Solving Problems on Recursively Constructed Graphs

RICHARD B. BORIE

University of Alabama

and

R. GARY PARKER and CRAIG A. TOVEY

Georgia Institute of Technology

Fast algorithms can be created for many graph problems when instances are confined to classes of graphs that are recursively constructed. This article first describes some basic conceptual notions regarding the design of such fast algorithms, and then the coverage proceeds through several recursive graph classes. Specific classes include trees, series-parallel graphs, k-terminal graphs, treewidth-k graphs, k-trees, partial k-trees, k-jackknife graphs, pathwidth-k graphs, bandwidth-k graphs, cutwidth-k graphs, branchwidth-k graphs, Halin graphs, cographs, cliquewidth-k graphs, k-NLC graphs, k-HB graphs, and rankwidth-k graphs. The definition of each class is provided. Typical algorithms are applied to solve problems on instances of most classes. Relationships between the classes are also discussed.

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Theory; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Bandwidth, branchwidth, cliquewidth, cograph, cutwidth, dynamic programming, Halin graph, pathwidth, rankwidth, series parallel, tree, treewidth

ACM Reference Format:

Borie, R. B., Parker, R. G., and Tovey, C. A. 2008. Solving problems on recursively constructed graphs. ACM Comput. Surv. 41, 1, Article 4 (December 2008), 51 pages DOI = 10.1145/1456650.1456654 http://doi.acm.org/10.1145/1456650.1456654

1. INTRODUCTION

This survey/tutorial article discusses how to design efficient algorithms for fundamental graph problems (such as vertex cover, dominating set, coloring, and Hamiltonicity) when instances are restricted to belong to a recursively constructible graph class. The idea for this article originated from an earlier survey [Borie et al. 2004], but the present work has been expanded to include a wider selection of important and well-known

Authors' addresses: R. B. Borie, Department of Computer Science, University of Alabama, Box 870290, Tuscaloosa, AL 35487-0290; email: borie@cs.ua.edu; R. G. Parker and C. A. Tovey, School of Industrial and Systems Engineering, Georgia Institute of Technology, 765 Ferst Drive NW, Atlanta, GA 30332-0205.

recursive graph classes, and also to include more problem/algorithm variations (such as min-max, counting, bottleneck, and polynomial-time). We hope this article can serve as a useful guide for all researchers who want to solve problems on recursive graph classes, and also as a good starting point for those who are interested in pursuing research in this area.

Throughout, $G = (V, E)$ denotes a finite graph where V and E are sets of vertices and edges, respectively. A *recursively constructed graph class* is defined by a set (usually finite) of primitive or *base graphs*, in addition to one or more operations (called *composition rules*) that compose larger graphs from smaller subgraphs in the class. Each operation involves fusing specific vertices from each subgraph or adding new edges between specific vertices from each subgraph. (For some graph classes, we will refer to these specific vertices as *terminals*.) Each graph in a recursive class has at least one corresponding *decomposition tree* that reveals how to build it from base graphs.

An efficient algorithm for a problem restricted to a recursively constructed graph class typically employs a dynamic programming approach as follows. Usually we begin by defining a more general problem, such that the solution to the original problem can be obtained from the solution to the general one. Now, solve the general problem on the base graphs defined for the given class (this step is usually trivial), and then combine the solutions for subgraphs into a solution for a larger graph that is formed by the specific composition rules that govern construction of members in the class. Once the solution to the general problem for the entire graph has been produced, use it to obtain the solution to the original problem.

A linear-time algorithm is achieved by determining a finite number of equivalence classes that correspond to each node in a member graph's tree decomposition. These equivalence classes typically correspond to the finite state set in the dynamic programming algorithm. The number of such equivalence classes is constant with respect to the size of the input graph, but may depend upon a parameter (k) associated with the class. A polynomial-time algorithm can often be created if the number of equivalence classes required for the problem grows only polynomially with input graph size. It is also important that a graph's tree decomposition be given as part of the instance or that it can be produced efficiently. When the simplest version of a problem (cardinality or existence) can be solved using this dynamic programming approach, then other more complicated versions (involving vertex or edge weights, counting, bottleneck, min-max, etc.) can generally be routinely solved.

The organization of the article proceeds as follows. Generally, each section is devoted to a specific graph class, beginning with trees. In each section, the class is defined. In most sections, some algorithms are formally stated (or at least described in sufficient detail); also, some explicit examples are given, that is, solutions to some selected problems (vertex cover, Hamiltonian path, minimum-maximal matching, etc.) are demonstrated on an instance from the respective graph class. However, in other sections, no algorithms or examples are explicitly given; rather, we discuss how the current graph class is related to another class, so that problems on this class can be solved by reducing them to problems on the other class.

In most of our examples, problem selection is confined to cases where the specific problem is $\mathcal{NP}$-complete on arbitrary graphs. Note that there is some unevenness in terms of the numbers of algorithms exhibited across the various graph classes considered. Dictated by space requirements alone, this typically means that a richer breadth of algorithms is possible for some of the simpler classes such as trees, series-parallel graphs, and cographs. More importantly, however, in such cases, we are typically able to present a better variety of problems and in so doing demonstrate more easily some of the nuance that accompanies various structural differences insofar as algorithm design is concerned but where these designs carry forward to more complicated classes.

