ALGORITHMS FOR VERTEX PARTITIONING PROBLEMS ON PARTIAL K-TREES [*]

JAN ARNE TELLE [†] AND ANDRZEJ PROSKUROWSKI [‡]

Abstract.
In this paper, we consider a large class of vertex partitioning problems and apply to those the theory of algorithm design for problems restricted to partial k-trees. We carefully describe the details of algorithms and analyze their complexity in an attempt to make the algorithms feasible as solutions for practical applications.

We give a precise characterization of vertex partitioning problems, which include domination, coloring and packing problems and their variants. Several new graph parameters are introduced as generalizations of classical parameters. This characterization provides a basis for a taxonomy of a large class of problems, facilitating their common algorithmic treatment and allowing their uniform complexity classification.

We present a design methodology of practical solution algorithms for generally $\mathcal{NP}$-hard problems when restricted to partial k-trees (graphs with treewidth bounded by k). This "practicality" accounts for dependency on the parameter k of the computational complexity of the resulting algorithms.

By adapting the algorithm design methodology on partial k-trees to vertex partitioning problems, we obtain the first algorithms for these problems with reasonable time complexity as a function of treewidth. As an application of the methodology, we give the first polynomial-time algorithm on partial k-trees for computation of the Grundy Number.

1. Introduction.

Many inherently difficult ($\mathcal{NP}$-hard) optimization problems on graphs become tractable when restricted to trees, or to graphs with some kind of tree-like structure. A large class of such graphs is the class of partial k-trees (equivalently, graphs with treewidth bounded by k). Allthough tractability requires fixed k, this class contains all graphs with n vertices when the parameter k is allowed to vary through positive integers up to $n-1$. Many natural classes of graphs have bounded treewidth [21]. There are many approaches to finding a template for the design of algorithms on partial k-trees with time complexity polynomial, or even linear, in the number of vertices [23, 1]. Proponents of these approaches attempt to encompass as wide a class of problems as possible, often at the expense of simplicity of the resulting algorithms and also at the expense of increased algorithm time complexity as a function of k. In contrast, results giving explicit practical algorithms in this setting are usually limited to a few selected problems on either (full) k-trees [9], partial 1-trees or partial 2-trees [25]. We intend to cover the middle ground between these two extremes, by investigating the time complexity as a function of both input size and the treewidth k.

We assume that the input graph is given with a width k tree-decomposition, computable in linear time for fixed k [6]. Our algorithms employ a *binary parse tree* of the input partial k-tree, easily derived from a tree-decomposition of the graph. This parse tree is based on very simple graph operations that mimic the construction process of an embedding k-tree. We propose a design methodology that for many $\mathcal{NP}$-hard problems results in algorithms with time complexity linear in the size of the input graph and only exponential in its treewidth, lowering the exponent of previously known solutions. We give a careful description of the algorithm design details with the aim of easing the task of implementation for practical applications. We include a brief report on an ongoing implementation project.

A large class of inherently difficult discrete optimization problems can be expressed in the *vertex partitioning* formalism. This formalism involves neighborhood constraints on vertices in different

[*]This work forms part of the first author's PhD thesis at the University of Oregon and was supported in part by NSF grant CCR9213439.

[†]Department of Informatics, University of Bergen, 5020 Bergen, Norway. telle@ii.uib.no. Supported by a grant from the Norwegian Research Council.

[‡]Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, USA. andrzej@cs.uoregon.edu.

classes (blocks) of a partition and provides a basis for a taxonomy of vertex partitioning problems. We define this formalism and then use it to provide a uniform algorithmic treatment on partial k-trees of vertex partitioning problems. As an example of application of our paradigm, we give the first polynomial-time algorithms on partial k-trees for the Grundy Number. The efficiency of our algorithm follows from (i) the description of the Grundy Number problem as a vertex partitioning problem, (ii) a careful investigation of time complexity of vertex partitioning problems on partial k-trees, and (iii) a new logarithmic bound on the Grundy Number of a partial k-tree.

We present these ideas as follows: in section 3, we describe the binary parse tree of partial k-trees and the general algorithm design method, in section 4 we define vertex partitioning problems, in section 5 we apply the partial k-tree algorithm design method to vertex partitioning problems, and in section 6 we give the efficient solution algorithm for the Grundy Number on partial k-trees. We conclude the paper with a brief report on experiences with implementations.

2. Definitions. We denote the non-negative integers by $\mathbb{N}$ and the positive integers by $\mathbb{P}$. The graph $G = (V(G), E(G))$ has vertex set $V(G)$ and edge set $E(G)$. We consider simple, undirected graphs, unless otherwise specified. For $S \subseteq V(G)$ let $G[S] = (S, \{(u,v) : u,v \in S \wedge (u,v) \in E(G)\})$ denote the subgraph *induced* in G by S. For $S \subseteq V(G)$ let $G \setminus S = G[V(G) \setminus S]$. A *component* in a graph is a maximal connected subgraph. A *separator* of a graph G is a subset of vertices $S \subseteq V(G)$ such that $G \setminus S$ has more components than G. In a *complete* graph there is an edge for every two-element subset of vertices.

A graph G is a *k-tree* if it is a complete graph on k vertices (a *k-clique*) or if it has a vertex $v \in V(G)$ whose neighbors induce a k-clique of size k such that $G \setminus \{v\}$ is again a k-tree. Such a *reduction process* of G (or the corresponding *construction* process) determines its *parse tree.* A *partial* k-tree H is a subgraph of a k-tree and a construction process of this embedding k-tree defines a parse tree of H. A *tree-decomposition* of a graph G is a tree T whose nodes are subsets of vertices of G such that for every edge (u,v) of G, there is a node containing both u and v, and for every vertex u of G, the set of nodes of T that contain u induces a (non-empty, connected) subtree of T. The nodes of T are often called *bags.* The *width* of a tree-decomposition T is defined as one less than the maximum size of a bag. The *treewidth* of G is the minimum width of a tree-decomposition of G. It is fairly easy to see that a parse tree of a partial k-tree G defines (through maximal cliques of G) a width k tree-decomposition of G. Similarly, based on such a decomposition one can find a k-tree embedding G. For any partial k-tree G with at least k vertices there is a k-tree H with the same number of vertices for which G is a subgraph. The fact that we can assume vertex sets equality follows from the treewidth formulation.

A linear ordering $\pi = v_1, ..., v_n$ of the vertices of a graph is a *perfect elimination ordering (peo)* if for each i, $1 \leq i \leq n$, the higher-numbered neighbors of v_i induce a clique. A k-tree H has a peo $\pi = v_1, ..., v_n$ such that for each i, $1 \leq i \leq n-k$, the vertex set $B_i = \{v_i\} \cup (N_H(v_i) \cap \{v_{i+1}, ..., v_n\})$ induces a $k+1$-clique in H. The set $B_i \setminus \{v_i\}$ is a minimal separator of the graph H. See Figure 1 for an example of a partial 3-tree *embedded* in a 3-tree. In analogy with the role $(k+1)$-cliques of H play in a width k tree-decomposition of H we call $B_i, 1 \leq i \leq n-k$, a $(k+1)$-bag in G under π and each of its k-vertex subsets is similarly called a k-bag of G under π. The remaining definitions in this and the following sections are all for given graphs G, H, a peo $\pi = v_1, ..., v_n$ and bags B_i as above. We first define a *peo-tree* P of G: The peo-tree P of G based on π is a rooted tree with nodes $V(P) = \{B_1, ..., B_{n-k}\}$. The node B_{n-k} is the root of P; a node B_i, $1 \leq i < n-k$, has as its parent in P the node B_j, $i < j \leq n-k$, such that j is the minimum bag index with $|B_i \cap B_j| = k$ (note that this intersection does not contain v_i). The peo-tree P is a clique tree of H and also a width k tree-decomposition of both G and H (since $B_i \cap B_j$ is a separator of G). See Figure 1 for an example of a peo-tree.

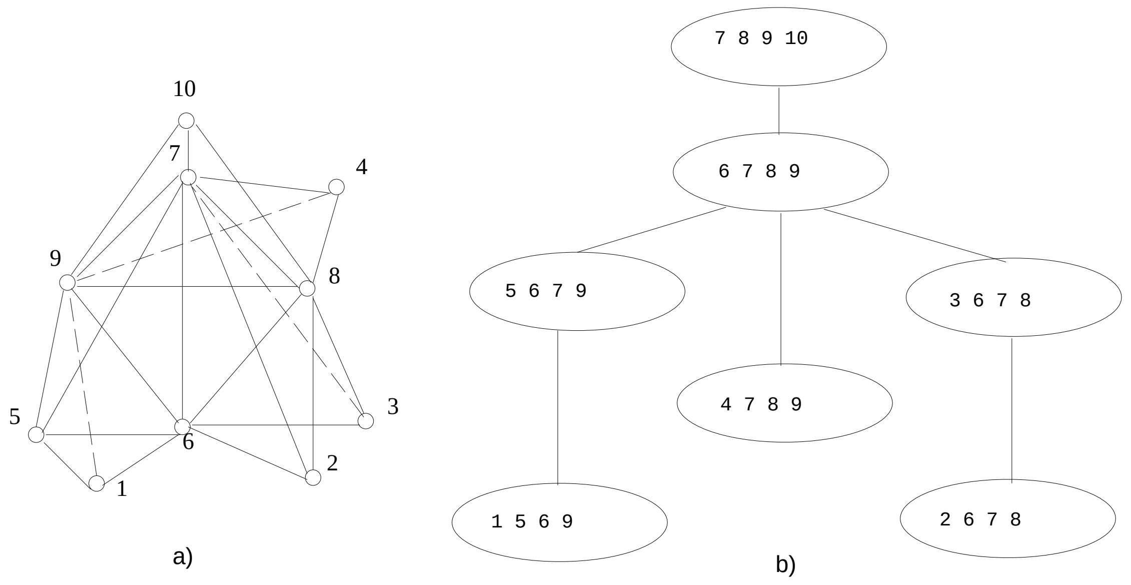

FIG. 1. *a) A partial 3-tree G, embedded in a 3-tree H, dashed edges in $E(H) - E(G)$. b) Its peo-tree P wrt. peo=1,2,3,4,5,6,7,8,9,10.*

3. Practical Algorithms on Partial k-Trees. Many $\mathcal{NP}$-hard problems on graphs, when restricted to partial k-trees, for fixed values of k, have solution algorithms that execute in polynomial, or even linear time as a function of input graph size. In this section, we improve on the practicality of such algorithms, both in terms of their complexity and their derivation, by accounting for dependency on the treewidth k. Since each such algorithm is designed for fixed k, we consider a class of algorithms parameterized by k. We first define a binary parse tree of partial k-trees that is based on very simple graph operations. Then, we discuss the derivation and complexity analysis of dynamic programming solution algorithms which follow this parse tree.

