of a product node (right)

We give a few implementation hints that are not expressed in the programs above. A node must remember the parent from which it is currently receiving commands. Alternatively, the list of nodes that are still expecting replies can be maintained as a stack. In this way, the parent node can simply be popped from the stack when sending a message to it. Besides this stack, it may be convenient to maintain a *child* attribute also for a product node, to make it easy to know from which child a message is received.

5.7 Analysis of the Python implementation ENUM1

As mentioned, the concept of generator expressions in PYTHON uses a different convention for signaling the end of the data stream. Compared to Algorithm ENUM2, which signals the end of the data simultaneously with the delivery of the last item, PYTHON does this only in response to the subsequent request, just like an end-of-file condition is conventionally handled. Such a behavior is necessary in order to accommodate zero-length loops. Here is a side-by-side comparison between the two conventions.

Algorithm ENUM2 (5):	the PYTHON convention:
$\rightarrow$ V+NEXT	$\rightarrow$ NEXT
$\leftarrow$ DONE	$\leftarrow$ DONE
$\rightarrow$ V+NEXT	$\rightarrow$ NEXT
$\leftarrow$ DONE	$\leftarrow$ DONE
...	...
$\rightarrow$ V+NEXT	$\rightarrow$ NEXT
$\leftarrow$ LAST	$\leftarrow$ DONE
- - - - - -	- - - - - -
	$\rightarrow$ NEXT
	$\leftarrow$ STOP

The NEXT message corresponds to PYTHON's `next()` method, and the STOP message is PYTHON's `StopIteration` exception, which returns without producing a result. After receiving a STOP message, a node might have to go again to one of its children to produce an actual solution. Therefore, we need a more elaborate argument to show that the procedure still has only linear delay.

We remark that the simpler protocol (5) in the left column is only possible because there are no null nodes that produce no solution. Without this assumption, the linear-delay argument for the PYTHON version ENUM1 that we are going to present would also break down.

In Algorithm ENUM1, the union and product nodes do not perform any operations except coordinating the loops over their children. The control flow inside a node that results from these loops is shown in Figure 14. One difference to Algorithm ENUM2 is that ENUM1 does not visit a basis node for each vertex in every iteration. In the inner loop of a product node, the solution of the outer loop remains unchanged, and therefore it is not necessary to enter the corresponding part of the tree. This is the reason why there is no need for a separate VISIT message like in Algorithm ENUM2, (as opposed to V+NEXT). The loops are terminated by STOP messages. In the flow graphs of Figure 14, the very first NEXT message that starts an iteration has been marked with a star. This indicates that the node is entered by calling the function `enumerate_solutions`, while subsequent NEXT messages correspond to the cases when the node is re-entered after a *yield* statement.

A *visit* of a node is the time between receiving a request from a parent and sending back a reply. This includes recursive visits of descendant nodes. When a node replies DONE after "producing" a valid solution, we call this a *proper visit*. When a node replies STOP to signal that there are no more solutions, we speak of a *dummy visit*. When a node is entered for the first time, with a NEXT* request, it will always produce a solution. We denote such a proper visit a *first visit*.

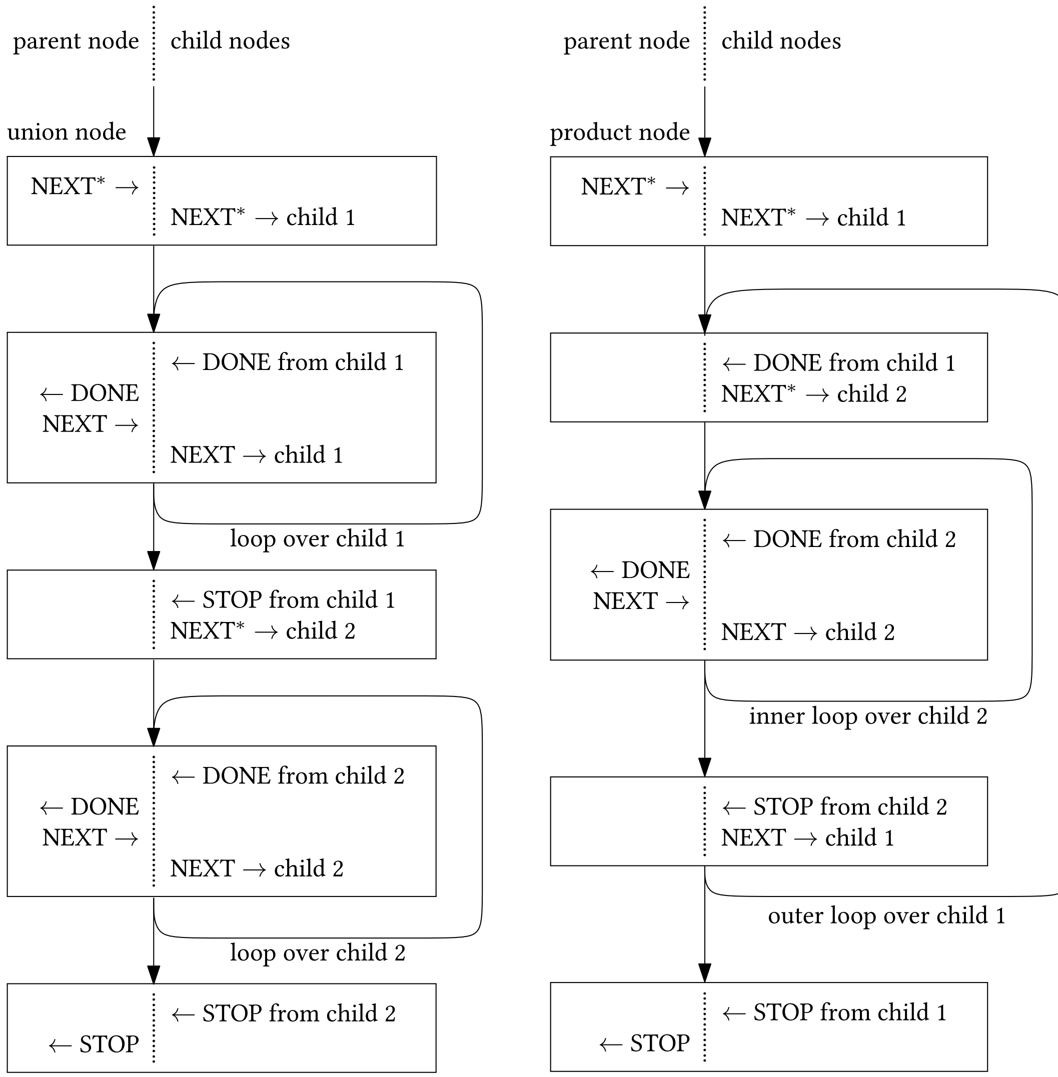


Fig. 14. Message flow of Algorithm ENUM1 in a union node K (left) and a product node (right)

Table 2 shows the visits to the child nodes that are caused by each type of visit. This information can be directly extracted from the flow graphs of Figure 14.

LEMMA 5.3. *Let K be a node that is associated to a subtree A. We consider a visit of K, from the time when K receives a message from its parent to the first time when it returns a message to its parent.*

1) *In a first visit and in a dummy visit, the set of visited nodes forms a well-structured enumeration tree with root K. In total, the number p of visited product nodes is $|A| - 1$.*
2) *In a proper visit, the total number p of visited product nodes is at most $2(|A| - 1)$.*
3) *Any visit is finished after visiting $O(|A|)$ nodes in total.*

PROOF. 1) It can be directly seen in Table 2 that dummy visits lead only to dummy visits, first visits lead only to first visits, and they follow the pattern of a well-structured enumeration tree.

node type	type of visit	visits of children
union node	first	first(1)
	proper	proper(1) or dummy(1) + first(2) or proper(2)
	dummy	dummy(2)
product node	first	first(1) + first(2)
	proper	proper(2) or dummy(2) + proper(1) + first(2)
	dummy	dummy(1) + dummy(2)

Table 2. The visits of the children (child 1 or child 2) that are spawned by a visit of a node, according to the type of visit. In this table, "proper" denotes a proper visit that is not a first visit.

2) We prove this by induction, following the partial order defined by the expression DAG. As induction basis, we consider the basis nodes. They have $|A| = 1$ and $p = 0$, and the statement is clearly true.

Let us now consider a union node K. If only one of its children is visited, induction works. The bad case is "dummy(1) + first(2)". But in that case, we apply statement 1 and get exactly $p = (|A| - 1) + (|A| - 1) = 2(|A| - 1)$ visited product nodes.

When K is a product node, let us denote the vertex sets associated to the children by A_1 and A_2, with $|A_1| + |A_2| = |A|$. The case "proper(2)" is easy: $p = 1 + 2(|A_2| - 1) \leq 2(|A| - 1)$. In the other case, "dummy(2) + proper(1) + first(2)", we apply the induction hypothesis for the first child and statement 1 of the lemma twice for the second child, and we get the upper bound

$$p \leq 1 + 2(|A_1| - 1) + 2(|A_2| - 1) < 2(|A| - 1).$$

3) Consider the tree of recursive node visits, with repetitions allowed: Every node appears as often as it is visited. Removing the product nodes decomposes the tree into components. Each component consists purely of union nodes, possibly extended with basis nodes at the leaves. If there are p visits to product nodes, the number of resulting components is at most $4p + 1$, since every product node has at most three arcs to its child visits and one arc to its parent.

We now use the property of the expression DAG that it contains at most 8 successive levels of union nodes without intervening product nodes. Thus, even if we generously allow every union node to cause 3 visits of its children, the number of visited union nodes in a component is bounded by a constant. Since the number of components is $O(p)$, the total number of visits is bounded by $O(p)$. By statements 1 and 2 of the lemma, $p = O(|A|)$, and the claim follows. □

Theorem 5.4. *The Python program ENUM1 of Section 5.3 enumerates the minimal dominating sets of a tree with linear delay, after linear setup time. After the last solution, the algorithm terminates in linear time.*

Proof. This follows from Lemma 5.3: Every solution is produced by a proper visit of the target node. After the last solution, there is a single dummy visit. □

A third algorithm ENUM3, similar in spirit to the Python program but without dummy visits, is given in Appendix B.