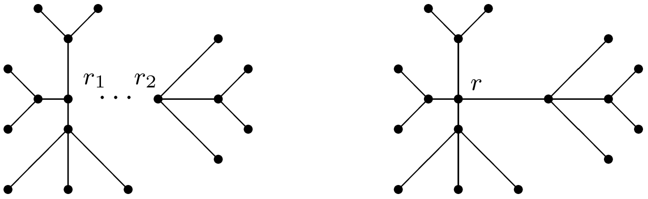


Fig. 1. Recursive construction of a tree.

Most sections are concluded with some general remarks pertaining to the specific graph class. The detail in this coverage will vary broadly from section to section. In some cases, remarks will describe structural, graph-theoretic attributes relevant to the graph class; sometimes, the remarks will describe special computational and algorithmic properties of the class; and in others, the discussion simply serves to act as a transition that clarifies how a given graph class relates to others that are discussed in other sections.

In algorithmic descriptions, we shall generally employ notation where *G.x* (or sometimes $G[x]$) denotes an attribute *x* (corresponding to an equivalence class) of a given graph G. Within most algorithms, *G.x* carries only $O(1)$ pieces of information, rather than a complete description of the equivalence class; in the remaining algorithms, *G.x* will have size $O(|V|^j)$ for some constant j. When we write, for example, *G.x* = *minimum-cardinality vertex cover*, then *G.x* carries with it two pieces of information: primarily, the size of a minimum cardinality vertex cover, and secondarily, a *pointer* to a *particular instance* of such a set. This information is carried forward in computations and assignments involving *G.x*. Moreover, our pseudocode will generally entail only the primary information, such as existence, size, or weight, and suppress the secondary information, that is, the pointer to a particular instance.

2. TREES

A *tree* is a connected, acyclic graph. Important for our purposes here, however, is a simple recursive definition: A graph with a single vertex r (and no edges) is a tree with root r (the sole base graph). Now, let (G, r) denote a tree with root r. Then $(G_1, r_1) \oplus (G_2, r_2)$ is a tree formed by taking the disjoint union of G_1 and G_2 and adding an edge (r_1, r_2). The root of this new tree is $r = r_1$. Figure 1 illustrates the construction. We have abused terminology slightly in that the pairs (G, r) actually denote *rooted trees*. However, the identification of distinguished vertices r_1 and r_2 (and hence r) is relevant here solely as a device in the recursive construction.

Following, we state several algorithms for problems on trees. With the first algorithm, we provide some elementary exposition to underscore the basic concepts. For subsequent cases, we will simply state the specific algorithms. We generally express each algorithm in pseudocode by writing formulas that compute the attribute values at each node in the decomposition tree. These formulas utilize the optimal substructure property, that is, optimal solutions of subproblems are used to compute optimal solutions for larger problems. However, due to the presence of overlapping subproblems, straightforward recursive implementations would be very inefficient. As is typical in such cases, the algorithms can be implemented using standard dynamic programming techniques. Either employ a bottom-up computation strategy (start at the leaves of the decomposition tree and work toward the root) or a top-down strategy (use recursion with memoization).

$\oplus$	**a**	**b**
a	a	a
b	b	

Fig. 2. Multiplication table for vertex cover in a tree.

```
if $|V|$=1 then
    $G.a \leftarrow 1$
    $G.b \leftarrow 0$
else if $G = G_1 \oplus G_2$ then
    $G.a \leftarrow$ min $\{G_1.a + G_2.a, G_1.a + G_2.b\}$
    $G.b \leftarrow G_1.b + G_2.a$
$G.c \leftarrow$ min $\{G.a, G.b\}$
```

Fig. 3. Algorithm for minimum cardinality vertex cover in a tree.

2.1. Algorithm for Minimum Cardinality Vertex Cover in a Tree

A *vertex cover* in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ with the property that every edge in E has at least one of its endpoints in S. A minimum vertex cover for G is a smallest S with this property.

Now, given any tree (or subtree) G, with a designated root vertex which we now denote by *root* $[G]$, it must be that any admissible vertex cover of vertices either includes *root* $[G]$ or does not. Whether or not vertex covers of two trees G_1, G_2 can be combined into a vertex cover of the tree $G_1 \oplus G_2$ depends *only* on these inclusions. Hence, let the equivalence classes be as follows.

$G.a$ = min cardinality vertex cover that includes *root* $[G]$
$G.b$ = min cardinality vertex cover that excludes *root* $[G]$
$G.c$ = min cardinality vertex cover