3.1. Binary Parse Tree. Based on the peo-tree of a partial k-tree as defined above, we construct a *binary parse tree*. We first introduce an algebra of *i-sourced* graphs. Terms in this algebra will evaluate to partial k-trees and their expression trees will be the *binary parse trees* of the resulting graphs.

Let a graph with i distinguished vertices (also called sources) have type G_i. We define the following graph operations:

- *Primitive*: $\rightarrow G_{k+1}$. This 0-ary operation introduces the graph $G[B]$, for some $(k+1)$-bag B.
- *Reduce*: $G_{k+1} \rightarrow G_k$. This unary operation eliminates a source designation of the $(k+1)$-st source vertex, leaving the graph otherwise unchanged.
- *Join*: $G_{k+1} \times G_k \rightarrow G_{k+1}$. This binary operation takes the union of its two argument graphs (say, A and B), where the sources of the second graph (a k-bag S_B) are a subset of the sources of the first graph (a $(k+1)$-bag S_A); these are the only shared vertices, and adjacencies for shared vertices are the same in both graphs. In other words, $V(A) \cap V(B) = S_B \subseteq S_A$ and $E(A[S_B]) = E(B[S_B])$, giving the resulting graph $Join(A, B) = (V(A) \cup V(B), E(A) \cup E(B))$ with sources S_A.
- *Forget*: $G_{k+1} \rightarrow G_0$. This operation eliminates the source designation of all source vertices.

The above definitions imply that in a term of the sourced graphs algebra that evaluates to a graph G, the source sets are $(k+1)$-bags and k-bags in a width k tree-decomposition of G. A *binary parse tree* of a graph G is the expression tree of such a term.

We show how to construct a binary parse tree from a peo-tree. Intuitively, each node of the peo-tree is "stretched" into a leaf-towards-root path of the binary parse tree. Let P be a peo-tree of a partial k-tree G under a peo π. For a node B_i of P, $1 \leq i \leq n-k$, with c children, define a path starting in a *Primitive* node evaluating to $G[B_i]$, with c *Join* nodes as interior vertices (one for each child of B_i), and ending in a *Reduce* node which drops the source designation of v_i. From the resulting collection of $|V(P)|$ Primitive-Join*-Reduce paths (note the total number of Join nodes is $|E(P)| = |V(P)| - 1$) we construct the binary parse tree by assigning Reduce nodes as children of the appropriate Join nodes. The only exception is the Reduce node associated with the root of P, which becomes the child of a new Forget node, the root of the resulting binary parse tree. The Reduce node associated with a node B_i of P with parent $parent(B_i)$ becomes the child of a Join node on the path associated with $parent(B_i)$. These assignments are easily done so that each Join node has a unique Reduce node as a child. Note that we have the freedom of choosing the order in which the children of a given node in P are *Join*ed. This freedom, and also a possible choice of π, can be exploited to keep the resulting parse tree shallow, an important attribute in the design of parallel algorithms for partial k-trees. See Figure 2 for an example of a binary parse tree; note the $|V(P)|$ paths from leaves to their Reduce ancestors.

THEOREM 3.1. *Given a peo-tree P of a partial k-tree G, the graph algebra term that corresponds to the constructed binary parse tree T evaluates to G.*

Proof. The constructed tree T is the expression tree of a well-formed term in the given algebra, since Primitive nodes are exactly its leaves, and children of other nodes have the right types. Primitive nodes contain all edges of G, as they represent all subgraphs induced by $(k+1)$-bags of G. For each node B_i of P, the Reduce operation associated with it merely drops the source designation of v_i. Thus, we need only show that the Join operations act correctly on their argument graphs by identifying their sources. The Join operations are in a natural one-to-one correspondence with the edges of the peo-tree P, a tree-decomposition of G, where identification of vertices is done simply by taking the union of the two bags at endpoints of the edge. Let a Join operation $Join(X, Y)$ correspond in this way to the edge between a node B_i of P and its parent B_j. We have $|B_i \cap B_j| = k$ with $B_i \setminus B_j = \{v_i\}$. By structural induction on T, we assume that subtrees representing X (of type G_{k+1}) and Y (of type G_k) have correctly identified vertices of G, so that the sources of X and Y are B_j and $B_i \setminus \{v_i\}$, respectively. The operation $Join(X, Y)$ identifies exactly the vertices $B_i \cap B_j$, and the resulting subtree rooted at this node has sources B_j. The Forget node at the root of T drops all source designations, so the graph algebra term that corresponds to the constructed binary parse tree T evaluates to G. □

We say that T *represents* G. Since P is a peo-tree with $n-k$ nodes, the binary parse tree T of G derived from P has $n-k$ Primitive leaves and $n-k$ Reduce nodes, one for each node of P, it has $n-k-1$ Join nodes, one for each edge of P, and a single Forget node at the root.

3.2. Complexity Analysis Accounting for Treewidth. The following algorithm design methodology is an adaptation to the binary parse tree of the earlier paradigm of [3]. A dynamic programming solution algorithm for a problem on a partial k-tree G will follow a bottom-up traversal of the binary parse tree T. As usual, with each node u of T we associate a data structure *table*. Each index of these tables represents a different constrained version of the problem. The corresponing entry of a table associated with a node u of T characterizes the optimal solutions to the constrained subproblem restricted to G_u, the sourced subgraph of G represented by the subtree of T rooted at u. The table of a leaf is initialized according to the base case, usually by a brute-force strategy. The

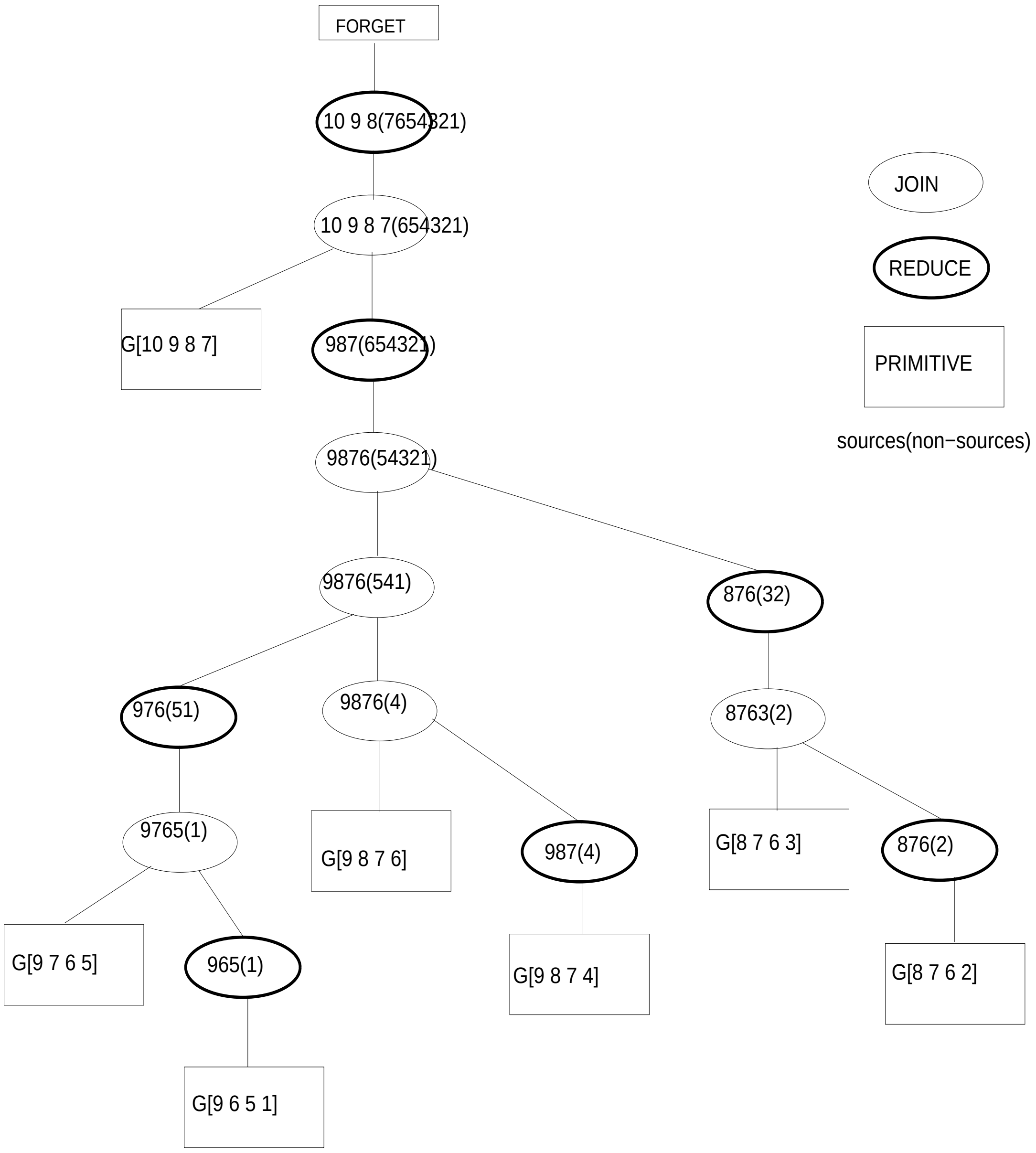

FIG. 2. *The binary parse tree T of the partial 3-tree G based on the peo-tree P (see Figure 1). Nodes $u \in V(T)$ labeled by $V(G_u)$ with non-sources in parenthesis.*

table of an interior node is computed in a bottom-up traversal of T according to the tables of its children. The overall solution is obtained from the table at the root of T.

The paradigm for designing such algorithms is especially attractive for the class of *vertex state* problems. For a vertex state problem, we define a set of *vertex states*, that represent the different ways that a solution to a subproblem can affect a single source vertex.

We illustrate these concepts by an example. Suppose we want to solve the minimum dominating set problem on a partial k-tree G: minimize $|S|$ over all $S \subseteq V(G)$ such that every vertex not in S has at least one neighbor in S. Relative to some partial dominating set $S \subseteq V(G_u)$, a source vertex $v \in V(G_u)$ of a node u of the parse tree could be in one of 3 states: [dominator] $v \in S$; [non-dominator, non-dominated] $v \notin S \wedge |N_{G_u}(v) \cap S| = 0$; [non-dominator, dominated] $v \notin S \wedge |N_{G_u}(v) \cap S| \geq 1$ (we call S a partial dominating set since at non-root nodes of the parse tree source vertices can be in the state [non-dominator, non-dominated].) A table entry at node u gives the minimum number of dominator **non-sources** in G_u necessary to ensure that all non-sources are either dominators or dominated and that the vertex states for source vertices of G_u correspond to the table index.