6 UPPER BOUNDS

We will now use the counting algorithm of Section 4 to analyze the possible numbers of minimal dominating sets among the trees with n vertices:

The following iteration computes the set $\mathcal{V}_n$ of all possible vectors of rooted trees of n vertices.

$$\mathcal{V}_1 := \{(0,1,0,0,0,1)\} \tag{7}$$

$$\mathcal{V}_n := \bigcup_{1 \le i < n} \mathcal{V}_i \circ \mathcal{V}_{n-i}, \text{ for } n \ge 2 \tag{8}$$

The operation $\circ$ in (8) is the elementwise composition using $\star$ applied to sets of vectors:

$$V \circ V' = \{\, x \star y \mid x \in V, y \in V' \,\}$$

The largest number M_n of minimal dominating sets among the trees with n vertices is then obtained by the following formula:

$$M_n = \max\{\, \bar{M}(v) \mid v \in \mathcal{V}_n \,\} = \max\{\, G + S + d + p \mid (G,S,L,d,p,f) \in \mathcal{V}_n \,\} \tag{9}$$

Table 3 below tabulates the results of this computation, and Figure 15 represents it graphically. We will discuss the results in Section 6.3.

Incidentally, with the same recursion, we also determined the *smallest* number of minimal dominating sets that a tree can have: it is 2, for trees with at least 2 vertices, as witnessed by the star $K_{1,n-1}$. It is easy to see that there must always be at least 2 minimal dominating sets: A tree is a bipartite graph, and in a connected bipartite without isolated vertices, each color class forms a minimal dominating set.

6.1 Data reduction by majorization

The last column in Table 3 reports the sizes of the sets $\mathcal{V}_n$. These sets get very large, and it is advantageous to remove vectors that cannot contribute to trees with the maximum number of minimal dominating sets.

If the elementwise order

$$(G_1, S_1, L_1, d_1, p_1, f_1) \ge (G_2, S_2, L_2, d_2, p_2, f_2)$$

holds for two vectors in $\mathcal{V}_i$, we can obviously omit $(G_2, S_2, L_2, d_2, p_2, f_2)$ from $\mathcal{V}_i$ without losing the chance to find the largest number of minimal dominating sets. This is true because the operation $\star$ is monotone in both arguments. We say that $(G_1, S_1, L_1, d_1, p_1, f_1)$ *majorizes* $(G_2, S_2, L_2, d_2, p_2, f_2)$. (Normally, we would call this relation *dominance*, but since we are using "dominating" sets already with a graph-theoretic meaning, we have chosen this alternative term.)

A more widely applicable majorization rule is obtained by observing that there is a partial order of *preference* between the categories:

$$\mathbf{G} > \mathbf{S} > \mathbf{L} \text{ and } \mathbf{d} > \mathbf{p} \tag{10}$$

This means, for example, that **G** is less restrictive than **S** in the following sense: Consider a minimal dominating set for T, whose intersection with a subtree A is of category **S**. Replacing this partial solution inside A by any other partial solution of category **G** will lead to a valid minimal dominating set. As a consequence, replacing a partial solution D of category **S** by a partial solution of category **G** in the subtree A cannot reduce the number of minimal dominating sets that can be built by extending D to the whole tree T.

A formal proof of this claim is based on the fact that the $\star$-operation is monotone in both arguments with respect to the partial order (10). It can be checked in Table 1 that, for example, $\mathbf{G} \star B$ is at least as good as $\mathbf{S} \star B$ according to the partial order, or that $A \star \mathbf{d}$ is always at least as good as $A \star \mathbf{p}$. In this comparison, any result category is of course preferable to the case "–" when no valid solution is built. Also, changing a category to a more preferred category will never change a final category (which is counted as a solution) to a non-final one.

As a consequence, if, for instance, we subtract 1 from S and add 1 to G, the new vector $(G+1, S-1, L, d, p, f)$ ought to majorize the original vector (G, S, L, d, p, f), even though the elementwise comparison fails. An easy way to accommodate these more powerful majorization rules is to transform the vectors (G, S, L, d, p, f) into

$$(G,\ G+S,\ G+S+L,\ d,\ d+p,\ f)$$

before comparing them elementwise. We denote this wider majorization criterion by the symbol $\succeq$, and define

$$(G_1, S_1, L_1, d_1, p_1, f_1) \succeq (G_2, S_2, L_2, d_2, p_2, f_2) \iff$$
$$(G_1, G_1+S_1, G_1+S_1+L_1, d_1, d_1+p_1, f_1) \geq (G_2, G_2+S_2, G_2+S_2+L_2, d_2, d_2+p_2, f_2),$$

where the comparison on the right-hand-side is just the elementwise comparison between 6-tuples.

We summarize our considerations in the following lemma

Lemma 6.1. *1) If $v \succeq v'$ and $w \succeq w'$ then $v \star w \succeq v' \star w'$.*
2) If $v \succeq v'$, then $\bar{M}(v) \geq \bar{M}(v')$.
3) If $v \succeq v'$ holds for two vectors $v, v' \in \mathcal{V}_i$, we may remove v' from $\mathcal{V}_i$ without changing the sizes M_n of the largest minimal dominating sets found in the recursion (7–9).

Proof. The first statement follows from the monotonicity of the composition of Table 1 when applied to single categories, as discussed above. Alternatively, it can be checked by a straightforward calculation. The second statement is easy to see.

To see the third claim, we introduce the *majorized hull* of a set $P \subseteq \mathbb{R}^6_{\geq 0}$, denoted by hull$(P)$: It is the set of all nonnegative 6-vectors that are majorized by some vector in P according to the relation $\succeq$:

$$\mathrm{hull}(P) := \{x \in \mathbb{R}^6_{\geq 0} \mid x \preceq y \text{ for some } y \in P\}$$

When representing hull(P), we can remove from P all elements that are majorized by other elements. Algebraically, the justification for this reduction comes from the following equations.

$$\mathrm{hull}(P \cup Q) = \mathrm{hull}(\mathrm{hull}(P) \cup \mathrm{hull}(Q)) \tag{11}$$
$$\mathrm{hull}(P \circ Q) = \mathrm{hull}(\mathrm{hull}(P) \circ \mathrm{hull}(Q)) \tag{12}$$

Equation (11) follows from the transitivity of $\preceq$, and (12) comes directly from part 1 of the lemma.

Reading the equations (11–12) from left to right, they say: If we are interested only in the hull of a union $P \cup Q$ or a "product" $P \circ Q$, we might as well take the hull of P and Q before performing the operation. By statement 2 of the lemma, the hull of $\mathcal{V}_n$ is sufficient for computing M_n by (9). Since the set $\mathcal{V}_n$ is built up in the iteration (8) from smaller sets $\mathcal{V}_i$ by $\circ$ and $\cup$ operations, this justifies the application of the hull operation at every level, proving part 3 of the lemma. □

6.2 The convex hull

We can further reduce the size of the point sets by taking the convex hull, conv(P). We combine the convex hull and the majorized hull in one operation $\mathrm{hull}^+(P) = \mathrm{conv}(\mathrm{hull}(P)) = \mathrm{hull}(\mathrm{conv}(P))$, which we call the *majorized convex hull.* The majorized convex hull can also be formed by taking the convex hull together with the rays in directions $(-1,1,0,0,0,0)$, $(0,-1,1,0,0,0)$, $(0,0,0,-1,1,0)$, as well as the coordinate directions $(0,0,-1,0,0,0)$, $(0,0,0,0,-1,0)$, and $(0,0,0,0,0,-1)$, and clipping the result to the nonnegative orthant.

We have the same properties as for the majorized hull:

Lemma 6.2.

$$\operatorname{conv}(P \cup Q) = \operatorname{conv}(\operatorname{conv}(P) \cup \operatorname{conv}(Q)) \tag{13}$$

$$\operatorname{conv}(P \circ Q) = \operatorname{conv}(\operatorname{conv}(P) \circ \operatorname{conv}(Q)) \tag{14}$$

$$\operatorname{hull}^+(P \cup Q) = \operatorname{hull}^+(\operatorname{hull}^+(P) \cup \operatorname{hull}^+(Q)) \tag{15}$$

$$\operatorname{hull}^+(P \circ Q) = \operatorname{hull}^+(\operatorname{hull}^+(P) \circ \operatorname{hull}^+(Q)) \tag{16}$$

Proof. Equation (13) is standard. To prove (14), we first prove

$$\operatorname{conv}(P \circ Q) \supseteq \operatorname{conv}(P) \circ \operatorname{conv}(Q), \tag{17}$$

using the fact that the function $\star\colon \mathbb{R}^6_{\geq 0} \times \mathbb{R}^6_{\geq 0} \to \mathbb{R}^6_{\geq 0}$ is bilinear. An element formed from two convex combinations on the right-hand side is of the form

$$\sum_i \mu_i p_i \star \sum_j \nu_j q_j = \sum_i \sum_j \mu_i \nu_j (p_i \star q_j),$$

with $\sum_i \sum_j \mu_i \nu_j = 1$, and is hence an element of $\operatorname{conv}(P \circ Q)$. From (17), the inclusion $\operatorname{conv}(P \circ Q) \supseteq \operatorname{conv}(\operatorname{conv}(P) \circ \operatorname{conv}(Q))$ follows by a standard convexity argument, and the reverse conclusion is an easy consequence of the inclusion $P \subseteq \operatorname{conv}(P)$.

The two last equations, (15) and (16), follow by combining the equations (13–14) for the convex hull with the equations (11–12) for the majorized hull. □

We are interested in the maximum total $\bar{M}$, which is a linear function, and hence the convex hull is sufficient. Equation (14) tells us that to compute $\operatorname{conv}(P \circ Q)$, it is sufficient to compute $v \star w$ for the vertices of P and Q and take the convex hull.