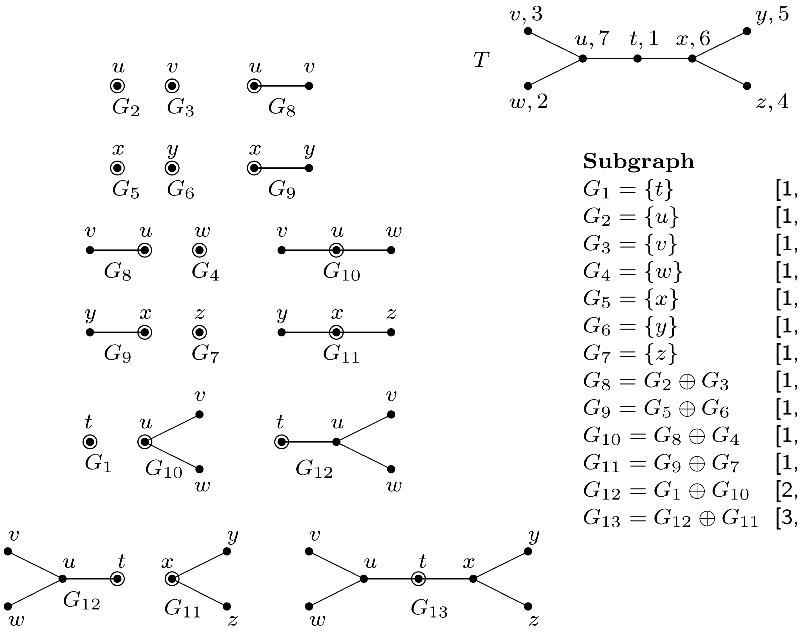
It is convenient to describe the computation by a *multiplication table* as in Figure 2, which shows all possible outcomes from the composition $G_1 \oplus G_2$. Observe that in the table, we have suppressed notation slightly, that is, rather than listing $G.a$ and $G.b$ we simply specify **a** and **b**. The row-by-column product assumes the convention where subgraphs (subtrees) G_1 and G_2 correspond to the row and column, respectively.

The values for the equivalence classes $G.a$, $G.b$, $G.c$ are known trivially for the base graph, that is, a single-vertex tree (which is its own *root*). For composed graphs, the values may be computed via $O(1)$ sums and comparisons across the outcomes in the table. There is only one product that produces a possible member of $G.b$, but $G.a$ may be produced from a pair of possible products; the minimum of these yields the desired $G.a$. The final step computes $G.c$, which at the root of the decomposition tree yields the solution. Formally, we have the algorithm shown in Figure 3.

A decomposition tree for trees is easy to determine and, accordingly, can be assumed to be part of the instance. Also, it is clear that the successive computation of attributes $G.a$, $G.b$ and, finally, $G.c$ actually carry forward two pieces of information: first the *size of a member*, and second *one particular member*, of the equivalence class.

```
if $|V|$=1 then
    $G.d \leftarrow$ weight(root $[G]$)
    $G.e \leftarrow 0$
else if $G = G_1 \oplus G_2$ then
    $G.d \leftarrow G_1.d + G_2.f$
    $G.e \leftarrow G_1.e + G_2.d$
$G.f \leftarrow$ min $\{G.d, G.e\}$
```

Fig. 4. Algorithm for minimum weighted vertex cover in a tree.



Subgraph	
$G_1 = \{t\}$	[1, 0, 1, 1, 0, 1]
$G_2 = \{u\}$	[1, 0, 1, 7, 0, 7]
$G_3 = \{v\}$	[1, 0, 1, 3, 0, 3]
$G_4 = \{w\}$	[1, 0, 1, 2, 0, 2]
$G_5 = \{x\}$	[1, 0, 1, 6, 0, 6]
$G_6 = \{y\}$	[1, 0, 1, 5, 0, 5]
$G_7 = \{z\}$	[1, 0, 1, 4, 0, 4]
$G_8 = G_2 \oplus G_3$	[1, 1, 1, 7, 3, 3]
$G_9 = G_5 \oplus G_6$	[1, 1, 1, 6, 5, 5]
$G_{10} = G_8 \oplus G_4$	[1, 2, 1, 7, 5, 5]
$G_{11} = G_9 \oplus G_7$	[1, 2, 1, 6, 9, 6]
$G_{12} = G_1 \oplus G_{10}$	[2, 1, 1, 6, 7, 6]
$G_{13} = G_{12} \oplus G_{11}$	[3, 2, 2, 12, 13, 12]

Fig. 5. Minimum cardinality and minimum weight vertex covers in a tree.

2.2. Algorithm for Minimum Weighted Vertex Cover in a Tree

Following the approach used to find the minimum-cardinality vertex cover, switching to a weighted version of the vertex cover problem is straightforward. Let

$G.d$ = min weight vertex cover that includes *root* $[G]$
$G.e$ = min weight vertex cover that excludes *root* $[G]$
$G.f$ = min weight vertex cover

The algorithm is shown in Figure 4. The stated expression for $G.d$ results from this simplification:

$$\min\{G_1.d + G_2.d, G_1.d + G_2.e\} = G_1.d + \min\{G_2.d, G_2.e\} = G_1.d + G_2.f.$$

2.2.1. Example. Consider the tree denoted by T in Figure 5. Vertices are labelled $t, u, \ldots, z$ and next to each label is a vertex weight. The algorithms of Figures 3 and 4 are applied, and the computations are summarized by the listing on the right. The 6-tuples aligned with each composed subgraph, G_k, correspond to values

```
if $|V|$=1 then
    $G.g \leftarrow 1$
    $G.h \leftarrow 0$
else if $G = G_1 \oplus G_2$ then
    $G.g \leftarrow G_1.g + G_2.i$
    $G.h \leftarrow G_1.h + G_2.g$
$G.i \leftarrow$ if $G.d < G.e$ then $G.g$
    else if $G.d > G.e$ then $G.h$
    else $\max\{G.g, G.h\}$
```

Fig. 6. Algorithm for maximum cardinality minimum weight vertex cover in a tree.

```
if $|V|$=1 then
    $G.j \leftarrow$ weight(*root* $[G]$)
    $G.k \leftarrow \infty$
else if $G = G_1 \oplus G_2$ then
    $G.j \leftarrow \max\{G_1.j, G_2.l\}$
    $G.k \leftarrow \max\{G_1.k, G_2.j\}$
$G.l \leftarrow$ if $G.d < G.e$ then $G.j$
    else if $G.d > G.e$ then $G.k$
    else $\min\{G.j, G.k\}$
```

Fig. 7. Algorithm for bottleneck vertex cover in a tree.