Consider the binary parse tree in Figure 2. The table of the lower-left Join node, labelled 9765(1), would have 3^4 entries, one entry for each assignment of one of the three vertex states to the four sources. In the subgraph associated with this Join node (see Figure 1), the sources 9,7,6 and 5 form a clique and vertices 5 and 6 share the neighboring non-source vertex 1. We first describe the vertex state assignments that indicate an illegal configuration. Since we are solving a minimization problem, the corresponding table entries will have value $+\infty$:

- two sources have the pair of states [dominator] and [non-dominator, non-dominated].
- 7 or 9 have state [non-dominator, dominated] but no source has state [dominator].
- 5 or 6 have state [non-dominator, non-dominated].

The latter case is illegal since then non-source vertex 1 can neither be dominator nor dominated. For the remaining possibilities we have two cases:

- 5 or 6 have state [dominator].
- 5 and 6 both have state [non-dominator, dominated].

In the first case table entries have value zero, since then no dominator non-sources are needed to dominate the non-sources and ensure these vertex states for sources. In the latter case table entries have value one, since non-source vertex 1 will then have to be a dominator itself (1 has neighbors only 5 and 6 and must be either dominated or dominator.)

As mentioned earlier, the sources of G_u constitute a k or $(k+1)$-bag and form a separator of G, which renders possible the table update for all the operations, and in particular $Join(A, B)$ based on the tables of A and B. An algorithm for a given problem must describe the tables involved and also describe how tables are computed during traversal of the parse tree. A candidate table is verified by the correctness proof of table update procedures for all operations involved. The introduction of Reduce and Join greatly simplifies this verification process, since these operations make only minimal changes to their argument graphs. In general, the algorithm computing a parameter $R(G)$ for a partial k-tree G given with a tree-decomposition has the following structure:

```
Algorithm-R, where $R$ is a graph parameter
Input: $G, k$, width $k$ tree-decomposition of $G$
Output: $R(G)$
(1) Based on tree-decomposition find a binary parse tree $T$ of $G$.
(2) Initialize Primitive Tables at leaves of $T$.
(3) Traverse $T$ bottom-up using Join-Tables and Reduce-Table.
(4) Optimize Root Table at root of $T$ gives $R(G)$.
```

Note that a tree-decomposition of width k is given as part of the input. For a given graph G on n

vertices and any fixed k, Bodlaender [6] gives an $O(n)$ algorithm for deciding whether the treewidth of G is at most k and in the affirmative case finding a width k tree-decomposition of G. The time complexity of his algorithm has a coefficient that is exponential in a polynomial in k, a polynomial which is not given explicitly in his paper. Improving on his algorithm to decrease this polynomial is an important problem, which we do not address here. A construction of a k-tree embedding, given a tree-decomposition, is described in [21]. From a k-tree embedding it is straightforward to find a peo and the corresponding peo-tree and to construct the binary parse tree as described in the previous subsection. The time for step (1) becomes $O(nk^2)$.

For a vertex state problem R with vertex state set A, the most expensive operation in the partial k-tree algorithm outlined above is the computation of the table associated with the *Join* operation. The complexity of this computation at a node of the parse tree is proportional to the number of pairs of indices, one index from the table of each of its two children. The table index sets associated with the children of a Join node for the problem R have size $|A|^k$ and $|A|^{k+1}$, and there are fewer than n *Join* nodes in the parse tree. The overall complexity of the algorithm, given a tree-decomposition, is dominated by the total of Join-Tables computation and is equal to $T(n,k,A) = \mathcal{O}(n|A|^{2k+1})$. When $|A|$ does not depend on n we have a finite-state problem and a linear-time algorithm on partial k-trees, for fixed k. Note that a vertex state problem can be solved in polynomial time whenever $|A|$ is polynomial in n.

In section 4, we define a class of vertex partitioning problems, and then in section 5 we give a procedure to produce a set of vertex states and table update procedures for each such problem definition.

4. Vertex Partitioning Problems. In this section, we define a class of discrete optimization problems in which each vertex has a *state*, an attribute that is verifiable by a local neighborhood check.

Our motivation for definition of these vertex partitioning problems is twofold. On the one hand, this formalism provides a general and uniform description of many existing problems in which a solution consists of a selection of vertex subsets. On the other, being vertex state problems, their restriction to partial k-trees have efficient solution algorithms that can be designed according to a general paradigm that follows their vertex partitioning description.

Considering partitions of the vertex set of a given graph is an attempt to unify graph properties expressible by either vertex subsets such as independent dominating set or by vertex coloring of graphs. Both these constructs are constrained by the structure of neighborhoods of vertices in different subsets. We define this formally.

DEFINITION 4.1. *A degree constraint matrix D_q is a q by q matrix with entries being subsets of natural numbers $\{0,1,...\}$. A D_q-partition in a graph G is a partition $V_1, V_2, ..., V_q$ of $V(G)$ such that for $1 \leq i,j \leq q$ we have $\forall v \in V_i : |N_G(v) \cap V_j| \in D_q[i,j]$.*

For technical reasons, we will allow a partition $V_1,...,V_q$ of $V(G)$ to possibly have some empty partition classes, *i.e.*, if the degree constraints on a partition class V_i are satisfied by $V_i = \emptyset$ then we allow this possibility. Given a degree constraint matrix D_q, it is natural to ask about the existence of a D_q-partition in an input graph. We call this the $\exists D_q$ problem. We might also ask for an extremal value of the cardinality of a vertex partition class over all D_q-partitions. Additionally, given a sequence of degree constraint matrices, $D_1, D_2, ...$, we might want to find an extremal value of q for which a D_q-partition exists in the input graph. We call these partition minimization and partition maximization problems.

To illustrate and give weight to this formalism, we express some well known problems [1] in the

[1][GTx] as a citation refers to the Graph Theory problem number x in Garey and Johnson [12]

terminology of vertex partitioning and also define new vertex partitioning problems as generalizations of old problems. In each case, correctness of the vertex partitioning formulation follows immediately from Definition 4.1.

4.1. Vertex subset problems. Many domination-type problems can be called *vertex subset* problems, as they ask for existence or optimization of a vertex subset with certain neighborhood properties. For example:

INDEPENDENT DOMINATING SET (IDS)
INSTANCE: Graph G.
QUESTION: Does G have an independent dominating set, *i.e.* is there a subset $S \subseteq V(G)$ such that S is independent (no two vertices in S are neighbors) and dominating (each vertex not in S has a neighbor in S)?

Equivalently, the IDS problem is defined with $\sigma = \{0\}$, $\rho = \{1, 2, \ldots\}$ and asking: Does G have a

$$D_2 = \begin{pmatrix} \sigma & \mathbb{N} \\ \rho & \mathbb{N} \end{pmatrix}$$

partition? Such a description defines a $[\rho, \sigma]$-*property*. Table 1 shows some classical vertex subset properties expressed using this notation [14, 8]. The complexity of optimization and existence problems defined over $[\rho, \sigma]$-properties for general graphs was studied in [26]; the existence problem is $\mathcal{NP}$-complete whenever both ρ and σ are finite non-empty sets and $0 \notin \rho$ (note the IDS problem is trivial, every graph has such a set.)

ρ	σ	Standard terminology
$\{0,1,...\}$	$\{0\}$	Independent set
$\{1,2,...\}$	$\{0,1,...\}$	Dominating set
$\{0,1\}$	$\{0\}$	Strong Stable set or 2-Packing
$\{1\}$	$\{0\}$	Perfect Code or Efficient Dominating set
$\{1,2,...\}$	$\{0\}$	Independent Dominating set
$\{1\}$	$\{0,1,...\}$	Perfect Dominating set
$\{1,2,...\}$	$\{1,2,...\}$	Total Dominating set
$\{1\}$	$\{1\}$	Total Perfect Dominating set
$\{0,1\}$	$\{0,1,...\}$	Nearly Perfect set
$\{0,1\}$	$\{0,1\}$	Total Nearly Perfect set
$\{1\}$	$\{0,1\}$	Weakly Perfect Dominating set
$\{0,1,...\}$	$\{0,1,...,p\}$	Induced Bounded-Degree subgraph
$\{p,p+1,...\}$	$\{0,1,...\}$	p-Dominating set
$\{0,1,...\}$	$\{p\}$	Induced p-Regular subgraph

TABLE 1

Some vertex subset properties.■

4.2. Uniform vertex partitioning problems. For a $[\rho, \sigma]$-property, we can also define partition maximization, partition minimization, and q-partition existence problems by taking the degree constraint matrix D_q with diagonal entries σ and off-diagonal entries ρ. We call these problems $[\rho, \sigma]$-*Partition* problems. For example:

GRAPH K-COLORABILITY[GT4]
INSTANCE: Graph G, positive integer k

QUESTION: Is G k-colorable, *i.e.* is there a partition of $V(G)$ into k independent sets?

The graph k-colorability problem is defined with $\sigma = \{0\}$, $\rho = \{0, 1, \ldots\}$, D_k a k by k degree constraint matrix with diagonal elements σ and off-diagonal elements ρ, and asking: Does G have a D_k-partition?

Chromatic number is the partition minimization problem over degree constraint matrices $D_1, D_2, \ldots$ each one defined as D_k above. Similarly, *Domatic Number* [GT3] asks for a partition into maximum number of dominating sets ($\sigma = \mathbb{N}$, $\rho = \{1, 2, \ldots\}$) and *Partition into Perfect Matchings* [GT16] asks for a partition into minimum number of induced 1-regular subgraphs ($\sigma = \{1\}, \rho = \mathbb{N}$).

As an example of a generalization consider the degree constraint matrix defining a partition into two Perfect Dominating Sets

$$D_2 = \begin{pmatrix} \mathbb{N} & \{1\} \\ \{1\} & \mathbb{N} \end{pmatrix}$$

and the question: Does a given graph G have a D_2-partition? This problem, which asks for a special cut of the graph, can also be posed as a vertex labelling question:

PERFECT MATCHING CUT

INSTANCE: Graph G.

QUESTION: Does G have a perfect matching cut, *i.e.* can the vertices of G be labelled with two labels such that each vertex has exactly one neighbor labelled differently from itself?