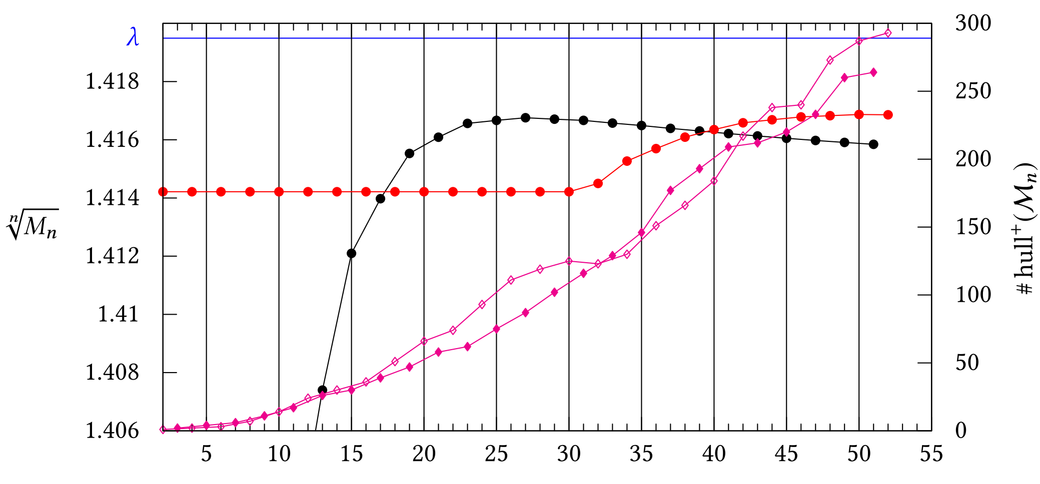


Fig. 15. The n-th root of the maximum number M_n of minimal dominating sets of trees with n vertices. Even and odd values of n (red and black dots) behave differently. The pink curves through the diamonds show the growth of the convex non-majorized hulls, $\operatorname{hull}^+(\mathcal{V}_n)$. Again, even and odd values of n behave differently.

n	$\sqrt[n]{M_n}$	M_n	$\#\,\mathrm{hull}^+(\mathcal{V}_n)$	$\#\,\mathrm{hull}(\mathcal{V}_n)$	$\lvert\mathcal{V}_n\rvert$
1	1	1	1	1	1
2	1.41421356237310	2	1	1	1
3	1.25992104989487	2	2	2	2
4	1.41421356237310	4	2	2	4
5	1.31950791077289	4	4	4	7
6	1.41421356237309	8	3	5	13
7	1.36873810664220	9	6	9	24
8	1.41421356237310	16	7	13	45
9	1.38702322584422	19	11	19	85
10	1.41421356237310	32	14	32	159
11	1.40157620020641	41	17	39	308
12	1.41421356237309	64	24	73	588
13	1.40739771128108	85	26	85	1180
14	1.41421356237309	128	30	144	2326
15	1.41209815120249	177	30	176	4753
16	1.41421356237310	256	36	279	9591
17	1.41397457411881	361	39	337	19793
18	1.41421356237309	512	51	492	40638
19	1.41553085871039	737	47	612	84641
20	1.41421356237310	1024	66	841	176255
21	1.41608793848702	1489	58	1055	369635
22	1.41421356237310	2048	74	1320	775935
23	1.41656252137841	3009	62	1641	1634901
24	1.41421356237309	4096	93	1969	3451490
25	1.41666558384650	6049	75	2435	7303232
26	1.41421356237310	8192	111	2805	15481738
27	1.41675632056381	12161	87	3456	32868146
28	1.41421356237309	16384	119	3871	
29	1.41670718070637	24385	102	4656	
30	1.41421356237310	32768	125	5329	
31	1.41666501243844	48897	116	6227	
32	1.41449859435768	65960	123	7248	
33	1.41657202787702	97921	129	8436	
34	1.41526678247498	134432	130	9719	
35	1.41648981352598	196097	146	11277	
36	1.41569656428574	272224	151	12878	
37	1.41639156076937	392449	177	14890	
38	1.41609068088382	551392	166	16931	
39	1.41630342192653	785409	193	19088	
40	1.41634892845829	1113808	184	22214	
41	1.41621264079532	1571329	209	24075	
42	1.41658315523612	2249920	217	28344	
43	1.41613031644569	3143681	212	30029	
44	1.41668758343879	4529600	238	35068	
45	1.41605019185075	6288385	220	36809	
46	1.41678485046458	9119680	240	42438	
47	1.41597689193916	12578817	233	44773	
48	1.41682808199910	18332576	273	50902	
49	1.41590722737106	25159681	260	54417	
50	1.41686791092506	36852608	287	61859	
51	1.41584303009330	50323457	264	66246	
52	1.41685798299446	73955200	293		

Table 3. The maximum number M_n of minimal dominating sets of a tree with n vertices. $\#\,\mathrm{hull}(\mathcal{V}_n)$ denotes the number of generating vertices of $\mathrm{hull}(\mathcal{V}_n)$ (the non-majorized vertices of $\mathcal{V}_n$), and $\#\,\mathrm{hull}^+(\mathcal{V}_n)$ is the number of extreme non-majorized vertices in $\mathrm{hull}^+(\mathcal{V}_n)$.

6.3 The upper bound for trees of a given size

We have carried out the iteration (8) for calculating M_n, both with the majorized hull, hull($\mathcal{V}_n$), and the majorized convex hull, hull$^+$($\mathcal{V}_n$). The results are presented in Table 3 and Figure 15. Figure 15 shows clearly that the trees with even and odd n behave differently. For a while, $\sqrt[n]{M_n}$ for the even trees remains constant at $\sqrt{2}$, which comes from the comb graphs of Figure 1a, while the odd trees rise from a low start. They overtake the even trees for $n = 19$ and reach a local maximum at $n = 27$. The corresponding value $\sqrt[27]{12161} \approx 1.416756$ was the best lower bound on λ known so far, due to Krzywkowski [2013]. The optimal tree with 27 vertices, which has 12161 minimal dominating sets, consists of two snowflakes and an additional vertex that is attached to the centers of the two snowflakes. We suspect that Krzywkowski must have run a program like ours to come up with this tree. In fact, *all* optimal trees of odd order that are reported in the table have the same "double-snowflake" structure, see for example the left and the right half in Figure 16. The number of arms of the snowflakes must be varied to reach the desired number of vertices; the arms are distributed as equally as possible to the two snowflakes. (For $n \le 7$, these trees degenerate to paths.) At $n = 32$, the even values start to increase, leading to new records for $n \ge 46$, while the odd values continue to decrease. All optimal trees of even order n that we found for $n \ge 32$ have a similar structure, see Figure 16. They consist of two double-snowflakes of odd order n_1 and n_2 with $n_1 + n_2 = n$ and n_1 and n_2 as close together as possible, connected by an edge between two snowflake centers. When there is a choice, the center of the smaller snowflake is used as an endpoint of the connecting edge. The trees of this pattern reach their local maximum at $\sqrt[50]{M_{50}} = \sqrt[50]{36\,852\,608} \approx 1.41686791$. Beyond this size, they decline, and at some point, trees with three, five, or six snowflakes will probably begin to take the lead.

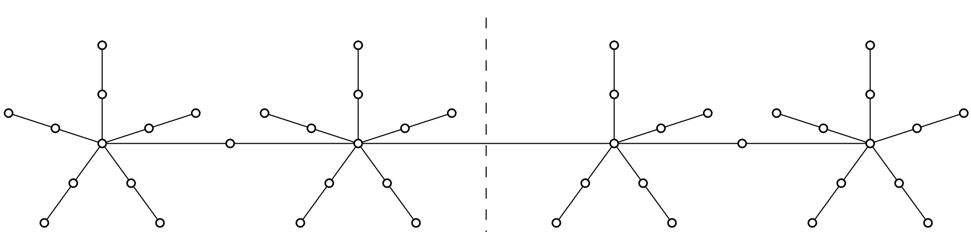


Fig. 16. An optimal tree with 44 vertices. The left and the right half is an optimal tree with 23 and 21 vertices, respectively.

The even optimal trees with $2^{n/2}$ minimal dominating sets are far from unique: One can start with an arbitrary tree on $n/2$ vertices and add a new leaf adjacent to each vertex, see Figure 1b. We did not check whether the other classes of optimal trees that we found are unique.

In Figure 15 it is apparent that the values $\sqrt[n]{M_n}$ stay well below the true bound λ. There is no way how one could have guessed the limiting behavior from these numbers, even if the range of sizes n could be substantially extended.

We can now describe how Part 2 of Theorem 1.1 is obtained. For $n \ge 38$, we construct a tree with at least $0.649748 \cdot \lambda^n$ minimal dominating sets with the help of the supermultiplicativity property of Observation 1(4) as follows. If $n \ge 37$ and n is congruent to $1, 2, \ldots, 13$ modulo 13, we combine the optimum tree of size $0, 14, 2, 16, 4, 18, 6, 20, 8, 35, 10, 37, 12$ from Table 3 with a record tree RT_{13k+1} from the end of Section 3 of appropriate size. (The factor 0.649748 in the claim is restricted by the tree of size 37 in this list.) For $n < 37$, the trees in Table 3 do the job.

Implementation details and program runs. The version of the program which uses only the majorized hull for pruning points is very straightforward and did not pose any challenges. We used a pairwise comparison of all generated elements to remove majorized vectors. The program was written in the Python programming language and has less than 100 lines, including rudimentary code to print optimal trees. As the fifth column of Table 3 shows, the number $\#\operatorname{hull}(\mathcal{V}_n)$ of non-majorized vectors grows quite large.

Therefore, we used the convex hull to further reduce the number of points that need to be stored and processed. For the convex-hull computations, we tested for each generated vector whether it is a convex combination of the remaining vectors, and deleted it in case of a positive answer. This test can be formulated as a linear programming problem. We wrote our program for the mathematical software system Sage[1], which provides straightforward access to linear programming. We used the default solver GLPK that is installed with Sage. As the fourth column shows, using the convex hull leads to a substantial reduction of the number $\#\operatorname{hull}^+(\mathcal{V}_n)$ of vertices that need to be stored and processed, allowing us to carry the computation further than without the convex-hull computations. We managed to compute the values up to M_{52}. The number of non-majorized convex hull vertices appears to increase quadratically with n. This means that the number of points that are generated in (8) and subjected to the redundancy test in the computation of each new entry M_n grows like n^5. The calculations ran for several weeks.

We must concede that, due to the error-prone nature of floating-point computations, the reported value for M_{52} cannot be considered as reliable. It is conceivable that an extreme vertex is erroneously pruned because of numerical errors in the solution of a linear program, leading to missing trees. However, as the dimension of the problem and the involved numbers are not very big, this is probably not an issue. (By contrast, for the results that we will mention below in Section 6.4, we undertook the effort to certify the linear-programming results a posteriori.) For $n \le 51$, where a number is reported in the fifth column, the values M_n are not subject to these reservations, because they are confirmed by the reliable calculations without convex-hull computation, which took several months. In any case, the given value of M_{52} is certainly valid as a lower bound, as it comes from a computation that represents an actual tree.

6.4 Characterization of the growth rate

Since the sequence M_n is supermultiplicative (Observation 1(4)) and bounded by an exponential function $M_n \le 2^n$, it follows from Fekete's Lemma that the limit

$$\lambda^* := \lim_{n\to\infty} \sqrt[n]{M_n} \tag{18}$$

exists and that

$$M_n \le (\lambda^*)^n. \tag{19}$$

In contrast to the previous parts, we now denote the growth rate by λ^*, and we will use λ for a generic "test value", not necessarily the correct growth rate. The following statement provides a characterization of λ^*.