$[G.a, G.b, G.c, G.d, G.e, G.f]$. The minimum cardinality and minimum weight of any vertex cover are $G.c = 2$ and $G.f = 12$, respectively; these are read from the computation for G_{13}. Standard backtracking can be applied to determine that the explicit solutions are sets $\{u, x\}$ and $\{t, v, w, x\}$, respectively.

2.3. Algorithm for Maximum Cardinality Minimum Weight Vertex Cover in a Tree

Among all minimum weight vertex covers, suppose we wish to determine one of maximum possible cardinality. Define

$G.g$ = max cardinality set that yields $G.d$
$G.h$ = max cardinality set that yields $G.e$
$G.i$ = max cardinality min weight vertex cover

The algorithm is shown in Figure 6.

2.4. Algorithm for Bottleneck Vertex Cover in a Tree

Consider a bottleneck version of the previous weighted vertex cover problem. In other words, among all minimum weight vertex covers, find the minimum possible maximum weight of any vertex in such a cover. Define

$G.j$ = min possible max weight vertex in a set that yields $G.d$
$G.k$ = min possible max weight vertex in a set that yields $G.e$
$G.l$ = min possible max weight vertex in a min weight vertex cover

The algorithm is shown in Figure 7.

```
if $|V|$=1 then
    $G.m \leftarrow 1$
    $G.n \leftarrow 1$
else if $G = G_1 \oplus G_2$ then
    $G.m \leftarrow G_1.m \times G_2.p$
    $G.n \leftarrow G_1.n \times G_2.m$
$G.p \leftarrow$ if $G.d < G.e$ then $G.m$
    else if $G.d > G.e$ then $G.n$
    else $G.m + G.n$
```

Fig. 8. Algorithm for counting the minimum weighted vertex covers in a tree.

$\oplus$	**t**	**u**	**v**
t	t	t	t
u	u	u	
v	u	v	

Fig. 9. Multiplication table for dominating set in a tree.

2.5. Algorithm for Counting the Minimum Weighted Vertex Covers in a Tree

Next we consider a counting version of the weighted vertex cover problem. Define

$G.m$ = the number of sets that yield $G.d$
$G.n$ = the number of sets that yield $G.e$
$G.p$ = the number of vertex covers with minimum weight

The algorithm is shown in Figure 8.

2.6. Algorithm for Minimum Cardinality Dominating Set in a Tree

For $G = (V, E)$, a subset of vertices $S \subseteq V$ is a *dominating set* if every vertex in $V - S$ is adjacent to some vertex in S. Let

$G.t$ = min cardinality dominating set that includes *root* $[G]$
$G.u$ = min cardinality dominating set that excludes *root* $[G]$
$G.v$ = min cardinality almost-dominating set (only *root* $[G]$ undominated)
$G.w$ = min cardinality dominating set

Interesting here is that, unlike the case of vertex cover, solutions in each subgraph to be composed need not be admissible, that is, need not be dominating sets. However, when a subgraph solution is inadmissible, the only violating member is the root vertex, which can ultimately become dominated by the root vertex of the subgraph with which it is composed under the respective $\oplus$. This is captured by attribute $G.v$. The multiplication table for the dominating set problem on trees is given in Figure 9. Specifically then we have the algorithm shown in Figure 10.

2.7. Algorithm for Longest Cardinality Path in a Tree

Let

$G.x$ = max length of path from leaf to root
$G.y$ = max length of path from leaf to leaf
$G.z$ = max length of any path

```
if $|V|$=1 then
    $G.t \leftarrow 1$
    $G.u \leftarrow \infty$
    $G.v \leftarrow 0$
else if $G = G_1 \oplus G_2$ then
    $G.t \leftarrow G_1.t$ + min $\{G_2.v, G_2.w\}$
    $G.u \leftarrow$ min $\{G_1.u + G_2.w, G_1.v + G_2.t\}$
    $G.v \leftarrow G_1.v + G_2.u$
$G.w \leftarrow$ min $\{G.t, G.u\}$
```

Fig. 10. Algorithm for minimum-cardinality dominating set in a tree.

$\oplus$	**x**	**y**	$\emptyset$
x	y		x
y			y
$\emptyset$	x	y	$\emptyset$

Fig. 11. Multiplication table for longest path in a tree.

```
if $|V|$=1 then
    $G.x \leftarrow 0$
    $G.y \leftarrow -\infty$
else if $G = G_1 \oplus G_2$ then
    $G.x \leftarrow$ max $\{G_1.x, G_2.x + 1\}$
    $G.y \leftarrow$ max $\{G_1.y, G_2.y, G_1.x + G_2.x + 1\}$
$G.z \leftarrow$ max $\{G.x, G.y\}$
```

Fig. 12. Algorithm for longest cardinality path in a tree.

We will also employ $\emptyset$ to signify a state that preserves consistency in the composition calculations in order to permit the case where a longest path in a composed graph is also the longest path in one of the constituent subgraphs. In other words, $\emptyset$ signifies a path with no edges (hence, always has value 0). A multiplication table is shown in Figure 11. Then, we can write the algorithm as shown in Figure 12.