As an example, binomial trees and hypercubes have perfect matching cuts, This follows immediately from their iterative definition, *i.e.* the binomial tree B_0 is a single vertex and for $i > 0$ the binomial tree B_i is constructed by adding a new leaf to every vertex in B_{i-1}. In [15] the complexity of uniform vertex partitioning problems is studied; Perfect Matching Cut is $\mathcal{NP}$-complete even when restricted to 3-regular graphs.

We can also consider vertex partitions into subsets with *different* properties. In general, take vertex subset properties $[\rho_1, \sigma_1], [\rho_2, \sigma_2], \ldots, [\rho_q, \sigma_q]$, and construct a degree constraint matrix D_q with column i having entry σ_i in position i and ρ_i elsewhere. The $\exists D_q$-problem asks if a graph G has a partition $V_1, V_2, \ldots, V_q$ of $V(G)$ where V_i is a $[\rho_i, \sigma_i]$-set in G.

4.3. Iterated removal problems. A variation of these problems arises by asking if a graph G has a partition $V_1, V_2, \ldots, V_q$ where V_i is a $[\rho, \sigma]$-set in $G \setminus (V_1 \cup V_2 \cup \ldots \cup V_{i-1})$. To define this we use the degree constraint matrix D_q with diagonal entries σ, above-diagonal entries $\mathbb{N}$ and below-diagonal entries ρ. We call the resulting problems $[\rho, \sigma]$-*Iterated Removal* problems, since V_1 is a $[\rho, \sigma]$-set in $G_1 = G$, while V_2 is a $[\rho, \sigma]$-set in $G_2 = G_1 \setminus V_1$, and in general V_i is a $[\rho, \sigma]$-set in $G_i = G_{i-1} \setminus V_{i-1}$ $(1 < i \leq q)$. Here we may have to add the requirement that all partition classes be non-empty. For example:

GRAPH GRUNDY NUMBER [GT56, undirected version]

INSTANCE: Graph G, positive integer k.

QUESTION: Is the Grundy number of G at least k, *i.e.* is there a function $f : V(G) \to \{1, 2, \ldots, k'\}$ for some $k' \geq k$ such that, for each v, $f(v)$ is the least positive integer not contained in the set $\{f(u) : u \in N_G(v)\}$.

Note that if such a function f exists, then the color classes $V_i = \{v : f(v) = i\}, 1 \leq i \leq k'$ form a partition of $V(G)$ and each V_i is an independent dominating set in the graph $G \setminus (V_1 \cup V_2 \cup \ldots \cup V_{i-1})$. We can therefore define the Graph Grundy Number problem as an *Iterated Removal* partition maximization problem. Let $\sigma = \{0\}$, $\rho = \{1, 2, \ldots\}$, and let $D_{k'}$ be a k' by k' degree constraint matrix with diagonal entries σ, above-diagonal entries $\mathbb{N}$ and below-diagonal entries ρ. The Graph Grundy Number problem is: Does G have a $D_{k'}$-partition, with non-empty partition classes, for some $k' \geq k$?

4.4. H-Coloring and H-covering problems. For some vertex partitioning problems the degree constraint matrix is constructed using the adjacency matrix of an arbitrary graph H. For example:

H-COLORING (GRAPH HOMOMORPHISM)[GT52, fixed H version]
INSTANCE: Graph G.
QUESTION: Is there a homomorphism from G to H, *i.e.* is there a function $f : V(G) \to V(H)$ such that $uv \in E(G) \Rightarrow f(u)f(v) \in E(H)$?

We frame H-coloring as a vertex partitioning problem using the degree constraint matrix $D_{|V(H)|}$, obtained from the adjacency matrix of H by replacing 1-entries with $\mathbb{N}$ and 0-entries with $\{0\}$. The question to be asked is: Does G have a $D_{V(H)}$-partition? H-coloring is $\mathcal{NP}$-complete if H is not bipartite and polynomial-time solvable otherwise [16].

H-COVERING
INSTANCE: Graph G.
QUESTION: Does G cover H, *i.e.* is there a degree-preserving function $f : V(G) \to V(H)$ such that for all $v \in V(G)$ we have $\{f(u) : u \in N_G(v)\} = N_H(f(v))$?

Similarly, the H-cover problem, whose complexity was studied in [19], is formulated as an $\exists D_q$ problem using the adjacency matrix of H with singleton entries $\{1\}$ and $\{0\}$.

5. Algorithms for Vertex Partitioning Problems on Partial k-Trees. We give algorithms for solving vertex partitioning problems on partial k-trees. These algorithms take a graph G and a width k tree-decomposition of G as input. Earlier work by Arnborg et al. [2] establishes the existence of pseudo-efficient algorithms for most, but not all, of these problems. They are pseudo-efficient in the sense that their time complexity is polynomial in the size of the input for fixed k, but with horrendous multiplicative constants ("towers" of powers of k). In contrast to this behavior, the algorithms presented here have running times with more reasonable bounds as a function of both input size and treewidth, *e.g.*, $\mathcal{O}(n2^{4k})$ for well-known vertex subset problems. Since these problems are $\mathcal{NP}$-hard in general and a tree-decomposition of width $n-1$ is easily found for any graph on n vertices, we should not expect polynomial dependence on k.

We devote most of this section to describe algorithms that solve $\exists D_q$-problems, for any degree constraint matrix D_q (as defined in the preceding section). In section 5.4 we describe extensions to partition minimization and maximization problems, and problems asking for an extremal value of the cardinality of a vertex partition class.

The algorithms will follow the general outline given in section 3.2, giving an answer YES if the input graph has a D_q-partition and NO otherwise. We first discuss the pertinent vertex and separator states and give a description of the tables involved in the algorithm. We then fill in details of table operations, prove their correctness and give their time complexities.

5.1. Vertex and Separator States. To define the set of vertex states A for an $\exists D_q$ problem, we start with the definition of the problem as captured by the degree constraint matrix D_q. To check whether a given partition $V_1, ..., V_q$ of $V(G)$ is a D_q-partition we first assign to each vertex $v \in V(G)$ with $v \in V_i$ and $|N(v) \cap V_j| = d_j, j = 1, ..., q$ the state $(i)(d_1, d_2, ..., d_q)$ and then check if this state satisfies the constraints imposed by row i of D_q. The states allowed by D_q are called the *final* vertex states. In our partial k-tree algorithms we must consider a refined version of the original problem. For a given partition on a subgraph, a vertex may start out in a state not allowed by D_q and then acquire neighbors through Join operations so that the augmented partition indeed becomes a D_q-partition. To define this larger set of vertex states which are either final or can become final by adding new neighbors we need to define the *augmented* degree constraint matrix AD_q.

For $t \in \mathbb{N}$, we view $\geq t$ as a single element, and define the sets $Y_t \overset{df}{=} \{0,1,...,t\}$, $W_0 = \{\geq 0\}$, $W_t \overset{df}{=} Y_{t-1} \cup \{\geq t\}$ if $t > 0$ and let $R \overset{df}{=} \{Y_t : t \in \mathbb{N}\} \cup \{W_t : t \in \mathbb{N}\} \cup \{\mathbb{N}\}$. Note that $|Y_t| = |W_t| = t+1$. We now define a function $\beta : 2^{\mathbb{N}} \to R$ such that $AD_q[i,j] = \beta(D_q[i,j])$.

Definition 5.1. *$AD_q[i,j] = \beta(D_q[i,j])$ where*

$$\beta(D_q[i,j]) = \begin{cases} Y_t & \textit{if } \exists t \in D_q[i,j] \textit{ such that } t = \max\{D_q[i,j]\} \\ W_t & \textit{if } \exists t \in D_q[i,j] \textit{ with } t \textit{ minimum s.t. } \{t, t+1, ...\} \subseteq D_q[i,j] \\ \mathbb{N} & \textit{otherwise} \end{cases}$$

The set of vertex states A for an $\exists D_q$ problem is defined according to the rows of matrix AD_q. A vertex state consists of a pair $(i)(M)$ where $1 \leq i \leq q$ indexes a row of AD_q and M is an element of the Cartesian product $AD_q[i,1] \times AD_q[i,2] \times ... \times AD_q[i,q]$. We assume that $AD_q[i,j] \neq \mathbb{N}$ for any entry of AD_q, as otherwise we would have an infinite vertex state set and our algorithmic template would not work. Equivalently, we assume that every entry of the degree constraint matrix D_q is cofinite.

Definition 5.2. *For an $\exists D_q$ problem, with cofinite entries of D_q, we define the vertex state set A and a subset, the final vertex state set $F \subseteq A$:*
$A = \{(i)(M_{i1}M_{i2}...M_{iq}) \ : i \in \{1,...,q\} \wedge \forall j(j \in \{1,...,q\} \Rightarrow M_{ij} \in AD_q[i,j])\}$

$F = \{(i)(M_{i1}M_{i2}...M_{iq}) \in A : i \in \{1,...,q\} \wedge \forall j(j \in \{1,...,q\} \Rightarrow$
$(M_{i,j} \in D_q) \vee (AD_q[i,j] = W_t \wedge M_{i,j} = \geq t))\}$

Before continuing, let us first consider an example. Figure 3 shows the matrix D_3 such that the $\exists D_3$ problem decides whether vertices of a graph can be partitioned into 3 independent dominating sets. Note that the partition given in the example is not a D_3-partition, as can be seen from vertex a which needs a new neighbor in V_3 if this partition is to be augmented to a D_3-partition of some supergraph. By applying Definition 5.1 we get for $i = 1,2,3$ $AD_3[i,i] = \beta(D_3[i,i]) = \beta(\{0\}) = Y_0 = \{0\}$ and for $i \neq j$ $AD_3[i,j] = \beta(D_3[i,j]) = \beta(\{1,2,...\}) = W_1 = \{0, \geq 1\}$. Applying Definition 5.2 we then get the 12 vertex states in the vertex state set A:

$$\{(1)(0\ 0\ 0),\ (1)(0\ 0\ \geq 1),\ (1)(0\ \geq 1\ 0),\ (1)(0\ \geq 1\ \geq 1),$$
$$(2)(0\ 0\ 0),\ (2)(0\ 0\ \geq 1),\ (2)(\geq 1\ 0\ 0),\ (2)(\geq 1\ 0\ \geq 1),$$
$$(3)(0\ 0\ 0),\ (3)(0\ \geq 1\ 0),\ (3)(\geq 1\ 0\ 0),\ (3)(\geq 1\ \geq 1\ 0)\}$$