Proposition 6.3. *The growth constant λ^* equals the smallest the value λ for which there exists a bounded convex set P with $P = \operatorname{hull}^+ P$ such that*

$$(0, 1, 0, 0, 0, 1)/\lambda \in P \tag{20}$$

and

$$P \circ P \subseteq P. \tag{21}$$

[1] http://www.sagemath.org/

Proof. First we show that the statement does not change if we omit the condition that P is convex and that $P = \text{hull}^+(P)$: If this condition is not fulfilled by some set P, we can simply replace P with $\text{hull}^+(P)$. This will of course not affect (20), and by (16), taking the majorized convex hull of P does not invalidate the condition $P \circ P \subseteq P$.

We can write down the smallest set P fulfilling the required properties (20) and (21). It is

$$P_0 := \bigcup_{n \ge 1} \mathcal{V}_n / \lambda^n. \tag{22}$$

Let us see why this is true. By assumption (20), $\mathcal{V}_1/\lambda = \{(0,1,0,0,0,1)/\lambda\}$ must be contained in P_0. Let us now consider a vector $v \in \mathcal{V}_n$. It must be the result $w \star w'$ for some vectors $w \in \mathcal{V}_i$ and $w' \in \mathcal{V}_j$ with $i + j = n$. If we assume by induction that w/λ^i and w'/λ^j are in P_0, we conclude from (21) that $w/\lambda^i \star w'/\lambda^j = v/\lambda^n$ is also in P_0.

We will now prove the proposition through a sequence of equivalent statements:

$$\begin{aligned}
\text{bounded } P \text{ exists for } \lambda &\iff P_0 \text{ is bounded} && (23)\\
&\iff \text{the sequence } \|\mathcal{V}_n\|_1/\lambda^n \text{ is bounded} && (24)\\
&\iff \text{the sequence } M_n/\lambda^n \text{ is bounded} && (25)\\
&\iff \lim_{n\to\infty} \sqrt[n]{M_n/\lambda^n} \le 1 && (26)\\
&\iff \lambda^*/\lambda \le 1 \iff \lambda \ge \lambda^* && (27)
\end{aligned}$$

The equivalence between the first and the last statement is the claim of the proposition.

The equivalence (23) has already been shown above. In (24), we have decided to use the l_1 norm for expressing boundedness: $\|\mathcal{V}_n\|_1 := \max\{\, \|v\|_1 \mid v \in \mathcal{V}_n \,\}$. The equivalence follows from the definition (22) of P_0. When proceeding to (25), we are replacing the l_1-norm $\|v\|_1$ by the function $\bar{M}(v)$, which sums only 4 of the 6 entries of v. To justify this change, we show that it does not change the notion of boundedness. It is sufficient to prove the following relation:

$$M_n \le \|\mathcal{V}_n\|_1 \le M_{n+3} \tag{28}$$

The left inequality is trivial, because $G + S + d + f \le G + S + L + d + p + f$. The converse inequality is not true, because the categories **L** and **p** are not counted for $\bar{M}$. However, by appending a path of length 3 to the root, we ensure that every partial solution, no matter of which category, can be completed to a valid minimal dominating set in the larger tree. Algebraically, this can be checked by the following calculation:

$$\begin{aligned}
v_0 \star (v_0 \star (v_0 \star (G,S,L,d,p,f))) &= (G+S+L, d+f, d+p, G+S+d+p, f, G+d+f)\\
\bar{M}(v_0 \star (v_0 \star (v_0 \star (G,S,L,d,p,f)))) &= 2G + 2S + L + 2d + p + 2f \ge \|(G,S,L,d,p,f)\|_1
\end{aligned}$$

This means that, for every tree with n nodes and vector v, there is a tree with $n+3$ nodes and vector v' such that $\bar{M}(v') \ge \|v\|_1$. This establishes the right inequality of (28).

Let us proceed to the equivalence between (25) and (26). It is obvious except in the borderline case when the limit $\lim_{n\to\infty} \sqrt[n]{M_n/\lambda^n}$ equals 1, so let us postpone this case for the moment. The remaining steps till (27) are straightforward in view of the known value of the limit (18).

For the borderline case $\lambda = \lambda^*$, (19) tells us that $M_n/\lambda^n \le 1$ for all n, and thus the equivalence between (25) and (26–27) holds also in this case. □

6.5 Automatic determination of the growth factor

The property of P that is required in Proposition 6.3 is monotone in the sense that if it can be fulfilled for some λ, the same set P will also work for all larger values of λ. This holds because

$v_1 = v_1 \star v_{32}$	$= (0.9, 0, 0, 0, 0, 0)$	$v_{27} = v_{20} \star v_3 = v_{13} \star v_5$	$= (16, 1, 31, 62, 0, 2)\lambda^{-13}$
v_2	$= (0, 1, 0, 0, 0, 1)\lambda^{-1}$	$v_{28} = v_{21} \star v_3 = v_9 \star v_8$	$= (24, 1, 15, 60, 0, 4)\lambda^{-13}$
$v_3 = v_2 \star v_2$	$= (1, 0, 0, 1, 0, 0)\lambda^{-2}$	$v_{29} = v_6 \star v_{12} = v_{22} \star v_3$	$= (28, 1, 7, 56, 0, 8)\lambda^{-13}$
$v_4 = v_2 \star v_3$	$= (0, 1, 1, 1, 0, 1)\lambda^{-3}$	$v_{30} = v_4 \star v_{17} = v_{23} \star v_3$	$= (30, 1, 3, 48, 0, 16)\lambda^{-13}$
$v_5 = v_2 \star v_4$	$= (1, 1, 0, 1, 1, 1)\lambda^{-4}$	$v_{31} = v_2 \star v_{25}$	$= (30, 9, 2, 32, 8, 24)\lambda^{-13}$
$v_6 = v_4 \star v_3$	$= (0, 1, 3, 3, 0, 1)\lambda^{-5}$	$v_{32} = v_2 \star v_{24}$	$= (31, 1, 1, 32, 0, 32)\lambda^{-13}$
$v_7 = v_2 \star v_5$	$= (1, 1, 1, 2, 0, 2)\lambda^{-5}$	$v_{33} = v_2 \star v_{26}$	$= (1, 63, 0, 1, 63, 63)\lambda^{-14}$
$v_8 = v_2 \star v_6$	$= (1, 3, 0, 1, 3, 3)\lambda^{-6}$	$v_{34} = v_2 \star v_{27}$	$= (2, 62, 16, 17, 31, 62)\lambda^{-14}$
$v_9 = v_6 \star v_3$	$= (0, 1, 7, 7, 0, 1)\lambda^{-7}$	$v_{35} = v_{26} \star v_3$	$= (0, 1, 127, 127, 0, 1)\lambda^{-15}$
$v_{10} = v_7 \star v_3 = v_4 \star v_5$	$= (2, 1, 3, 6, 0, 2)\lambda^{-7}$	$v_{36} = v_{19} \star v_5 = v_{27} \star v_3$	$= (32, 1, 63, 126, 0, 2)\lambda^{-15}$
$v_{11} = v_2 \star v_8$	$= (3, 1, 1, 4, 0, 4)\lambda^{-7}$	$v_{37} = v_{13} \star v_8 = v_{28} \star v_3$	$= (48, 1, 31, 124, 0, 4)\lambda^{-15}$
$v_{12} = v_2 \star v_9$	$= (1, 7, 0, 1, 7, 7)\lambda^{-8}$	$v_{38} = v_9 \star v_{12} = v_{29} \star v_3$	$= (56, 1, 15, 120, 0, 8)\lambda^{-15}$
$v_{13} = v_9 \star v_3$	$= (0, 1, 15, 15, 0, 1)\lambda^{-9}$	$v_{39} = v_{30} \star v_3 = v_6 \star v_{17}$	$= (60, 1, 7, 112, 0, 16)\lambda^{-15}$
$v_{14} = v_6 \star v_5 = v_{10} \star v_3$	$= (4, 1, 7, 14, 0, 2)\lambda^{-9}$	$v_{40} = v_4 \star v_{24} = v_{32} \star v_3$	$= (62, 1, 3, 96, 0, 32)\lambda^{-15}$
$v_{15} = v_{11} \star v_3 = v_4 \star v_8$	$= (6, 1, 3, 12, 0, 4)\lambda^{-9}$	$v_{41} = v_{26} \star v_5 = v_{36} \star v_3$	$= (64, 1, 127, 254, 0, 2)\lambda^{-17}$
$v_{16} = v_2 \star v_{12}$	$= (7, 1, 1, 8, 0, 8)\lambda^{-9}$	$v_{42} = v_{19} \star v_8 = v_{37} \star v_3$	$= (96, 1, 63, 252, 0, 4)\lambda^{-17}$
$v_{17} = v_2 \star v_{13}$	$= (1, 15, 0, 1, 15, 15)\lambda^{-10}$	$v_{43} = v_{38} \star v_3 = v_{13} \star v_{12}$	$= (112, 1, 31, 248, 0, 8)\lambda^{-17}$
$v_{18} = v_2 \star v_{14}$	$= (2, 14, 4, 5, 7, 14)\lambda^{-10}$	$v_{44} = v_9 \star v_{17} = v_{39} \star v_3$	$= (120, 1, 15, 240, 0, 16)\lambda^{-17}$
$v_{19} = v_{13} \star v_3$	$= (0, 1, 31, 31, 0, 1)\lambda^{-11}$	$v_{45} = v_6 \star v_{24} = v_{40} \star v_3$	$= (124, 1, 7, 224, 0, 32)\lambda^{-17}$
$v_{20} = v_9 \star v_5 = v_{14} \star v_3$	$= (8, 1, 15, 30, 0, 2)\lambda^{-11}$	$v_{46} = v_{26} \star v_8 = v_{42} \star v_3$	$= (192, 1, 127, 508, 0, 4)\lambda^{-19}$
$v_{21} = v_6 \star v_8 = v_{15} \star v_3$	$= (12, 1, 7, 28, 0, 4)\lambda^{-11}$	$v_{47} = v_{43} \star v_3 = v_{19} \star v_{12}$	$= (224, 1, 63, 504, 0, 8)\lambda^{-19}$
$v_{22} = v_4 \star v_{12} = v_{16} \star v_3$	$= (14, 1, 3, 24, 0, 8)\lambda^{-11}$	$v_{48} = v_{13} \star v_{17} = v_{44} \star v_3$	$= (240, 1, 31, 496, 0, 16)\lambda^{-19}$
$v_{23} = v_2 \star v_{17}$	$= (15, 1, 1, 16, 0, 16)\lambda^{-11}$	$v_{49} = v_9 \star v_{24} = v_{45} \star v_3$	$= (248, 1, 15, 480, 0, 32)\lambda^{-19}$
$v_{24} = v_2 \star v_{19}$	$= (1, 31, 0, 1, 31, 31)\lambda^{-12}$	$v_{50} = v_{26} \star v_{12} = v_{47} \star v_3$	$= (448, 1, 127, 1016, 0, 8)\lambda^{-21}$
$v_{25} = v_2 \star v_{20}$	$= (2, 30, 8, 9, 15, 30)\lambda^{-12}$	$v_{51} = v_{48} \star v_3 = v_{19} \star v_{17}$	$= (480, 1, 63, 1008, 0, 16)\lambda^{-21}$
$v_{26} = v_{19} \star v_3$	$= (0, 1, 63, 63, 0, 1)\lambda^{-13}$	$v_{52} = v_{49} \star v_3 = v_{13} \star v_{24}$	$= (496, 1, 31, 992, 0, 32)\lambda^{-21}$

$v_{53} = v_{24} \star v_{19}$	$= (63, 961, 0, 63, 1922, 961)\lambda^{-23}$
$v_{54} = v_{52} \star v_3 = v_{19} \star v_{24}$	$= (992, 1, 63, 2016, 0, 32)\lambda^{-23}$
$v_{55} = v_{33} \star v_{26}$	$= (127, 3969, 0, 127, 7938, 3969)\lambda^{-27}$

Table 4. The 55 vertices generating the polytope P; $\lambda = \sqrt[13]{95} \approx 1.4195$.