2.8. Remarks

The decomposition of any tree can easily be found in $O(|V|)$ time, and it will have $O(|V|)$ size. Each algorithm in this section has $O(1)$ equivalence classes, and hence runs in $O(|V|)$ time.

The definition and number of equivalence classes that are required to solve a given problem depend on both the graph class and the problem to be solved. Of course, the effect of graph class is demonstrated throughout in subsequent sections. For an easy example of problem-specific dependency, consider the following generalization of the well-known independent set problem. Let the distance in connected graph $G = (V, E)$ between $U \subset V$ and $W \subset V$ be the shortest (edge) length of a path from any $u \in U$ to any $w \in W$, and let a *κ-independent set* be a subset of V containing no two distinct vertices at distance κ or less. Finding a maximum-cardinality κ-independent set requires $\kappa + 2$ equivalence classes $G.c$ and $G.i$ (for $0 \leq i \leq \kappa$), where $G.c$ is the maximum cardinality κ-independent set, $G.\kappa$ denotes the maximum cardinality κ-independent set with distance at least κ to the root, and $G.i$ (for $0 \leq i < \kappa$) denotes the maximum cardinality κ-independent set at distance i to the root. Finally, the multiplication table entry for

$G_1.i, G_2.j$ is $\min\{i, j+1\}$ if $i + j \geq \kappa$ and null otherwise. The resulting algorithm may be superlinear if κ is not $O(1)$.

We end this section by noting that a host of other problems could well have been selected to represent the basic recursive method on trees. Of course, many problems are trivial on trees (maximum clique, chromatic number, etc.) and even more interesting problems such as matching, independent set, and the like are well known to be solved by fast, direct algorithms. Indeed, the coverage in this subsection is certainly not intended to promote recursive methods for solving problems on trees per se, but rather to simply demonstrate, with little overhead, the basic algorithmic process and, importantly, one that carries over to more interesting and complex graph classes. It is worth noting, however, that while many problems are or can be solved on trees using recursive techniques, some are resistant. For example, the minimum bandwidth problem remains hard on trees [Garey et al. 1978].

3. SERIES-PARALLEL GRAPHS

A graph is *series-parallel* if it has no subgraph homeomorphic to K_4 [Duffin 1965]. Defined recursively, the graph consisting of a single edge (v_1, v_2) is a series-parallel graph with distinguished terminals $l = v_1$ and $r = v_2$. Now, let (G, l, r) denote a series-parallel graph G with terminal vertices l and r. A *series operation* $(G_1, l_1, r_1) \odot_s (G_2, l_2, r_2)$ forms a series-parallel graph by identifying r_1 with l_2; the terminals of the new graph are l_1 and r_2. A *parallel operation* $(G_1, l_1, r_1) \odot_p (G_2, l_2, r_2)$ forms a series-parallel graph by identifying l_1 with l_2 and r_1 with r_2; the terminals of the new graph are l_1 and r_1. Last, a *jackknife operation* $(G_1, l_1, r_1) \odot_j (G_2, l_2, r_2)$ forms a series-parallel graph by identifying r_1 with l_2; the new terminals are l_1 and r_1 (or l_1 and l_2). Series-parallel graphs are easily recognizable and their decomposition trees can be constructed in linear time [Valdes et al. 1982].

The algorithms stated next will deal only with cardinality examples; weighted or counting versions are straightforward and follow in natural ways as previously illustrated for these extensions on trees (see Figures 4 and 8). Recall that when designing algorithms for trees, only a single point of composition involving constituent subtrees was relevant, but now, series-parallel graphs have two such points: the terminal vertices. Here, we have relativized these as *left* and *right*. For brevity we have omitted the multiplication table for the jackknife operation in some of the following cases, as well as explicit computational statements from some tables altogether. The intent, however, is that when such refinements are adopted, there should be no diminution of clarity.

3.1. Algorithm for Minimum Cardinality Vertex Cover in a Series-Parallel Graph

From the recursive structure of series-parallel graphs and upon observing that a vertex must be either included in or expressly excluded from any admissible cover, we can immediately state the following equivalence classes; the similarity with the constructions for the vertex cover problem on trees should be obvious.

$G.a$ = min cardinality vertex cover containing both *left* $[G]$ and *right* $[G]$
$G.b$ = min cardinality vertex cover containing *left* $[G]$ but not *right* $[G]$
$G.c$ = min cardinality vertex cover containing *right* $[G]$ but not *left* $[G]$
$G.d$ = min cardinality vertex cover containing neither *left* $[G]$ nor *right* $[G]$
$G.e$ = min cardinality vertex cover

There are three multiplication tables corresponding to series, parallel, and jackknife operations, as shown in Figure 13. The explicit algorithm is shown in Figure 14. Note

$\odot_s$	**a**	**b**	**c**	**d**
a	a	b		
b			a	b
c	c	d		
d			c	d

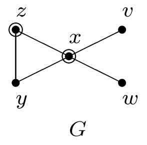
$\odot_p$	**a**	**b**	**c**	**d**
a	a			
b		b		
c			c	
d				d