The three states at the rightmost column above (the 4th, 8th and 12th) constitute the final state set F, corresponding to the three rows of the degree constraint matrix D_3. For any partition V_1, V_2, V_3 of $V(G)$ and a vertex $v \in V(G)$ this algorithm uses the following natural definition of $state_{V_1,V_2,V_3}(v)$:

$$state_{V_1,V_2,V_3}(v) = \begin{cases} (1)(0\ \ 0\ \ 0) & \text{if } v \in V_1 \text{ and } |N_G(v) \cap V_1| = 0 \wedge \\ & \wedge |N_G(v) \cap V_2| = 0 \wedge |N_G(v) \cap V_3| = 0 \\ ... & \\ (3)(\geq 1\ \geq 1\ 0) & \text{if } v \in V_3 \text{ and } |N_G(v) \cap V_1| \geq 1 \wedge \\ & \wedge |N_G(v) \cap V_2| \geq 1 \wedge |N_G(v) \cap V_3| = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that this state function is total (defined everywhere) for the set of partitions that could possibly be augmented to a D_3-partition by addition of neighbors to the graph, *i.e.*, all vertices of

$$D_3 = \begin{bmatrix} \{0\} & \{1,2,...\} & \{1,2,...\} \\ \{1,2,...\} & \{0\} & \{1,2,...\} \\ \{1,2,...\} & \{1,2,...\} & \{0\} \end{bmatrix}$$

$$AD_3 = \begin{bmatrix} \{0\} & \{0, \geq 1\} & \{0, \geq 1\} \\ \{0, \geq 1\} & \{0\} & \{0, \geq 1\} \\ \{0, \geq 1\} & \{0, \geq 1\} & \{0\} \end{bmatrix}$$

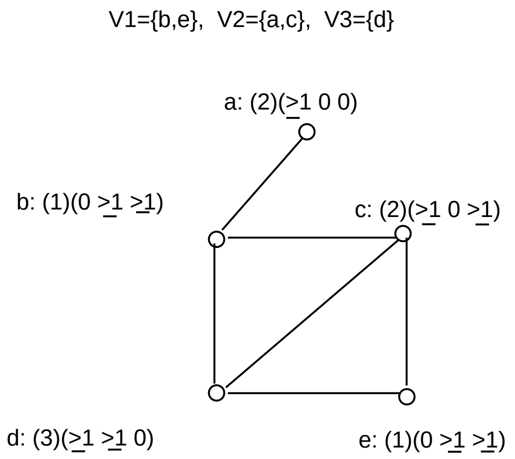

FIG. 3. *The degree constraint matrix D_3 and the augmented degree constraint matrix AD_3 for deciding whether there exists a partition into 3 independent dominating sets. To the right an example with the resulting vertex states for a given partition.*

the graph are assigned a state if (and only if) V_1, V_2, V_3 are independent sets. For a general $\exists D_q$-problem the state function is total for all partitions $V_1, ..., V_q$ that could possibly be augmented to D_q-partitions.

We return to the discussion of a general $\exists D_q$-algorithm and examine first the size of the vertex state set A. Assume for simplicity that the matrix D_q has all diagonal entries equal and all off-diagonal entries equal, with $A_\sigma = AD_q[i,i]$ and $A_\rho = AD_q[i,j]$ for $i \neq j$. With A the set of vertex states for the $\exists D_q$-problem, we thus have $|A| = q|A_\sigma||A_\rho|^{q-1}$ vertex states, since vertex states are of the form $(i)(M_{i1}M_{i2}...M_{iq})$ with $i \in \{1,2,...,q\}$, $M_{ii} \in A_\sigma$ and $M_{ij} \in A_\rho$ for $i \neq j$. We now examine the index set I_k of the table at a node u of the parse tree representing a subgraph with k sources. The table at node u will have $|I_k|$ entries. Let the bag of sources (the separator) at node u be $B_u = \{w_1, w_2, ..., w_k\}$. Each of the sources can take on a vertex state in $|A|$ and the table thus has index set $I_k = \{\mathbf{s} = s_1, ..., s_k\}$ where $s_i \in A$. Thus the size of the table is $|I_k| = |A|^k = q^k |A_\sigma|^k |A_\rho|^{k(q-1)}$. For the earlier example, partition into 3 independent dominating sets, we get $|I_k| = 12^k = 3^k 1^k 2^{k(3-1)}$.

Next, we discuss the values of table entries. For D_q, a subgraph G_u with sources $B_u = \{w_1, ..., w_k\}$ and a k-vector of vertex states $\mathbf{s} = \{s_1, ..., s_k\}$, $\forall i\ s_i \in A$ we define a family Ψ of equivalent partitions $V_1, V_2, ..., V_q$ of $V(G_u)$, such that in G_u, a source w_i has state s_i and a non-source vertex has a final state in F. Note that for a non-source vertex $v \in V_i$ we thus have $|N_G(v) \cap V_j| \in D_q[i,j], j = 1, ..., q$, as dictated by the degree constraint matrix.

DEFINITION 5.3. *For problem $\exists D_q$ with vertex states A and final states F, a graph G_u with sources $B_u = \{w_1, w_2, ..., w_k\}$ and a k-vector $\mathbf{s} = s_1, ...s_k : \forall i\ s_i \in A$ we define*

$$\Psi \overset{df}{=} \{V_1, ..., V_q \text{ a } q\text{-partition of } V(G_u) \ : \forall w_i \in B_u \quad \forall v \in V(G_u) \setminus B_u$$
$$(state_{V_1,...,V_q}(w_i) = s_i \text{ and } state_{V_1,...,V_q}(v) \in F)\}$$

Ψ forms an equivalence class of solutions to the subproblem on G_u, and its elements are called Ψ-partitions respecting G_u and $\mathbf{s}$. The binary contents of $Table_u[\mathbf{s}]$ records whether any solution respecting G_u and $\mathbf{s}$ exists:

DEFINITION 5.4.

$$Table_u[\mathbf{s}] = \begin{cases} 1 & \textit{if } \Psi \neq \emptyset \\ 0 & \textit{if } \Psi = \emptyset \end{cases}$$

5.2. Table Operations. We now elaborate on the operations of Initialize-Table, Reduce-Table, Join-Tables and Optimize-Root-Table in the context of an $\exists D_q$-problem with vertex states A and final states F. Each of the following subsections defines the appropriate procedure, gives the proof of its correctness and analyzes its complexity.

Initialize-Tables

A leaf u of T is a Primitive node and G_u is the graph $G[B_u]$, where $B_u = \{w_1, ..., w_{k+1}\}$. Let $Partitions(B_u)$ be the family of all q^{k+1} partitions of B_u into partition classes $V_1, ..., V_q$. Following Definition 5.4, we initialize $Table_u$ by a brute-force method in two steps:

(1) $\forall \mathbf{s} \in I_{k+1} : Table_u[\mathbf{s}] := 0$

(2) $\forall V_1, V_2, ..., V_q \in Partitions(B_u)$: if in $G[B_u]$ for $i = 1, ..., k{+}1$ we have $state_{V_1,...,V_q}(w_i) = s_i \in A$ then for $\mathbf{s} = s_1, ..., s_{k+1}$, $Table_u[\mathbf{s}] := 1$

We need only consider partitions that assign a state in A to all vertices, since any other partition is in violation of D_q and cannot be augmented to a D_q-partition of the input graph. The complexity of this initialization for each leaf of T is $\mathcal{O}(|I_{k+1}| + (k+1)q^{k+2})$, since for each partition we must check the q neighborhood constraints of $k+1$ vertices.

Reduce Table

A Reduce node u of T has a single child a such that $B_u = \{w_1, ..., w_k\}$ and $B_a = \{w_1, ..., w_{k+1}\}$. We compute $Table_u$ based on $Table_a$ as follows

$\forall \mathbf{s} \in I_k : Table_u[\mathbf{s}] := \bigvee_{\mathbf{p}} \ \{Table_a[\mathbf{p}]\}$

where the disjunction is over all $\mathbf{p} = \{p_1, ..., p_{k+1}\} \in I_{k+1}$ with $\forall l : 1 \leq l \leq k, p_l = s_l$ and $p_{k+1} \in F$. Correctness of the operation follows by noting that G_a and G_u designate the same subgraph of G, and differ only by w_{k+1} not being a source in G_u. By definition, $Table_u[\mathbf{s}]$ should store a 1 if and only if there is some Ψ-set respecting G_a and $\mathbf{s}$ where the state of non-sources, and thus also w_{k+1}, is constrained by D_q, and thus assigned a final state. The complexity of this operation for each Reduce node of T is $\mathcal{O}(|I_{k+1}|)$, assuming that in constant time we can both (i) decide whether an index of $Table_a$ represents a final state for w_{k+1} and (ii) access the corresponding entry of $Table_u$.

Join Tables

A Join node u of T has children a and b such that $B_u = B_a = \{w_1, ..., w_{k+1}\}$ and $B_b = \{w_1, ..., w_k\}$ is a k-subset of B_a. Moreover, G_a and G_b share exactly the subgraph induced by B_b, $G[B_b]$. We compute $Table_u$ by considering all pairs of table entries of the form $Table_a[\mathbf{p}], Table_b[\mathbf{r}]$. Recall that the separator state $\mathbf{p}$ consists of $k+1$ vertex states $p_1, p_2, ..., p_{k+1}$ where the state p_i is associated with vertex w_i. For a vertex state $p_i = (j)(M_{j1}, ..., M_{jq})$ we call j the partition class index, $class(p_i)$, and the cardinality M_{jl}, $size(p_i, l)$. In the algorithm for the Join operation, we first check that $\mathbf{p}, \mathbf{r}$ is a *compatible* separator state pair, meaning the partition class assigned to vertex $w_i, i \in \{1, ..., k\}$, is identical in both $\mathbf{p}$ and $\mathbf{r}$.

$$compatible(\mathbf{p}, \mathbf{r}) := \begin{cases} 1 & \text{if } class(p_i) = class(r_i) \ \forall i \in \{1, ..., k\} \\ 0 & \text{otherwise} \end{cases}$$

We then combine, in a manner described below, for each $w_i, i \in \{1, ..., k+1\}$, the contributions from $\mathbf{p}$ and $\mathbf{r}$ to give the resulting separator state $\mathbf{s} = combine(\mathbf{p}, \mathbf{r})$, and update $Table_u[\mathbf{s}]$ based on $Table_a[\mathbf{p}]$ and $Table_b[\mathbf{r}]$. For a vertex w_i under $\mathbf{s}$, the resulting q-vector of neighborhood sizes is computed by (componentwise) addition of its q-vectors under $\mathbf{p}$ and $\mathbf{r}$. Moreover, since the neighbors w_i has in $B_b = \{w_1, ..., w_k\}$ are the same in both G_a and G_b we must subtract the shared V_j neighbors w_i has in B_b under $\mathbf{p}$ and $\mathbf{r}$. This addition at the jth component is performed using the following definition of $a \oplus b \ominus c$ which adds two size values a, b and subtracts $c \in \mathbb{N}$. The definition of $a \oplus b \ominus c$ depends on whether a, b are of type Y_t or W_t, and returns a value of the same type, unless undefined.