P contains its majorized hull, and therefore property (20) remains fulfilled. This monotonic behavior opens the way for a semi-automatic experimental way to search for the correct growth factor λ^*.

1) Choose a trial value λ, and set $Q := \{(0, 1, 0, 0, 0, 1)/\lambda\}$.
2) Form the set $Q^2 := Q \circ Q$ of all pairwise products of Q.
3) Compute $P := \mathrm{hull}^+(Q \cup Q^2)$.
4) Let Q be the set of non-majorized vertices of P.
5) Repeat from Step 2 until the process converges or diverges.
6) If divergence occurs, λ was chosen too small, and a larger value must be tried. In case of convergence, try a smaller value.

In practice, divergence in Step 5) manifests itself in an exponential growth of the vector entries and is easy to detect once it sets in. The trees corresponding to the vectors which are "responsible" for the divergence have more than λ^n minimal dominating sets. By looking at such trees, we got the idea for the lower-bound construction of the star of snowflakes. In Section 3, we showed how the growth λ of this family of examples can be estimated easily. As it turned out, we were lucky, and the growth rate $\lambda = \sqrt[13]{95}$ of this construction was the correct value λ^*.

With this value of λ, we eventually determined a set P which does the job of proving the upper bound by Proposition 6.3. It is the set $P = \text{hull}^+(\{v_1, \ldots, v_{55}\})$ with the vectors given in Table 4, The *seed vector* $v_2 = (0, 1, 0, 0, 0, 1)/\lambda$ is in P by construction, and thus the first requirement on P is fulfilled. The vectors other than v_1 correspond to actual trees, and the exponent of $1/\lambda$ given in the table is their size. By looking at the alternate expressions after the first equality sign, one can see how each tree is constructed from smaller trees. Figure 17 shows the trees corresponding to a few selected vectors. When two trees are combined, the exponents of λ are added.

The "extra" vector $v_1 = (0.9, 0, 0, 0, 0, 0)$ has been chosen in the following way. The stars of snowflakes from Section 3 yield points $95^k(1 + o(1), o(1), o(1), o(1), o(1), o(1))\lambda^{-13k-2}$ if the vertex a is chosen as the tree root. These points converge to the vector $v_\infty := (1, 0, 0, 0, 0, 0)/\lambda \approx (0.7044, 0, 0, 0, 0, 0)$, and this vector must belong to P at least as a limit point. On the other hand, we know from by Part 1 of Theorem 1.1 that no finite tree corresponds to the point v_∞, and hence, this point will never be included in P by the algorithm. By choosing a larger rescaling v_1 of this vector, we move away from the infinitely many vectors converging to v_∞, hoping to swallow them (and possibly more points) into the convex hull, thus obtaining a smaller point set. The value 0.9 for the vector v_1 was chosen by experiment as being close to the largest value that led to convergence.

6.6 The necessity of irrational coordinates

For proving that $P \circ P \subseteq P$, we adapted the programs of Section 6.3, but the process of computation was not so straightforward and "automatic" as we had hoped. By construction, the vectors defining P are irrational. As we will now discuss, it is unavoidable to treat certain operations with these vectors as exact operations.

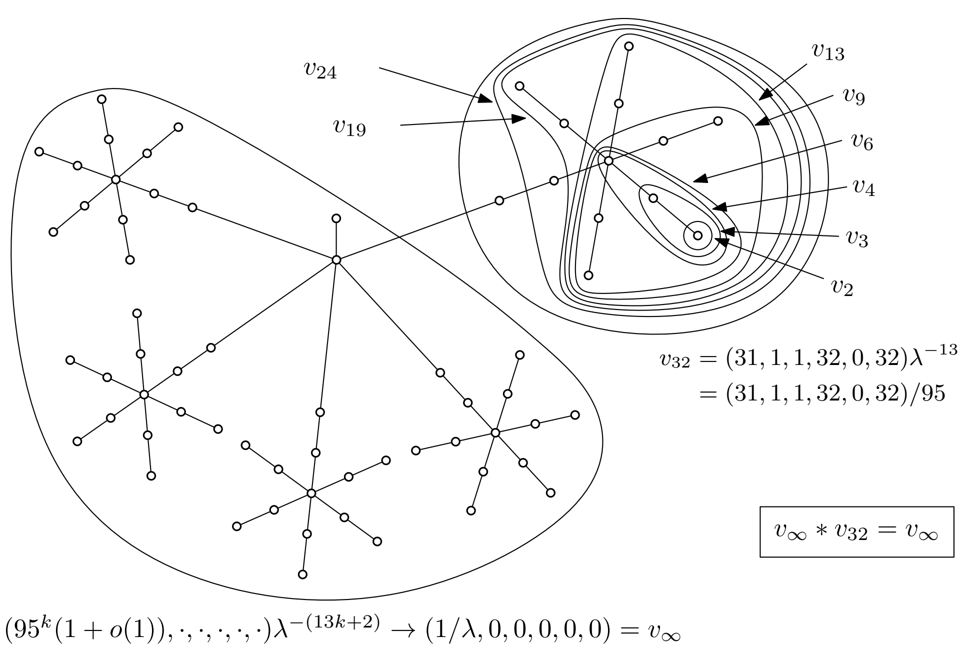


Fig. 17. Adding another snowflake to a star of $k \to \infty$ snowflakes

As illustrated in Figure 17, there is a chain of $\star$ operations, starting with the seed value v_2, and leading via $v_3, v_6, v_9, v_{13}, v_{19}, v_{24}$ to the vector $v_{32} = (31, 1, 1, 32, 0, 32)/95$, which corresponds to the snowflake rooted at one of its leaves. If these calculations were done imprecisely, then to maintain a conservative approximation, P would contain a value $\tilde{v}_{32}$ which is larger than the true value v_{32} in all non-zero components.

We shall now argue that such a value cannot exist in a bounded set P which is closed under the $\star$-operation. The reason is the relation $v_1 \star v_{32} = v_1$, which arises naturally from the definition of the stars of snowflakes: Adding another snowflake to a star of snowflakes yields a bigger star of snowflakes. In the limit, the relation expressing this composition converges to $v_\infty \star v_{32} = v_\infty$, and since v_1 is just a scaled copy of v_∞, we also have $v_1 \star v_{32} = v_1$.

Expressing this differently, the linear function $v \mapsto v \star v_{32}$ has v_1 as an eigenvector with eigenvalue 1. With the modified value, $v_1 \star \tilde{v}_{32}$ would be strictly larger than v_1 in the first component. Thus, the $\star$ operation with $\tilde{v}_{32}$ acts on v_1 like a multiplication with a factor F strictly larger than 1. The same holds true when v_1 is replaced by another non-zero vector of the form $(x, 0, 0, 0, 0, 0)$. By monotonicity, the first component of *any* vector in P (such as the vector $\tilde{v}_{32}$ itself, for instance) increases at least by the factor F when it is multiplied by $\tilde{v}_{32}$. It follows that P cannot remain bounded.

When constructing the set of vectors, we would have liked to use exact computation, but software that would perform exact linear programming with algebraic inputs was not readily available. Thus we used standard floating-point linear-programming computations to prune points of $Q \circ Q$ in the interior of the convex hull, but as we mentioned earlier, this is not reliable.

6.7 Certification of the results

To turn this computation into a proof, we extracted from the linear-programming solutions the coefficients which certified that a point is majorized by a convex combination of other points. We rounded these coefficients to multiples of 0.0001 while ensuring that their sum remains 1, and wrote them to a file. For illustration, we report in Appendix A the certifying coefficients for all products $v_9 \star v_j$, $j = 1, \ldots, 55$.

We then used a separate program to show that $v_i \star v_j \in P$ for all pairs of vertices v_i, v_j. The cases when the result is equal to another vertex of P are treated separately. The complete list of these cases is in Table 4, and they can be checked with integer arithmetic, taking out common factors of λ. The only exception is the equation $v_1 \star v_{32} = v_1$, but this can also be checked by an integer calculation since $\lambda^{-13} = 1/95$, and the comman fractional factor 0.9 on both sides can be canceled.

The remaining conditions were checked by floating-point calculations, using the stored coefficients from the file. The smallest gap occurred when showing that $v_{51} \star v_{41} \le v_{21}$. This elementwise comparison holds by a margin of 4.7×10^{-6}, which is far bigger than the accuracy of floating-point computations. The checking calculations involve only additions and multiplications of positive numbers. The largest power of λ^{-1} that occurs is 54, for computing $v_{55} \star v_{55}$, and there are just a couple of dozen more arithmetic steps before the final comparison is made for each pair i, j. Thus, errors do not accumulate over long sequences of calculations, and even single-precision floating-point calculations would be safe to use for checking this part of the proof.

The checking program is available in the source bundle of the preprint of this paper on arXiv [Rote 2019b] and on my homepage.[2] The file `minimal-dominating-sets-in-trees-docheck.py` is the main program. It consists of about 130 lines of Python code, including also the exact equality tests, and it reads data from two other files. The file `hullvertices.py` with data for the 55 vertices of P has 1774 bytes. Table 4 was generated from these data. The file `lambdas.py` with the coefficients of the 55^2 inequalities certifying that $P \circ P \subseteq P$ has 128 kBytes.