$\odot_j$	**a**	**b**	**c**	**d**
a	a	a		
b			b	b
c	c	c		
d			d	d

Fig. 13. Multiplication tables for vertex cover in a series-parallel graph.

```
if $|E|$=1 then
    $[G.a, G.b, G.c, G.d] \leftarrow [2, 1, 1, \infty]$
else if $G = G_1 \odot_s G_2$ then
    $G.a \leftarrow$ min $\{G_1.a + G_2.a - 1, G_1.b + G_2.c\}$
    $G.b \leftarrow$ min $\{G_1.a + G_2.b - 1, G_1.b + G_2.d\}$
    $G.c \leftarrow$ min $\{G_1.c + G_2.a - 1, G_1.d + G_2.c\}$
    $G.d \leftarrow$ min $\{G_1.c + G_2.b - 1, G_1.d + G_2.d\}$
else if $G = G_1 \odot_p G_2$ then
    $G.a \leftarrow G_1.a + G_2.a - 2$
    $G.b \leftarrow G_1.b + G_2.b - 1$
    $G.c \leftarrow G_1.c + G_2.c - 1$
    $G.d \leftarrow G_1.d + G_2.d$
else if $G = G_1 \odot_j G_2$ then
    $G.a \leftarrow G_1.a$ + min $\{G_2.a, G_2.b\}$ $-$ 1
    $G.b \leftarrow G_1.b$ + min $\{G_2.c, G_2.d\}$
    $G.c \leftarrow G_1.c$ + min $\{G_2.a, G_2.b\}$ $-$ 1
    $G.d \leftarrow G_1.d$ + min $\{G_2.c, G_2.d\}$
$G.e \leftarrow$ min $\{G.a, G.b, G.c, G.d\}$
```

Fig. 14. Algorithm for minimum cardinality vertex cover in a series-parallel graph.



Subgraph	
$G_1 = (z, y)$	$[2, 1, 1, \infty]$
$G_2 = (y, x)$	$[2, 1, 1, \infty]$
$G_3 = (z, x)$	$[2, 1, 1, \infty]$
$G_4 = (x, v)$	$[2, 1, 1, \infty]$
$G_5 = (x, w)$	$[2, 1, 1, \infty]$
$G_6 = G_1 \odot_s G_2$	$[2, 2, 2, 1]$
$G_7 = G_6 \odot_p G_3$	$[2, 2, 2, \infty]$
$G_8 = G_7 \odot_j G_4$	$[2, 3, 2, \infty]$
$G_9 = G_8 \odot_j G_5$	$[2, 4, 2, \infty]$

Fig. 15. Minimum cardinality vertex cover in a series-parallel graph.

that subtraction of values 1 and 2 in the respective computational statements avoids multiple counting when terminal vertices are merged.

3.1.1. Example. The algorithm of Figure 14 is illustrated on the series-parallel graph G shown in Figure 15. Vertices are labelled in G as indicated, and to the right the explicit computation is summarized according to the decomposition tree shown in Figure 16 (note that terminals are identified by circled vertices). The 4-tuples exhibit values for [*G.a, G.b, G.c, G.d*], and from the last computation it follows that either *G.a* or *G.c* produces an optimum. In the first case, the cover consists of $\{x, z\}$ while in the second, we have $\{x, y\}$. As a check for consistency, note that the value

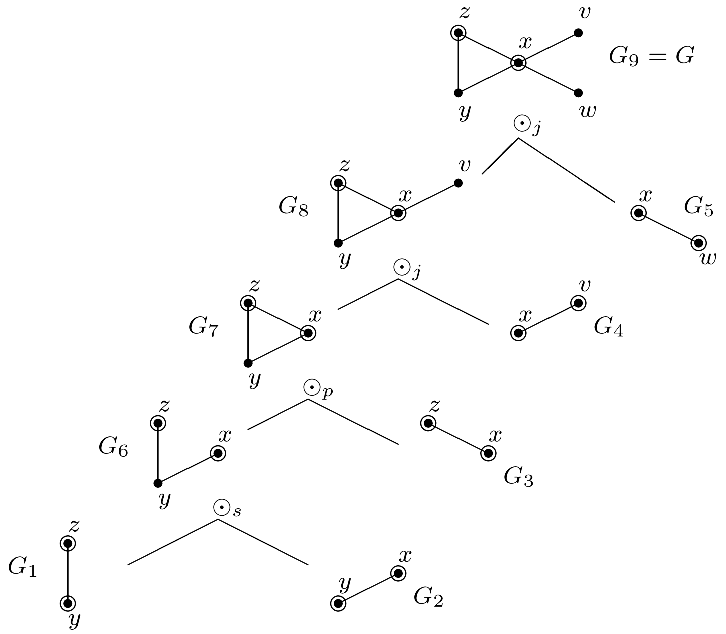


Fig. 16. Graph composition using series, parallel, and jackknife operations.

$\odot_s$	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**
i	i	j			m				
j			i	j		i	m	j	m
k	k	l			o				
l			k	l		k	o	l	o
m			i	j			m		
n	n	p			q				
o			k	l			o		
p			n	p		n	q	p	q
q			n	p			q		

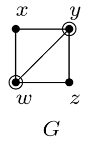
$\odot_p$	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**
i	i								
j		j			j				
k			k			k			
l				l			l	l	l
m		j			m				
n			k			n			
o				l			o	l	o
p				l			l	p	p
q				l			o	p	q

Fig. 17. Multiplication tables for dominating set in a series-parallel graph.

$G.c = 4$ correctly indicates that if vertex x is assumed excluded from any cover, then the only admissible outcome is $\{v, w, y, z\}$. Similarly, if neither x nor z can be included in any cover than there is no admissible solution at all which is revealed by $G.d = \infty$.