Definition 5.5. *For $a, b \in Y_t$ and $c \in \mathbb{N}$*

$$a \oplus b \ominus c = \begin{cases} a+b-c & \textit{if } a+b-c \in Y_t \\ \uparrow & \textit{otherwise} \end{cases}$$

For $a, b \in W_t$ and $c \in \mathbb{N}$

$$a \oplus b \ominus c = \begin{cases} \geq t & \textit{if either } a \textit{ or } b \textit{ is the element} \geq t \\ \geq t & \textit{if } a+b-c \in \{t, t+1, ...\} \\ a+b-c & \textit{if } a+b-c \in \{0, 1, ..., t-1\} \\ \uparrow & \textit{otherwise} \end{cases}$$

We thus use

$combine(\mathbf{p}, \mathbf{r}) := s_1, s_2, ..., s_{k+1}$ where $\forall i \in \{1, ..., k\}$ $\forall j \in \{1, ..., q\}$
$class(s_i) = class(r_i) = class(p_i)$ and $s_{k+1} = p_{k+1}$ and
$size(s_i, j) = size(p_i, j) \oplus size(r_i, j) \ominus$
$|\{w_l \in B_b : (w_i, w_l) \in E(G) \wedge class(p_l) = j\}|$

We can now state the two step procedure for the Join operation:

(1) $\forall \mathbf{s} \in I_{k+1} : Table_u[\mathbf{s}] := 0;$

(2) $\forall (\mathbf{p} \in I_{k+1}, \mathbf{r} \in I_k) :$ if $compatible(\mathbf{p}, \mathbf{r})$ and $Table_a[\mathbf{p}] = Table_b[\mathbf{r}] = 1$
then $Table_u[combine(\mathbf{p}, \mathbf{r})] := 1$

In step (2) we assume that $Table_u$ is accessed only if $combine(\mathbf{p}, \mathbf{r})$ designates a vertex state, *i.e.*, only if each of its $size$ components is defined.

Theorem 5.6. *The procedure given for the Join Operation at a node u with children a, b correctly updates $Table_u$ based on $Table_a, Table_b$.*

Proof. We argue the correctness of the Join operation at a node u with sources $B_u = \{w_1, ..., w_{k+1}\}$, based on correct table entries at its children a and b, with notation as before. Consider any $\mathbf{s} = s_1, ..., s_{k+1}$ such that there exists a partition $V_1, ..., V_q$ of $V(G_u)$ respecting D_q with $state_{V_1,...,V_q}(w_i) = s_i$ for $i = 1$ to $k+1$ in the graph G_u. We will show that then $Table_u[\mathbf{s}]$ is correctly set to the value 1. Let $A_1, ..., A_q$ and $B_1, ..., B_q$ be the induced partitions on $V(G_a)$ and $V(G_b)$, respectively, *i.e.*, $V_i \cap V(G_a) = A_i$ and $V_i \cap V(G_b) = B_i$. Let $\mathbf{p} = p_1, ..., p_{k+1}$ and $\mathbf{r} = r_1, ..., r_k$ be defined by $p_i = state_{A_1,...,A_q}(w_i)$ in G_a and $r_i = state_{B_1,...,B_q}(w_i)$ in G_b, respectively. By the assumption that $Table_a$ and $Table_b$ are correct we must have $Table_a[\mathbf{p}] = Table_b[\mathbf{r}] = 1$. This follows since any vertex in $V(G_a) \setminus B_u$ has the exact same state in G_a under $A_1, ..., A_q$ as it has in G_u under $V_1, ..., V_q$, by the fact that there are no adjacencies between a vertex in $V(G_a) \setminus B_u$ and a vertex in $V(G_b) \setminus B_u$. Similarly for G_b. We can check that from the definitions we have $compatible(\mathbf{p}, \mathbf{r}) = 1$ and $combine(\mathbf{p}, \mathbf{r}) = \mathbf{s}$, so indeed $Table_u[\mathbf{s}]$ is set to 1 when the pair $\mathbf{p}, \mathbf{r}$ is considered by the algorithm.

Now consider an $\mathbf{s}$ such that there is no q-partition of $V(G_u)$ respecting D_q and resulting in $\mathbf{s}$ as the state for the separator. We will show, by contradiction, that in this case $Table_u[\mathbf{s}]$ is set to 0 initially and then never altered. If $Table_u[\mathbf{s}] = 1$ there must be a compatible pair $\mathbf{p}, \mathbf{r}$ such that $combine(\mathbf{p}, \mathbf{r}) = \mathbf{s}$ and $Table_a[\mathbf{p}] = Table_b[\mathbf{r}] = 1$. Let $A_1, ..., A_q$ and $B_1, ..., B_q$ be partitions of $V(G_a)$ and $V(G_b)$, respectively, that cause these table entries to be set to 1. Then $V_1, ..., V_q$ defined by $V_i = A_i \cup B_i$ is a q-partition of $V(G_u)$ respecting D_q such that the resulting state for the separator is $\mathbf{s}$, because $B_u = \{w_1, ..., w_{k+1}\}$ separates G_u into $G_a \setminus B_u$ and $G_b \setminus B_u$. This contradicts our assumption that such a q-partition does not exist. We conclude that the Join-Tables operation is correct. □

For each Join node of T, the complexity of Join-Tables is $\mathcal{O}(kq|I_k||I_{k+1}|)$ since any pair of entries from tables of children is considered at most once, and for each compatible pair the combine operation considers kq size pairs.

Optimize Root Table

Let the root of T have child r with $B_r = \{w_1, ..., w_k\}$. We decide whether G has a D_q-partition based on $Table_r$ as follows

YES if $\exists \mathbf{s} = s_1, ..., s_k \in I_k$ with $Table_r[\mathbf{s}] = 1$ and $s_i \in F$ for $1 \leq i \leq k$
NO otherwise

Correctness of this optimization follows from the definition of table entries and final states and the fact that G_r is the graph G with sources B_r. The complexity of Optimize-Root-Table at the root of T is $\mathcal{O}(|I_{k+1}|)$, assuming that in constant time we can decide whether an index of $Table_r$ represents a final state for each vertex in B_r.

5.3. Overall Correctness and Complexity. Correctness of an algorithm based on this algoritmic template follows by induction on the binary parse tree T. As noted in Section 3.1, T has $n-k$ Primitive nodes, $n-k$ Reduce nodes and $n-k-1$ Join nodes. The algorithm finds the binary parse tree T, traverses it bottom-up executing the respective operation at each of its nodes, and performs Optimize-Root-Table at the root.

THEOREM 5.7. *The time complexity for solving an $\exists D_q$ problem, entries of D_q cofinite, with vertex state set A, on a partial k-tree G with n vertices, given a width k tree-decomposition of G is $\mathcal{O}(nkq|A|^{2k+1})$. If the augmented degree constraint matrix AD_q has $|A_\sigma| = \max_i\{|AD_q[i,i]|\}$ and $|A_\rho| = \max_{i \neq j}\{|AD_q[i,j]|\}$ it can be expressed as $\mathcal{O}(nq^{2(k+1)}|A_\sigma|^{2k+1}|A_\rho|^{(2k+1)(q-1)})$*

Proof. The first bound holds since the most expensive operation is Join Tables which costs $\mathcal{O}(kq|I_k||I_{k+1}|)$ where $|I_k| = |A|^k$, and there are less than n Join Table nodes in the binary parse tree. The refined bound holds since $|A| = q|A_\sigma||A_\rho|^{q-1}$. □

Note that the last bound holds in particular when AD_q has all diagonal entries equal to A_σ and all off-diagonal entries equal to A_ρ.

5.4. Extensions. Here we mention a few natural extensions of the problems described above: partition maximization and minimization, construction of a D_q-partition, complexity of vertex subset problems, optimization over a partition class cardinality and finally, implications on optimizations problems without a constant bound.

Recall that given a sequence of degree constraint matrices, $D_1, D_2, ...$, partition minimization or maximization problems involve finding an extremal value of q for which a D_q-partition exists in the input graph. To solve such problems with an upper bound $f(n,k)$ on the parameter in question for n-vertex partial k-trees, we need at most $f(n,k)$ calls to the $\exists D_q$ algorithm, for different values of q. Several parameters are bounded by the treewidth only, *e.g.* chromatic number and domatic number are bounded by $k+1$ on partial k-trees. We call a partition maximization (respectively,

Problem	q	$\lvert A_\sigma\rvert$	$\lvert A_\rho\rvert$	Time Complexity
CHROMATIC NUMBER	$1 \leq q \leq k+1$	1	1	$\mathcal{O}(nk^{2(k+1)})$
q-COLORING	q	1	1	$\mathcal{O}(nq^{2(k+1)})$
H-COVER	$q = \lvert V(H)\rvert$	1	2	$\mathcal{O}(n2^{3k\lvert V(H)\rvert})$
H-COLOR	$q = \lvert V(H)\rvert$	1	1	$\mathcal{O}(n\lvert V(H)\rvert^{2(k+1)})$
DOMATIC NUMBER	$1 \leq q \leq k+1$	1	2	$\mathcal{O}(n2^{3k^2})$
GRUNDY NUMBER	$1 \leq q \leq 1 + k \log n$	1	2	$\mathcal{O}(n^{3k^2})$
ITERATED DOM. REMOVAL	$1 \leq q \leq 1 + k \log n$	1	2	$\mathcal{O}(n^{3k^2})$

TABLE 2

Time complexity for specific problems on partial k-trees of n vertices∎

minimization) parameter *monotone* if existence of a D_q-partition implies the existence of a D_{q-1}-partition (respectively, a D_{q+1}-partition). For monotone properties we can apply binary search so that $\log f(n,k)$ calls to the $\exists D_q$ algorithm suffices. Resulting time bounds for specific problems are shown in Table 2, and discussed also in the following sections.

To construct a D_q-partition, in case of a positive answer for the $\exists D_q$ problem, we add pointers from a positive table entry to the table entries of children which updated it positively.

Table 1 in section 4 lists some vertex subset properties, which we called $[\rho, \sigma]$-properties, expressible by a degree constraint matrix D_2. Various $\mathcal{NP}$-hard optimization problems ask for an extremal value of the cardinality of a vertex subset with some $[\rho, \sigma]$ property. In an earlier paper [28], we give algorithms on partial k-trees for solving these problems.