[2] http://page.mi.fu-berlin.de/rote/Papers/material/Minimal+dominating+sets+in+a+tree:+counting,+enumeration,+and+extremal+results.zip

By evaluating $\bar{M}$ for the vertices of P, one finds that the maximum, $2/\lambda^2 \approx 0.99257841$ is achieved by v_3, corresponding to the tree with two vertices. This implies $M_n \le 0.992579\lambda^n$, thus proving part 1 of Theorem 1.1.

To illustrate some of the difficulties that we encountered when trying to find a reliable proof, we finish this section with the report of two failed calculation attempts with the use of floating-point linear-programming software.

(i) As argued above, a natural point to consider as a vertex of P is the point $v_\infty = (1/\lambda, 0, 0, 0, 0, 0)$. We started the calculation by putting with v_∞ into Q instead of v_1, together with the vectors $v_2, v_3, v_6, v_9, v_{13}, v_{19}, v_{24}, v_{32}$, for which we know that they must lie on the boundary of P. The hull Q stabilized with a set of 89 vertices after a couple of minutes. However, when we tried to check and reproduce the coefficients that were extracted from the linear program with more accurate arithmetic, we failed. This setup should lead to the "correct hull" $P = \text{hull}^+(P_0)$. However, we do not even know whether this set (or rather, its topological closure) is at all a polytope with finitely many vertices. It not, this approach is doomed unless one adds artificial points like our point v_1.

(ii) For comparison, we omitted both vectors v_∞ and v_1 altogether. For this case, we know that P should theoretically grow closer and closer to v_∞ but should never reach it. However, even in this case, the program terminated after a few minutes, with a hull of 94 vertices.

7 OUTLOOK AND OPEN QUESTIONS

7.1 The growth of a bilinear operation

We have already mentioned in Section 4.3 that the bilinear operation $\star$ on sextuples captures all the necessary information of the counting question, together with the starting vector v_0 and the terminal function $\bar{M}$ from (2). Once we know these algebraic data, we can abstract from the background of the original minimal dominating sets problem: What is the largest value that can be built by combining n copies of v_0 with $n-1$ applications of the (non-associative) operation $\star$, and how fast does this value grow with n? For example, with $n = 9$ elements, we could build the expression

$$\bar{M}((v_0 \star (v_0 \star ((v_0 \star v_0) \star (v_0 \star (v_0 \star v_0))))) \star (v_0 \star v_0)).$$

When we ask the analogous question for a *linear* operation $f : \mathbb{R}^d \to \mathbb{R}^d$, this is a basic problem of linear algebra that is well-understood. The answer is given by the dominant eigenvalue of f, and the growth does not depend on the starting vector (except for degenerate cases). What happens for a general bilinear operation $\star : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$? This question is open for further study. Let us assume that the operation has nonnegative coefficients. Proposition 6.3 gives a characterization of the exponential growth rate in terms of a convex body P. Is it sufficient to consider bodies P that are polytopes? With the correct choice of λ, will the iterative process converge to a polytope? How does the growth depend on the starting vector? When is there a single "characteristic" body P that works for all starting vectors? If the growth rate always attained by a "periodic" constructions, like our star of snowflakes? Is the growth rate necessarily an algebraic number? Is it computable or approximable?

The following speculative argument tries to explain why it might be no coincidence that λ turned out to be algebraic for minimal dominating sets. Perhaps these thoughts can be strengthened generalized to show that the growth rate is always an algebraic number. In our polytope P that we used for proving the upper bound of Theorem 1.1 (Table 4), a typical vertex v has an implicit power $v = \lambda^i u$ according to how it is generated, telling how it varies in terms of λ. The tight case, when λ cannot be improved without violating the condition $P \circ P \subseteq P$, is characterized by some point $\lambda^i u$ lying on the boundary of P, i.e., in some hyperplane through some vertices $\lambda^{i_k} u_k$. This condition generates a polynomial equation in λ, and thus, λ is an algebraic number. (In our case, the critical equation is $v_1 \star v_{32} = v_1$ as explained in Section 6.6. Since v_1 was not chosen in the form $v = \lambda^i u$, the above argument is not strictly valid in this case.)

We already mentioned that in the case of linear operators, the growth is determined by the eigenvalues. Eigenvalues have been considered also for bilinear (and multilinear) operations, but the usual approach it to set up an eigenvector equation of the form $x \star x = \lambda x$ (as it would be written in our notation) and investigate the solutions and the algebraic properties of this system, see for example [Kungching et al. 2013; Breiding 2017]. Are the eigenvectors and eigenvalues in this sense related to the growth rate for our question?

Finally, it is interesting to note that some problem-specific properties that we see in trees can be written as algebraic properties of the $\star$-operation. We list a few of them.

- It is clear that the order in which subtrees are added is irrelevant. This is reflected in the following "right commutative law":
$$(u \star v) \star w = (u \star w) \star v$$
- At the level of counting minimal dominating sets, it does not matter which node is chosen as the root. This is reflected in the following partial commutativity law under the operator $\bar{M}$:
$$\bar{M}(u \star v) = \bar{M}(v \star u)$$
- Observation 1(2) says that twins are irrelevant as far as minimal dominating sets are concerned:
$$(v \star v_0) \star v_0 = v \star v_0$$
- One property that cannot be directly expressed in purely algebraic terms is the supermultiplicativity of M_n. But the main case of its proof, Observation 1(3), can be reduced to a pure calculation: It says that the combination of two trees where each root has a leaf as a neighbor will multiply the number of solutions of the two subtrees:
$$\bar{M}((v \star v_0) \star (w \star v_0)) = \bar{M}(v \star v_0) \cdot \bar{M}(w \star v_0)$$
This holds even in a stronger form than needed, as the vector equation
$$(v \star v_0) \star (w \star v_0) = v \star v_0 \cdot \bar{M}(w \star v_0).$$

All these equations can be checked computationally by substituting the definitions and expanding the terms, preferable with a computer algebra system.

7.2 Other applications of the method

Proposition 6.3 and the algorithm of Section 6.5 give a versatile method for investigating growth problems that come from dynamic-programming recursions. This extends beyond trees to other structures that can be hierarchically built up in a tree-like fashion. As a next step, one might consider 2-trees or series-parallel graphs. The combinatorial case analysis leading to the "$\star$" operations will be more complicated. For example, for series-parallel graphs, one has to monitor the status of *two* terminal vertices instead of just one root vertex, and the number of categories will multiply.

In Section 5.4, we were interested in the *minimum* number of minimal dominating sets in *trees without twins*. Here the method of Proposition 6.3 has to be adapted. We have to maintain two sets of sextuples, distinguishing whether the root has a leaf neighbor or not.

One can also count other structures than minimal dominating sets, for example *maximal irredundant subsets* of vertices. In an *irredundant* set, every vertex has a private neighbor, but the set does not have to be dominating. A different generalization is the notion of (σ, ρ)-dominating sets, where the number of neighbors in D that a vertex is allowed to have is restricted to two sets σ and ρ of natural numbers: A vertex set D is a (σ, ρ)-dominating set if for every vertex in D, the number of its neighbors in D belongs to σ, and for every vertex not in D, it belongs to ρ. This definition captures many classical graph problems. For example, induced matchings are obtained with $\sigma = \{1\}$ and $\rho = \mathbb{N}$. Matthieu Rosenfeld [2019] has recently applied our approach to compute bounds for various classes of (σ, ρ)-dominating sets (the number of all these sets, as well as the maximal and the minimal ones) in trees, forests and graphs of bounded pathwidth.

7.3 Loopless enumeration and Gray codes

In Section 5.4, we discussed the possibility to generate minimal dominating sets D faster than in linear time per solution, by counting only the operations to insert or remove an element from D. A more ambitious goal would be to enumerate the solutions with *constant delay*. Such enumeration algorithms are called *loopless* or *loop-free*, see for example [Ehrlich 1973; Knuth 2011; Herter and Rote 2018]. The sequence in which the solutions are generated has to have the property that the difference between consecutive solutions is bounded in size by a constant. Such a sequence may be called a *Gray code*, in analogy with the classical Gray code that goes through all 0-1-sequences of a given length by flipping single bits at a time.

We have already seen in Figure 7 in Section 5.4 that a Gray code is impossible without preprocessing, and we have argued that it makes sense to restrict our attention to trees without twins. Is there a Gray code through all minimal dominating sets for this class of trees? To define such a Gray code in an inductive way, one might look at Table 1, remembering its interpretation as an equation for sets, and navigate the table in a clever way.

ACKNOWLEDGMENTS

This work was initiated at the Lorentz Center workshop on "Enumeration Algorithms Using Structure" in Leiden, the Netherlands, August 24–28, 2015.

REFERENCES

Paul Breiding. 2017. *Numerical and Statistical Aspects of Tensor Decompositions.* Ph.D. Dissertation. Technische Universität Berlin. DOI: 10.14279/depositonce-6148

Jean-François Couturier, Pinar Heggernes, Pim van 't Hof, and Dieter Kratsch. 2013. Minimal dominating sets in graph classes: Combinatorial bounds and enumeration. *Theoretical Computer Science* 487 (2013), 82–94. DOI: 10.1016/j.tcs.2013.03.026

Gideon Ehrlich. 1973. Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations. *J. Assoc. Comput. Mach.* 20, 3 (July 1973), 500–513. DOI: 10.1145/321765.321781

Petr Golovach, Pinar Heggernes, Mamadou Moustapha Kanté, Dieter Kratsch, and Yngve Villanger. 2017. Minimal dominating sets in interval graphs and trees. *Discrete Applied Mathematics* 216 (Jan. 2017), 162–170. DOI: 10.1016/j.dam.2016.01.038

Felix Herter and Günter Rote. 2018. Loopless Gray code enumeration and the Tower of Bucharest. *Theoretical Computer Science* 748 (2018), 40–54. DOI: 10.1016/j.tcs.2017.11.017

Donald E. Knuth. 2011. *Combinatorial Algorithms, Part 1.* The Art of Computer Programming, Vol. 4A. Addison-Wesley.

Marcin Krzywkowski. 2013. Trees having many minimal dominating sets. *Inf. Process. Lett.* 113, 8 (April 2013), 276–279. DOI: 10.1016/j.ipl.2013.01.020

Chang Kungching, Qi Liqun, and Zhang Tan. 2013. A survey on the spectral theory of nonnegative tensors. *Numerical Linear Algebra with Applications* 20, 6 (2013), 891–912. DOI: 10.1002/nla.1902

Matthieu Rosenfeld. 2019. Bounding the number of (σ, ρ)-dominating sets in trees, forests and graphs of bounded pathwidth. (March 2019). Manuscript in preparation.