3.2. Algorithm for Minimum-Cardinality Dominating Set in a Series-Parallel Graph

The dominating set problem was introduced previously. Here we specify appropriate classes that are suitable for solving the problem on series-parallel graphs. The corresponding multiplication tables are shown in Figure 17.



Subgraph	
$G_1 = (w, x)$	$[2, 1, 1, \infty, \infty, \infty, \infty, \infty, 0]$
$G_2 = (x, y)$	$[2, 1, 1, \infty, \infty, \infty, \infty, \infty, 0]$
$G_3 = (w, y)$	$[2, 1, 1, \infty, \infty, \infty, \infty, \infty, 0]$
$G_4 = (w, z)$	$[2, 1, 1, \infty, \infty, \infty, \infty, \infty, 0]$
$G_5 = (z, y)$	$[2, 1, 1, \infty, \infty, \infty, \infty, \infty, 0]$
$G_6 = G_1 \odot_s G_2$	$[2, 2, 2, 1, 1, 1, \infty, \infty, \infty]$
$G_7 = G_6 \odot_p G_3$	$[2, 1, 1, 1, \infty, \infty, \infty, \infty, \infty]$
$G_8 = G_4 \odot_s G_5$	$[2, 2, 2, 1, 1, 1, \infty, \infty, \infty]$
$G_9 = G_7 \odot_p G_8$	$[2, 1, 1, 2, \infty, \infty, \infty, \infty, \infty]$

Fig. 18. Minimum cardinality dominating set in a series-parallel graph.

$G.i$ = min dominating set with both *left* $[G]$ and *right* $[G]$
$G.j$ = min dominating set with *left* $[G]$ but not *right* $[G]$
$G.k$ = min dominating set with *right* $[G]$ but not *left* $[G]$
$G.l$ = min dominating set with neither *left* $[G]$ nor *right* $[G]$
$G.m$ = min almost-dominating set with *left* $[G]$ (only *right* $[G]$ undominated)
$G.n$ = min almost-dominating set with *right* $[G]$ (only *left* $[G]$ undominated)
$G.o$ = min almost-dominating set without *left* $[G]$ (only *right* $[G]$ undominated)
$G.p$ = min almost-dominating set without *right* $[G]$ (only *left* $[G]$ undominated)
$G.q$ = min almost-dominating set (only *left* $[G]$ and *right* $[G]$ undominated)
$G.r$ = min dominating set

We have not taken space to produce the jackknife table; its construction is left to the reader. Explicit computations related to the operations $\odot_s$, $\odot_p$, and $\odot_j$ follow from the entries in the tables in the fashion demonstrated thus far; initialization when $|E| = 1$ requires that

$$[G.i,\ G.j,\ G.k,\ G.l,\ G.m,\ G.n,\ G.o,\ G.p,\ G.q] \leftarrow [2, 1, 1, \infty, \infty, \infty, \infty, \infty, 0].$$

Upon completing the computation, a minimum cardinality dominating set is determined by $G.r \leftarrow \min\{G.i,\ G.j,\ G.k,\ G.l\}$.

3.2.1. Example. The algorithm implied by the tables in Figure 17 is applied to the graph G shown in Figure 18. As before, the constituent subgraphs are indicated where $G_9 = G$; the computation is summarized by the 9-tuples on the extreme right. We leave the explicit construction of G's decomposition tree to the reader. Interpreting the solution is easy. A smallest dominating set for the graph shown has size 1, following values from $G_9.j$ and $G_9.k$, and accordingly yields either $\{w\}$ or $\{y\}$ as the outcome.

3.3. Algorithm for Minimum-Cardinality Maximal Matching in a Series-Parallel Graph

A *matching* in a graph $G = (V, E)$ is a subset of edges $M \subseteq E$ with the property that no two edges in the subset are incident to a common vertex. A matching $M \subseteq E$ is maximal if there exists no proper superset $N \supset M$ that is also a matching. For series-parallel graphs the following classes can be formed.

$G.a$ = min maximal matching with both *left* $[G]$ and *right* $[G]$ saturated
$G.b$ = min maximal matching with *left* $[G]$ saturated and *right* $[G]$ free
$G.C$ = min maximal matching with *right* $[G]$ saturated and *left* $[G]$ free

$\odot_s$	**A**	**B**	**C**	**D**	**E**	**F**	**H**	**I**	**J**	**K**
A			A	B		A	E	B	E	B
B	A	B	A	B	E		E			E
C			C	D		C	H	D	H	D
D	C	D	C	D	H		H			H
E	A	B			E					
F			F	I		F	J	I	J	I
H	C	D			H					
I	F	I	F	I	J		J			J
J	F	I			J					
K	C	D	F	I	H		J			J

$\odot_p$	**A**	**B**	**C**	**D**	**E**	**F**	**H**	**I**	**J**	**K**
A				A			A	A	A	A
B			A	B		A	E	B	E	B
C		A		C	A		C	F	F	C
D	A	B	C	D	E	F	H	I	J	K
E			A	E		A	E	E	E	E
F		A		F	A		F	F	F	F
H	A	E	C	H	E	F	H	J	J	H
I	A	B	F	I	E	F	J	I	J	I
J	A	E	F	J	E	F	J	J	J	J
K	A	B	C	K	E	F	H	I	J	K

Fig. 19. Multiplication tables for min-max matching in a series-parallel graph.