THEOREM 5.8. *[28] Given a tree-decomposition of width k of a graph G, any optimization problem over a $[\rho, \sigma]$-property, with both ρ and σ cofinite, can be solved on G in $\mathcal{O}(n(|\beta(\rho)| + |\beta(\sigma)|)^{2k+1})$ steps.*

Problems defined over properties derived from Table 1 have complexity $\mathcal{O}(n2^{4k})$ (for parameterized properties we assume $p \leq 2$). Those algorithms are very similar to the ones given here, with values of table entries defined to be:

$$Table_u[\mathbf{s}] \stackrel{df}{=} \begin{cases} \bot & \text{if } \Psi = \emptyset \\ optimum_{\{V_1,V_2\}\in\Psi}\{|V_1|\} & \text{otherwise} \end{cases}$$

and table operations altered similarly to optimize this value. Any vertex partitioning problem optimizing over the cardinality of a partition class can be solved in a similar manner. The time complexity of the resulting algorithm for a problem given by the degree constraint matrix D_q remains as given in Theorem 5.7.

In Section 4 we discussed several new problems, including the general classes of $[\rho, \sigma]$ uniform partition problems, $[\rho, \sigma]$ iterated removal problems, H-coloring and H-covering problems. All these problems are encompassed by Theorem 5.7. Polynomial-time algorithms for partition maximization or minimization problems on partial k-trees are constructible in this manner only if an appropriate bound holds for the parameter in question. In the next section we first show that the Grundy Number of an n-vertex partial k-tree has an upper bound logarithmic in n and then construct a polynomial-time solution algorithm.

6. Grundy Number Algorithm.

The Grundy Number of a graph, defined in Section 4.3, is a tight upper bound on the number of colors used by the following 'naive greedy coloring' algorithm:

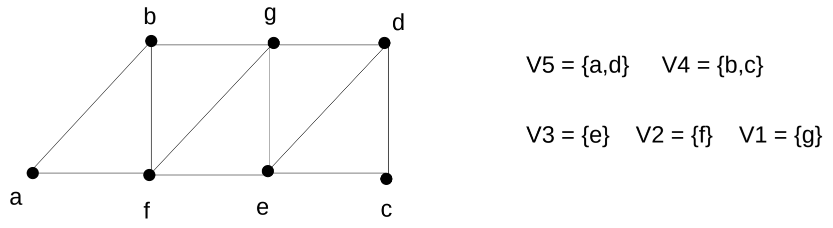

FIG. 4. *A 2-tree on 7 vertices with Grundy number 5 and an appropriate partition* $V1, V2, ..., V5$ ∎

repeatedly select an uncolored vertex and color it with the least available positive integer. The Grundy number is the highest color thus assigned to any vertex, maximized over all orderings of vertices, with the vertex partitioning iterated removal definition based on the fact that the set of vertices with color i form an independent dominating set in the graph induced by vertices with color i or higher.

Computing the Grundy number of an undirected graph is $\mathcal{NP}$-complete even for bipartite graphs and for chordal graphs [22]. A binomial tree on 2^{q-1} vertices, defined in section 4.2, has Grundy number q [13]. In general the non-existence of an $f(k)$ upper bound on the Grundy number of a partial k-tree explains the lack of a description of this problem in EMSOL [20] (note that [2] mistakenly gives a different impression). For trees there exists a linear time algorithm [13] but until now it was an open question whether polynomial time algorithms existed even for 2-trees.

The definition of Grundy number as a vertex partitioning problem requires all partition classes to be non-empty. In this section we first show how the algorithm template of section 5 can be easily adjusted to enforce this requirement. For a partial k-tree G with n vertices, we prove a logarithmic in n upper bound on the Grundy number of G. These results suffice to show the polynomial time complexity of computing the Grundy number of any partial k-tree, for fixed k.

To facilitate the presentation of these results, we reverse the ordering of the partition classes in the definition of Grundy Number from Section 4; this is expressed by the degree constraint matrix D_q with diagonal entries $\{0\}$, above-diagonal entries $\mathbb{P}$, and below-diagonal entries $\mathbb{N}$. Thus, for a graph G, the Grundy Number $GN(G)$ is the largest value of q such that its vertices $V(G)$ can be partitioned into non-empty classes $V_1, V_2, ..., V_q$ with the constraint that for $i = 1, ..., q$, V_i is an independent set and every vertex in V_i has at least one neighbor in each of the sets $V_{i+1}, V_{i+2}, ..., V_q$ (see Figure 4). Note that if we have at least one vertex $v \in V_1$ then this guarantees that every partition class is non-empty, since D_q requires v to have at least one neighbor in each of $V_2, V_3, ..., V_q$. In the algorithm for deciding whether a partial k-tree has a D_q-partition with non-empty classes, with D_q as described above, we extend the value of a table entry $Table_u[\mathbf{s}]$ by a single extra bit called *nonempty*. This bit will record whether there exists any partition $V_1, ..., V_q$ respecting G_u and the separator state $\mathbf{s}$ such that $V_1 \neq \emptyset$. In the following, we use notation as given in section 5, with the definition of table entries:

$$Table_u[\mathbf{s}] = \begin{cases} \langle 0,0 \rangle & \text{if } \Psi = \emptyset \\ \langle 1,0 \rangle & \text{if } \Psi \neq \emptyset \text{ but } \nexists V_1, V_2, ..., V_q \in \Psi \text{ with } V_1 \neq \emptyset \\ \langle 1,1 \rangle & \text{if } \Psi \neq \emptyset \text{ and } \exists V_1, V_2, ..., V_q \in \Psi \text{ with } V_1 \neq \emptyset \end{cases}$$

The two-step Initialize-Table procedure becomes:

(1) $\forall \mathbf{s} \in I_{k+1} : Table_u[\mathbf{s}] := \langle 0,0 \rangle$

(2) $\forall V_1, V_2, ..., V_q \in Partition(B_u)$: if in $G[B_u]$ for $i = 1, ..., k+1$ we have $state_{V_1,...,V_q}(w_i) = s_i \in A$ then for $\mathbf{s} = s_1, ..., s_{k+1}$

```
        if $V_1 = \emptyset$ set $Table_u[\mathbf{s}] := \langle 1, 0\rangle$
        else if $V_1 \neq \emptyset$ set $Table_u[\mathbf{s}] := \langle 1, 1\rangle$
```

Note that for a leaf u of the binary parse tree of G, all vertices of G_u are sources so the separator state $\mathbf{s}$, in step (2) above, contains the information determining if V_1 is empty. The Reduce-Table procedure remains as given in section 5 except that the disjunction is taken pairwise over both bits in the values of table entries, *i.e.*, $\langle a, b\rangle \vee \langle c, d\rangle = \langle a \vee c\rangle, \langle b \vee d\rangle$. For the Join-Table procedure, the concepts of compatibility and combining of pairs are unchanged, and the two-step update procedure becomes:

```
(1) $\forall \mathbf{s} \in I_{k+1} : Table_u[\mathbf{s}] := \langle 0, 0\rangle$;
(2) $\forall (\mathbf{p} \in I_{k+1}, \mathbf{r} \in I_k)$ : if $compatible(\mathbf{p}, \mathbf{r})$ and
  $Table_a[\mathbf{p}] = \langle 1, x\rangle$ and $Table_b[\mathbf{r}] = \langle 1, y\rangle$ and $Table_u[combine(\mathbf{p}, \mathbf{r})] = \langle z, w\rangle$
      then $Table_u[combine(\mathbf{p}, \mathbf{r})] := \langle 1, x \vee y \vee w\rangle$.
```

Optimize-Root-Table becomes:

```
YES if $\exists \mathbf{s} = s_1, ..., s_k \in I_k$ such that $Table_r[\mathbf{s}] = \langle 1, 1\rangle$ and $s_i \in F$ for $1 \leq i \leq k$.
NO otherwise
```

It is easy to see that the time complexity of the resulting algorithm remains as described by Theorem 5.7.

We now turn to the bound on the Grundy number $GN(G)$ of a partial k-tree G. Since the Grundy number of a graph may increase when some edges of the graph are removed, we cannot restrict our attention to k-trees, but must consider partial k-trees. A tree (*i.e.* a 1-tree) with Grundy number q, witnessed by a (Grundy) partition $V_1, ..., V_q$, must have at least 2^{q-1} vertices since each vertex of the set $\bigcup_{1 \leq i < j} V_i$ has a unique neighbor in V_j thus doubling the size of $\bigcup_{1 \leq i \leq j} V_i$ for each consecutive $1 < j \leq q$. This argument relies on the fact that 1-trees do not have cycles. For a partial k-tree G with $k \geq 2$ and Grundy number q we cannot guarantee the existence of a perfect elimination ordering of vertices that respects a $V_q, ..., V_1$ Grundy partition of $V(G)$, as in the 1-tree example above. See Figure 4 for an example of a 2-tree on 7 vertices with Grundy number 5 which does not have a perfect elimination ordering respecting the partial order given by any Grundy partition $V_5, V_4, ..., V_1$. Hence, the upper bound given below has a somewhat less trivial proof than the 1-tree case.

THEOREM 6.1.

The Grundy number of a partial k-tree G on n vertices, $n \geq k \geq 1$, is at most $1 + k \log_2 n$.