Günter Rote. 2019a. The maximum number of minimal dominating sets in a tree. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA19), San Diego*, Timothy Chan (Ed.). SIAM, 1201–1214. DOI: 10.1137/1.9781611975482.73

Günter Rote. 2019b. Minimal Dominating Sets in a Tree: Counting, Enumeration, and Extremal Results. (March 2019). arXiv:1903.04517

A CERTIFYING COMPUTATIONS FOR $v_9 \star v_j$

For illustration, we show a section of the data that are used in the proof of the closure property (21) of the polytope P in Section 6.7. Such data exist for each product $v_i \star v_j$, $1 \le i, j \le 55$. The coefficients stand for exact four-digit decimal numbers, which add up to 1 on each line.

$v_9 \star v_1 \le v_{13}$

$v_9 \star v_2 \le v_3$

$v_9 \star v_3 = v_{12}$

$v_9 \star v_4 \le 0.0158\, v_1 + 0.1798\, v_5 + 0.4284\, v_3 + 0.3760\, v_{13}$

$v_9 \star v_5 = v_{19}$

$v_9 \star v_6 \le v_{10}$

$v_9 \star v_7 \le 0.1454\, v_1 + 0.1066\, v_5 + 0.2046\, v_3 + 0.5434\, v_{13}$

$v_9 \star v_8 = v_{27}$

$v_9 \star v_9 \le v_{10}$

$v_9 \star v_{10} \le 0.0823\, v_1 + 0.1504\, v_5 + 0.0609\, v_3 + 0.7064\, v_{13}$

$v_9 \star v_{11} \le 0.2096\, v_1 + 0.0732\, v_5 + 0.0922\, v_3 + 0.6250\, v_{13}$

$v_9 \star v_{12} = v_{37}$

$v_9 \star v_{13} \le v_{10}$

$v_9 \star v_{14} \le 0.0518\, v_1 + 0.1396\, v_5 + 0.7853\, v_{13} + 0.0233\, v_8$

$v_9 \star v_{15} \le 0.1127\, v_1 + 0.0979\, v_5 + 0.0110\, v_3 + 0.7784\, v_{13}$

$v_9 \star v_{16} \le 0.2412\, v_1 + 0.0595\, v_5 + 0.0351\, v_3 + 0.6642\, v_{13}$

$v_9 \star v_{17} = v_{43}$

$v_9 \star v_{18} \le 0.0171\, v_{20} + 0.2959\, v_3 + 0.3547\, v_{28} + 0.0909\, v_5 + 0.2414\, v_{27}$

$v_9 \star v_{19} \le v_{10}$

$v_9 \star v_{20} \le 0.0373\, v_1 + 0.0420\, v_5 + 0.8224\, v_{13} + 0.0983\, v_8$

$v_9 \star v_{21} \le 0.0631\, v_1 + 0.0422\, v_9 + 0.0367\, v_5 + 0.8125\, v_{13} + 0.0455\, v_8$

$v_9 \star v_{22} \le 0.1238\, v_1 + 0.0795\, v_9 + 0.0593\, v_5 + 0.7373\, v_{13} + 0.0001\, v_8$

$v_9 \star v_{23} \le 0.2567\, v_1 + 0.0556\, v_5 + 0.0054\, v_3 + 0.6823\, v_{13}$

$v_9 \star v_{24} = v_{48}$

$v_9 \star v_{25} \le 0.0911\, v_5 + 0.3149\, v_3 + 0.3847\, v_{28} + 0.1192\, v_{37} + 0.0901\, v_{27}$

$v_9 \star v_{26} \le v_{10}$

$v_9 \star v_{27} \le 0.0310\, v_1 + 0.0150\, v_{12} + 0.8388\, v_{13} + 0.1152\, v_8$

$v_9 \star v_{28} \le 0.0393\, v_1 + 0.0632\, v_9 + 0.0222\, v_{12} + 0.8270\, v_{13} + 0.0483\, v_8$

$v_9 \star v_{29} \le 0.0650\, v_1 + 0.1454\, v_9 + 0.0157\, v_5 + 0.7462\, v_{13} + 0.0277\, v_8$

$v_9 \star v_{30} \le 0.1296\, v_1 + 0.1188\, v_9 + 0.0226\, v_5 + 0.7151\, v_{13} + 0.0139\, v_8$

$v_9 \star v_{31} \le 0.0654\, v_1 + 0.1382\, v_9 + 0.2283\, v_3 + 0.0405\, v_5 + 0.5276\, v_{13}$

$v_9 \star v_{32} \le 0.2643\, v_1 + 0.0243\, v_5 + 0.6898\, v_{13} + 0.0216\, v_8$

$v_9 \star v_{33} \le 0.9543\, v_{49} + 0.0188\, v_3 + 0.0269\, v_{45}$

$v_9 \star v_{34} \le 0.0942\, v_5 + 0.3247\, v_3 + 0.3909\, v_{28} + 0.1772\, v_{37} + 0.0130\, v_{27}$

$v_9 \star v_{35} \le v_{10}$

$v_9 \star v_{36} \le 0.0284\, v_1 + 0.0667\, v_{12} + 0.8450\, v_{13} + 0.0599\, v_8$

$v_9 \star v_{37} \le 0.0277\, v_1 + 0.0813\, v_9 + 0.0661\, v_{12} + 0.8249\, v_{13}$

$v_9 \star v_{38} \le 0.0365\, v_1 + 0.1762\, v_9 + 0.0151\, v_{12} + 0.7502\, v_{13} + 0.0220\, v_8$

$v_9 \star v_{39} \le 0.0665\, v_1 + 0.1946\, v_9 + 0.0036\, v_5 + 0.7132\, v_{13} + 0.0221\, v_8$

$v_9 \star v_{40} \le 0.1329\, v_1 + 0.1365\, v_9 + 0.0028\, v_5 + 0.7039\, v_{13} + 0.0239\, v_8$

$v_9 \star v_{41} \le 0.0279\, v_1 + 0.0944\, v_{12} + 0.8461\, v_{13} + 0.0316\, v_8$

$v_9 \star v_{42} \le 0.0210\, v_1 + 0.1225\, v_9 + 0.0652\, v_{12} + 0.7913\, v_{13}$

$v_9 \star v_{43} \le 0.0229\, v_1 + 0.1975\, v_9 + 0.0354\, v_{12} + 0.7442\, v_{13}$

$v_9 \star v_{44} \le 0.0359\, v_1 + 0.2301\, v_9 + 0.0148\, v_{12} + 0.7119\, v_{13} + 0.0073\, v_8$

$v_9 \star v_{45} \le 0.0679\, v_1 + 0.2169\, v_9 + 0.0080\, v_{12} + 0.6966\, v_{13} + 0.0106\, v_8$

$v_9 \star v_{46} \le 0.0185\, v_1 + 0.1434\, v_9 + 0.0661\, v_{12} + 0.7720\, v_{13}$

$v_9 \star v_{47} \le 0.0162\, v_1 + 0.2226\, v_9 + 0.0360\, v_{12} + 0.7252\, v_{13}$

$v_9 \star v_{48} \le 0.0213\, v_1 + 0.2550\, v_9 + 0.0218\, v_{12} + 0.7019\, v_{13}$

$v_9 \star v_{49} \le 0.0363\, v_1 + 0.2568\, v_9 + 0.0163\, v_{12} + 0.6906\, v_{13}$

$v_9 \star v_{50} \le 0.0135\, v_1 + 0.2352\, v_9 + 0.0379\, v_{12} + 0.7134\, v_{13}$

$v_9 \star v_{51} \le 0.0144\, v_1 + 0.2721\, v_9 + 0.0232\, v_{12} + 0.6903\, v_{13}$

$v_9 \star v_{52} \le 0.0211\, v_1 + 0.2834\, v_9 + 0.0167\, v_{12} + 0.6788\, v_{13}$

$v_9 \star v_{53} \leq 0.3716\, v_{20} + 0.3132\, v_{28} + 0.2973\, v_{21} + 0.0179\, v_{24}$
$v_9 \star v_{54} \leq 0.0144\, v_1 + 0.2965\, v_9 + 0.0184\, v_{12} + 0.6707\, v_{13}$
$v_9 \star v_{55} \leq 0.3078\, v_{20} + 0.3709\, v_{28} + 0.3010\, v_{21} + 0.0203\, v_{24}$

B ANOTHER ENUMERATION ALGORITHM: ENUM3

We present another variation of an algorithm for enumerating minimal dominating sets through the expression DAG. It combines the positive features of Algorithms ENUM1 and ENUM2. In the outer loop of product nodes, subtrees where nothing changes are not visited, potentially saving a lot of work. In this respect, we follow ENUM1. Like ENUM2, the end of a loop is signaled simultaneously with the delivery of the last solution. Thus, the dummy visits of ENUM1 are avoided. Unlike ENUM2, we also distinguish the first element of a loop with a special message.

The algorithm is shown in Figures 18 and 19. Like Algorithm ENUM2 in Section 5.3, this is a low-level description without generators or coroutines. All message passing is explicit. However, the algorithm is presented in a different style from ENUM2: Instead of a family of patterns like Figures 8–11, the algorithm is written more conventionally as a series of nested case distinctions. Certain operations that have been left out in Section 5.3 are explicitly stated, for example, remembering the child of a product node that is currently visited (or recognizing it when a message is received from it). This changed style reflects the author's insecurity about the best way to present such enumeration algorithms.

We shall now discuss some details. Messages are sent across the arcs of the expression DAG. There are two types of *request messages*: PRODUCE-FIRST and PRODUCE-NEXT. They always flow downward in the network, from the root towards the leaves. There are two types of *reply messages*: DONE and LAST. They always flow upward in the network.

Every union and product node has a *state* attribute from a small choice of possibilities. In addition, every product node records which of its children has received a message in its *child* attribute. As in the algorithms of Section 5, we have an additional *master node* with a single outgoing arc to the target node. Its only job is to send PRODUCE-NEXT requests until it receives a LAST message that signals completion of the enumeration.

The current node is denoted by a global variable K. Depending on the type of node and on the message received, the program may consult the *child* or *state* attributes of K. It will then possibly update the attributes, and move to an adjacent node with a new message, which is stored in the global variable *message*. The solution D is maintained as another global variable.

As in Algorithm ENUM2 in Section 5.5, we explore various subtrees of the expression DAG in a depth-first search manner, and we maintain a "call stack" of nodes that are still expecting a reply. In the program, "go to node K'" means: push the current node K on the stack, and set $K := K'$, while "go to the parent" means: pop K from the stack.

The algorithm carries out very simple operations, but it is not apparent what happens. We will discover some structure by describing the process from multiple views: from a single arc and then from a single node.