Proof. Let the Grundy number of G be $GN(G) = q$ with $V_1, V_2, ..., V_q$ an appropriate partition of $V(G)$ as described above. For $1 \leq i \leq q$, define G_i to be the graph $G \setminus (\cup V_j, j > i)$. Thus $G_q = G$ and in general G_i is the graph induced by vertices $V_1 \cup V_2 ... \cup V_i$ with V_i a dominating set of G_i. Let $n_i = |V(G_i)|$ and $m_i = |A(G_i)|$. By induction on i from k to q we show that in this range

$$n_i \geq (\frac{k+1}{k})^{i-1}$$

For the base case $i = k$ we have $(2/1)^0 \leq 1 \leq n_1$ and $(3/2)^1 < 2 \leq n_2$ and for $k \geq 3$ $(1 + 1/k)^{k-1} \leq (1 + 1/k)^k \leq e < 3 \leq n_k$. Note that the inequality is strict for $k \geq 2$. We continue with the inductive step of the proof, with the inductive assumption that the inequality holds for j in the range k to

$i-1$ and establish the inequality for $j=i$. Note that $m_i - m_{i-1}$ counts the number of edges in G_i with at least one endpoint in V_i. Since every vertex in $V(G_{i-1}) = V_1 \cup V_2 \cup ... \cup V_{i-1}$ has at least one G_i-neighbor in V_i we get a lower bound on m_i

$$m_i \geq m_{i-1} + n_{i-1}$$

G_i is a subgraph of a k-tree, and if $i \geq k$ then it is a partial k-tree on $n_i \geq k$ vertices. It is well-known that G_i is then a subgraph of a k-tree on n_i vertices, and from the iterative construction of k-trees it is easy to show that we have

$$m_i \leq \frac{k(k-1)}{2} + (n_i - k)k$$

Rearranging terms we get the following bound on n_i for $k \leq i \leq q$

$$n_i \geq \frac{m_i}{k} + \frac{k+1}{2}$$

Repeatedly substituting the m_i bound in the above, we get

$$n_i \geq \frac{m_{i-1} + n_{i-1}}{k} + \frac{k+1}{2} \geq ... \geq \frac{n_k + n_{k+1} + ... + n_{i-2} + n_{i-1}}{k} + \frac{m_k}{k} + \frac{k+1}{2}$$

In the right-hand side we substitute for all n_j the inductive bound $n_j \geq (\frac{k+1}{k})^{j-1}$ to get

$$n_i \geq \frac{1}{k} \sum_{j=k-1}^{i-2} (\frac{k+1}{k})^j + \frac{m_k}{k} + \frac{k+1}{2} = (\frac{k+1}{k})^{i-1} - (\frac{k+1}{k})^{k-1} + \frac{m_k}{k} + \frac{k+1}{2}$$

Since V_j is a dominating set in G_j for $1 \leq j \leq k$ we must have $m_k \geq (k-1)k/2$ which we substitute in the above to get the desired bound

$$n_i \geq (\frac{k+1}{k})^{i-1} - (\frac{k+1}{k})^{k-1} + k \geq (\frac{k+1}{k})^{i-1}$$

Note that the last bound is strict for $k \geq 2$. For $i = q$ we thus get $q \leq 1 + \log_{(k+1)/k} n_q$ (note that $q = GN(G)$ and $n_q = n$) which is a tight bound for $k = 1$. For $k \geq 2$, the base is not an integer and, because of the strict inequality mentioned above we can apply the floor function to the log. Converting bases of the logarithm we get $GN(G) \leq 1 + (\log_2 \frac{k+1}{k})^{-1} \log_2 n \leq 1 + k \log_2 n$. □

THEOREM 6.2. *Given a partial k-tree G on n vertices its Grundy number can be found in $\mathcal{O}(n^{3k^2})$ time.*

Proof. First note that a tree-decomposition can be found in time linear in n [6]. Define the Grundy number problem using the degree constraint matrix D_q with diagonal entries $\{0\}$, above-diagonal entries $\mathbb{P}$, and below-diagonal entries $\mathbb{N}$. We then use the algorithm from section 6.2.2 extended with the *nonempty* information as described above. The correctness of each table operation procedure is easily established, so that by induction over the parse tree we can conclude that the root-optimization procedure will correctly give the answer YES if and only if the input graph has an appropriate partition $V_1, ..., V_q$ with non-empty classes. An affirmative answer implies that $GN(G) \geq q$. Using the bound $GN(G) \leq 1 + k \log_2 n$ we run the $\exists D_q$ algorithm for descending values of q starting with $q = 1 + k \log_2 n$ and halting as soon as an affirmative answer is given. The complexity of this algorithm is then given by appropriately applying Theorem 5.7, with $|A_\sigma| = 1$ and $|A_\rho| = 2$. □

Consider any maximum iterated $[\rho, \sigma]$ removal problem with $\rho = \mathbb{P}$, asking how many times we can remove a σ-constrained dominating set from a graph (compare with Grundy Number which removes independent dominating sets). This translates to a partition maximization problem where the degree constraint matrix has diagonal entries σ, above-diagonal entries $\mathbb{N}$ and below-diagonal entries $\mathbb{P}$. Note that the proof of the logarithmic bound on the Grundy Number in Theorem 6.1 does not use the fact that the classes V_i of the partition are independent sets, only the fact that they are dominating sets in the remaining graph. Thus we get a logarithmic upper bound also on these generalized maximum dominating iterated removal parameters on partial k-trees and a polynomial time algorithm for computing these parameters, for fixed k.

7. Conclusions. In this paper, we have presented a design methodology for practical solution algorithms on partial k-trees and a characterization of a class of vertex partitioning problems. These results were combined by adapting the algorithm design methodology on partial k-trees to vertex partitioning problems, yielding the first algorithms for these problems with reasonable time complexity as a function of treewidth.

Implementation of the resulting algorithms is a project at the University of Bergen [17]. The program for solving the Independent Set problem: maximize $|V_1|$ over partitions (V_1, V_2) satisfying

$$D_2 = \begin{pmatrix} \{0\} & \mathbb{N} \\ \mathbb{N} & \mathbb{N} \end{pmatrix}$$

is about 1000 lines of C++ code. Less than 100 of these lines are problem-specific, *i.e.* to produce a solution algorithm for any other vertex subset problem requires changing only a handful of functions.

The actual running time behaves as predicted by the bounds given in this paper, *e.g.* to solve the independent set problem on an n-node partial k-tree (using a 150 Mhz alpha prosessor-based Digital computer) takes roughly $10^{-5} \cdot n \cdot 2^k$ seconds. For example, on a 3000-node graph with treewidth 5 we solve the maximum independent set problem in about 1 second.

Various improvements can be made to these algorithms to reduce the average, if not worst-case, running-time. For example, one can use parse trees with smaller bags in "thin" parts of the graph or computing table entries can be based only on non-zero table entries in the children.

A recent result [29] shows that control-flow graphs of structured (goto-free) programs have small treewidth, *e.g.* treewidth at most 3 for Pascal programs and treewidth at most 6 for C programs. Moreover, a tree-decomposition of the control-flow graph can be easily computed from the program structure (in fact from the 3-address code), making our algorithms, which require a k-tree embedding (tree-decomposition) relatively easily applicable in various compiler optimization settings.

REFERENCES

[1] S.Arnborg, S.T.Hedetniemi and A.Proskurowski (editors) *Discrete Applied Mathematics* vol. 54 (2-3) (1994) 97-291.

[2] S.Arnborg, J.Lagergren and D.Seese, Easy problems for tree-decomposable graphs, *J. of Algorithms* 12 (1991) 308-340.

[3] S.Arnborg and A.Proskurowski, Linear time algorithms for NP-hard problems on graphs embedded in k-trees, *Discr. Appl. Math.* 23 (1989) 11-24.

[4] M.W.Bern, E.L.Lawler and A.L.Wong, Linear-time computation of optimal subgraphs of decomposable graphs, *J. of Algorithms* 8 (1987) 216-235.

[5] H.L.Bodlaender, Dynamic programming on graphs with bounded treewidth, *Proceedings ICALP 88*, LNCS vol.317 (1988) 105-119.

[6] H.L. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, in *Proceedings STOC'93.*, 226-234.

[7] R.B.Borie, R.G.Parker and C.A.Tovey, Automatic generation of linear algorithms from predicate calculus descriptions of problems on recursive constructed graph families, *Algorithmica*, (1992) 7:555-582.

[8] E.J.Cockayne, B.L.Hartnell, S.T.Hedetniemi and R.Laskar, Perfect domination in graphs, *J. Combinatorics Inform. Systems Sci.* 18 (1993) 136-146.

[9] D.G.Corneil and J.Keil, A dynamic programming approach to the dominating set problem on k-trees. *SIAM Journal of Alg. Disc. Meth.* (1987) 8:535-543.

[10] B.Courcelle, The monadic second-order logic of graphs I: Recognizable sets of finite graphs, *Information and Computation* 85: (1990) 12-75.

[11] B.Courcelle and M.Mosbah, Monadic second-order evaluations on tree-decomposable graphs, *Theoretical Computer Science* 109 (1993) 49-82.

[12] M.R. Garey and D.S. Johnson, *Computers and Intractability*, W.H.Freeman and Co., 1978.

[13] S.M.Hedetniemi, S.T..Hedetniemi and T.Beyer, A linear algorithm for the Grundy (coloring) number of a tree, *Congressus Numerantium vol.* 36 (1982) 351-362.

[14] S.T.Hedetniemi and R.Laskar, Bibliography on domination in graphs and some basic definitions of domination parameters, *Discrete Mathematics* 86 (1990).

[15] P.Heggernes and J.A.Telle, Partitioning Graphs into Generalized Dominating Sets, *Proceedings of XV International Conference of the Chilean Computer Society*, Arica, Chile, 1995, 241-252.

[16] P. Hell and J. Nešetřil, On the complexity of H-colouring, *Journal of Combinatorial Theory B 48* (1990) 92-110.

[17] B. Hiim, forthcoming Technical Report - Dept. of Informatics, University of Bergen.

[18] T. Kloks, Treewidth, LNCS vol. 842 (1994).

[19] J. Kratochvíl, A. Proskurowski, J.A. Telle: On the complexity of graph covering problems, in *Proceedings of 20th International Workshop on Graph-Theoretic Concepts in Computer Science 1994 - WG 94*, LNCS vol. 903 (1995) 93-105.

[20] J.Lagergren, private communication (1994).

[21] J. van Leeuwen, Graph Algorithms, in *Handbook of Theoretical Computer Science vol. A* , Elsevier, Amsterdam, (1990) pg.550.

[22] A. McRae, PhD thesis, Clemson University (1994).

[23] A.Proskurowski and M.Sysło, Efficient computations in tree-like graphs, in *Computing Suppl.* 7 (1990) 1-15.

[24] N. Robertson and P.D. Seymour, Graph minors II: algorithmic aspects of treewidth, *J. of Algorithms* 7 (1986) 309-322.

[25] K.Takamizawa, T.Nishizeki and N.Saito, Linear-time computability of combinatorial problems on series-parallel graphs, *J. ACM* 29(1982) 623-641.

[26] J.A.Telle, Complexity of domination-type problems in graphs, *Nordic Journal of Computing* 1(1994) 157-171.

[27] J.A.Telle, Vertex Partitioning Problems: Characterization, Complexity and Algorithms on Partial k-Trees, Ph.D. thesis, University of Oregon, CIS-TR-94-18.

[28] J.A.Telle and A.Proskurowski, Practical algorithms on partial k-trees with an application to domination-type problems, in *Proceedings Workshop on Algorithms and Data Structures, Montreal 93*, LNCS vol. 709 (1993) 610-621.

[29] M.Thorup, Structured programs have small treewidth and good register allocation, Technical Report DIKU-TR-95/18, Department of Computer Science, University of Copenhagen, Denmark, 1995.

[30] T.Wimer, Linear time algorithms on k-terminal graphs. Ph.D. thesis, Clemson University, South Carolina, (1988).