Message flow along an arc. The flow of messages along an arc is a strict alternation:

→ request(PRODUCE-FIRST)
← reply(DONE)
→ request(PRODUCE-NEXT)
← reply(DONE)
. . .
→ request(PRODUCE-NEXT)
← reply(LAST)

```
Let K be the master node.
message := PRODUCE-FIRST, go to the target node, and start the following loop.
loop
  let K be the current node
  case K is a basis node for vertex a:
        case K represents the set {a}:
              insert vertex a into D if it is not already in D
        case K represents the set ∅:
              remove vertex a from D if it is in D
        message := LAST, and go to the parent
  case K is the master node:
        report the current solution D
        case message = DONE:
              message := PRODUCE-NEXT, and go to the target node
        case message = LAST:
              exit from the loop and stop
  case K is a union node:
        case message = PRODUCE-FIRST:
              K.state := "child 1"
              message := PRODUCE-FIRST, and go to the first child
        case message = PRODUCE-NEXT:
              case K.state = "child 1":
                    message := PRODUCE-NEXT, and go to the first child
              case K.state = "transition from child 1 to child 2":
                    K.state := "child 2"
                    message := PRODUCE-FIRST, and go to the second child
              case K.state = "child 2":
                    message := PRODUCE-NEXT, and go to the second child
        case message = DONE:
              message := DONE, and go to the parent
        case message = LAST:
              case K.state = "child 1":
                    K.state := "transition from child 1 to child 2"
                    message := DONE, and go to the parent
              case K.state = "child 2":
                    K.state := "dormant"
                    message := LAST, and go to the parent
  case K is a product node:
        handle K by the algorithm in Figure 19
```

Fig. 18. Algorithm ENUM3

A reply message signals that a solution has been set up in the vertices of the subtree associated to the child. If no more solutions are available after the current one, this is signaled by the LAST reply.

Since we have ensured that every node represents a nonempty set of solutions, the PRODUCE-FIRST request will always produce a reply. Thus, the minimum total number of messages is two.

```
case K is a product node:
    case message = PRODUCE-FIRST:
        K.state := "working"
        K.child := 1
        message := PRODUCE-FIRST, and go to the first child
    case message = PRODUCE-NEXT:
        case K.state = "working" or K.state = "child 1 has finished":
            K.child := 2
            message := PRODUCE-NEXT, and go to the second child
        case K.state = "child 2 has finished":
            K.state := "working"
            K.child := 1
            message := PRODUCE-NEXT, and go to the first child
    case K.child = 1 and message = DONE:
        K.child := 2
        message := PRODUCE-FIRST, and go to the second child
    case K.child = 1 and message = LAST:
        K.state := "child 1 has finished"
        K.child := 2
        message := PRODUCE-FIRST, and go to the second child
    case K.child = 2 and message = DONE:
        message := DONE, and go to the parent
    case K.child = 2 and message = LAST:
        case K.state = "working":
            K.state := "child 2 has finished"
            message := DONE, and go to the parent
        case K.state = "child 1 has finished":
            K.state := "dormant"
            message := LAST, and go to the parent
```

Fig. 19. Algorithm ENUM3: Handling of a product node

After a block is finished with a LAST reply, a new block of messages can be initiated with another PRODUCE-FIRST message.

In contrast to the algorithm ENUM2 of Section 5.3, there is a special PRODUCE-FIRST request to initiate the dialogue. This allows the node to know when it needs to initialize itself. It also has the nice feature that it makes the message exchange symmetric with respect to time reversal.

When we now analyse the flow from the point of view of the different types of nodes, we will inductively assume that the message exchange with the children (if any) follows the pattern described above, and we will follow the operation of the node from the initial PRODUCE-FIRST request received from the parent to the final LAST reply. The *state* of all union and product nodes is initialized to "dormant", indicating that they are ready to receive a PRODUCE-FIRST message and start producing results. Actually, the "dormant" state has only informational value without effect for the algorithm.

Basis nodes. The basis nodes return immediately with a LAST message after setting up the solution D by inserting a vertex into D or removing it from D.

message from/to parent	child	message from/to child	*state*
			dormant
PRODUCE-FIRST →	1	→ PRODUCE-FIRST	
			child 1
DONE ←	1	← DONE	
			child 1
PRODUCE-NEXT →	1	→ PRODUCE-NEXT	
			child 1
DONE ←	1	← DONE	
			child 1
			…
			child 1
PRODUCE-NEXT →	1	→ PRODUCE-NEXT	
			child 1
DONE ←	1	← LAST	
			transition from child 1 to child 2
PRODUCE-NEXT →	2	→ PRODUCE-FIRST	
			child 2
DONE ←	2	← DONE	
			child 2
			…
			child 2
PRODUCE-NEXT →	2	→ PRODUCE-NEXT	
			child 2
LAST ←	2	← LAST	
			dormant

Fig. 20. The message flow from the viewpoint of a union node. In each line, the node receives a message from its parent and sends a message to one of its children, or vice verse. The number of the involved child is indicated in the second column.

Union nodes. The message flow of a union node is shown in Figure 20, and it is easy to understand. When receiving a PRODUCE message from its parent, the union node K will enter exactly one of its two children. Upon returning from a child, control will pass back to the parent of K. It is evident that K performs two successive loops over its children.

Product nodes. The message flow of a product node K is shown in Figure 21. The attribute $K.child$ always stores the number of the child that was entered from K. The default state is "working". If any child has recently sent the LAST message, this is recorded as the state "child 1 has finished" or "child 2 has finished". One can see that K implements a nested loop.

When receiving a PRODUCE message from its parent, K will enter the second child or both children before passing control back to the parent. The first child will only be visited on the first activation from the parent with the message PRODUCE-FIRST, or after the inner loop (of the second child) has been exhausted on the previous visit, which is indicated by the state "child 2 has finished". After the visiting the first child, the loop over the second child will be initialized with a PRODUCE-FIRST message.

We might as well have started from the desired behavior in Figures 20 and 21 and synthesized the program and the necessary states of the *state* variable from these diagrams.

The analysis of the algorithm is a straightforward modification of the analysis in Section 5. Recall that we defined a well-structured enumeration tree as a subtree E of the expression DAG that contains both children of every product node in E and exactly one child of every union node in E. A *partial well-structured enumeration tree* is defined similarly, except that a product node may also have just one child in E.

Proposition B.1. *If a node K receives a request from a parent, Algorithm ENUM3 will visit the nodes of partial well-structured enumeration tree with root K before replying to the parent.* □

The set of visited nodes is actually the same as those nodes that are visited by a proper visit in Algorithm ENUM1.

message from/to parent	*child*	message from/to child	*state*
			dormant
PRODUCE-FIRST →	1	→ PRODUCE-FIRST	
			working
	1	← DONE	
	2	→ PRODUCE-FIRST	
			working
DONE ←	2	← DONE	
			working
PRODUCE-NEXT →	2	→ PRODUCE-NEXT	
			working
DONE ←	2	← DONE	
			working
			...
PRODUCE-NEXT →	2	→ PRODUCE-NEXT	
			working
DONE ←	2	← LAST	
			child 2 has finished
PRODUCE-NEXT →	1	→ PRODUCE-NEXT	
			working
	1	← DONE	
	2	→ PRODUCE-FIRST	
			working
DONE ←	2	← DONE	
			working
			...
PRODUCE-NEXT →	2	→ PRODUCE-NEXT	
			working
DONE ←	2	← LAST	
			child 2 has finished
PRODUCE-NEXT →	1	→ PRODUCE-NEXT	
			working
	1	← LAST	
	2	→ PRODUCE-FIRST	
			child 1 has finished
DONE ←	2	← DONE	
			child 1 has finished
PRODUCE-NEXT →	2	→ PRODUCE-NEXT	
			child 1 has finished
DONE ←	2	← DONE	
			...
PRODUCE-NEXT →	2	→ PRODUCE-NEXT	
			child 1 has finished
LAST ←	2	← LAST	
			dormant

Fig. 21. The message flow from the viewpoint of a product node. Each inner loop over child 2 is grouped by a bracket. In this example, there are three iterations of the outer loop. As in Figure 20, each line represents one operation of the node under consideration, except when a received message from a child results in a message being sent to another child: then the operation appears on two consecutive lines. The *child* attribute in the second column identifies also the number of the child with whom the message exchange takes place.

A partial well-structured enumeration tree can easily be extended into a (complete) well-structured enumeration tree. Therefore, by Lemma 5.1(3), a partial well-structured enumeration tree whose root is associated to the vertex set A contains $O(|A|)$ nodes in total. We conclude:

Theorem B.2. *Algorithm ENUM3 enumerates the minimal dominating sets of a tree with linear delay, after linear setup time. After the last solution, the algorithm terminates in constant time.* □

C OVERVIEW OF NOTATIONS

- T = a tree $T = (V, E)$
- a graph $G = (V, E)$
- $n = |V|$ = number of vertices
- $D \subseteq V$ a dominating set
- $A \subseteq V$ a subtree
- **G**ood. ≠ graph G
- **S**elf
- **L**acking
- **d**ominated
- **p**rivate
- **f**ree
- subtrees A_1, A_2, B combined into a tree C
- vector $v = (G, S, L, d, p, f)$
- $\bar{M}(G, S, L, d, p, f) = G + S + d + p$ = #MDS
- with root r, and s
- special vertices a and b in the star of snowflakes
- general vertices a and b
- total number $M(T)$
- k = number of snowflakes
- RT_{13k+1} record trees
- M_n = max # MDS
- $\mathcal{V}_n$ = set of 6-vectors for trees of size n
- v_i = individual 6-vectors, vertices of P
- $v_0 = (0, 1, 0, 0, 0, 1)$, starting vector
- $\preceq, \succeq$ majorization
- $v \star v'$ for individual vectors, w, w'
- $V \circ V'$ for sets of vectors
- P, Q sets of vectors, P "polytope", Q discrete set
- λ, λ^* = growth rate
- μ_i, ν_j coefficients for convex combination
- hull(P) majorized hull
- # hull(P) number of its generating vertices = nonmajorized vertices (used only once)
- $\mathrm{hull}^+(P)$ majorized convex hull
- # $\mathrm{hull}^+(P)$ number of its extreme vertices number (used only once)
- $\mathbf{X} = \mathbf{X}(T)$ Expression Dag
- K, K', K_2, K_2 nodes in the expression DAG, also in the context of the program, as a record or *object*
- $R(K), R(K_1) \subseteq 2^V$ = the node subsets *represented* by K
- k iterations in a generator loop
- C_1, C_2 number of solutions represented by child 1/2
- t_1, t_2, t, t' average time for enumeration
- $k = a \log_2 n$ number of stars in the chain of star clusters example
- E subgraph of visited nodes, well-structured enumeration tree
- p number of visited product nodes