$G.D$ = min maximal matching with both *left* $[G]$ and *right* $[G]$ free
$G.E$ = min almost-maximal matching where *left* $[G]$ saturated and *right* $[G]$ needs edge
$G.F$ = min almost-maximal matching where *right* $[G]$ saturated and *left* $[G]$ needs edge
$G.H$ = min almost-maximal matching where *left* $[G]$ free and *right* $[G]$ needs edge
$G.I$ = min almost-maximal matching where *right* $[G]$ free and *left* $[G]$ needs edge
$G.J$ = min almost-maximal matching where both *left* $[G]$ and *right* $[G]$ need edges
$G.K$ = min almost-maximal matching where either *left* $[G]$ or *right* $[G]$ needs edge
$G.L$ = min maximal matching

The corresponding multiplication tables are shown in Figure 19. The construction of the jackknife table is left to the reader. Initialization when $|E| = 1$ requires that

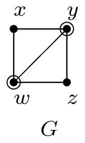
$$[G.A,\ G.B,\ G.C,\ G.D,\ G.E,\ G.F,\ G.H,\ G.I,\ G.J,\ G.K]$$
$$\leftarrow [1, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, 0].$$

Upon completing the computation, a minimum cardinality maximal matching is obtained by $G.L \leftarrow \min\{G.A,\ G.B,\ G.C,\ G.D\}$.

3.3.1. Example. In Figure 20 we reuse the same graph from the previous example to demonstrate the application of the algorithm implied by the tables of Figure 19. The final 10-tuple shown in Figure 20 indicates that a smallest maximal matching consists of a single edge, which is (w, y).

3.4. Algorithm for Hamiltonian Cycle and Hamiltonian Path in a Series-Parallel Graph

A *Hamiltonian cycle* in a graph is a cycle that includes each vertex exactly once, and a *Hamiltonian path* is a path that includes each vertex exactly once. Now, define the



Subgraph	
$G_1 = (w, x)$	$[1, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, 0]$
$G_2 = (x, y)$	$[1, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, 0]$
$G_3 = (w, y)$	$[1, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, 0]$
$G_4 = (w, z)$	$[1, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, 0]$
$G_5 = (z, y)$	$[1, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, 0]$
$G_6 = G_1 \odot_s G_2$	$[\infty, 1, 1, \infty, \infty, \infty, \infty, \infty, 0, \infty]$
$G_7 = G_6 \odot_p G_3$	$[1, 1, 1, \infty, \infty, \infty, \infty, \infty, 0, \infty]$
$G_8 = G_4 \odot_s G_5$	$[\infty, 1, 1, \infty, \infty, \infty, \infty, \infty, 0, \infty]$
$G_9 = G_7 \odot_p G_8$	$[1, \infty, \infty, \infty, \infty, \infty, \infty, \infty, 0, \infty]$

Fig. 20. Minimum cardinality maximal matching in a series-parallel graph.

$\odot_s$	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**
Q									
R		R	S			V	X	X	V
S							X		V
T		T	U						
U									
V		X		X					
W		W							
X		X							
Y		W		W					

$\odot_p$	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**
Q									Q
R		Q				T	S	U	R
S						U			S
T							U		T
U									U
V		T	U				X		V
W		S		U		X			W
X		U							X
Y	Q	R	S	T	U	V	W	X	Y

Fig. 21. Multiplication tables for Hamiltonian cycle and path in a series-parallel graph.

following classes, each of which corresponds to a Boolean value.

G.Q = Hamiltonian cycle exists
G.R = Hamiltonian path with endpoints at both *left* [*G*] and *right* [*G*]
G.S = Hamiltonian path with endpoint at *left* [*G*] but not *right* [*G*]
G.T = Hamiltonian path with endpoint at *right* [*G*] but not *left* [*G*]
G.U = Hamiltonian path with endpoint at neither *left* [*G*] nor *right* [*G*]
G.V = almost-Hamiltonian path with endpoint at *left* [*G*] (lacks *right* [*G*])
G.W = almost-Hamiltonian path with endpoint at *right* [*G*] (lacks *left* [*G*])
G.X = two disjoint paths, one ending at *left* [*G*] and one at *right* [*G*]
G.Y = the graph only contains terminal vertices
G.Z = Hamiltonian path exists

The corresponding multiplication tables are shown in Figure 21. Again, the construction of the jackknife table is left to the reader. Initialization when $|E| = 1$ requires that

$$[G.Q,\ G.R,\ G.S,\ G.T,\ G.U,\ G.V,\ G.W,\ G.X,\ G.Y] \leftarrow$$
$$[\mathit{false},\ \mathit{true},\ \mathit{false},\ \mathit{false},\ \mathit{false},\ \mathit{false},\ \mathit{false},\ \mathit{false},\ \mathit{true}].$$

Upon completing the computation, the existence of a Hamiltonian path can be determined by computing $G.Z \leftarrow G.R \vee G.S \vee G.T \vee G.U$.

Use of the jackknife operation always produces a graph that is not biconnected, and no such graph can have a Hamiltonian cycle. Accordingly, if the aim is deciding the existence of a Hamiltonian cycle in the graph, then it is safe to neglect the jackknife operation. On the other hand, if the aim is deciding the existence of a Hamiltonian path, then the jackknife operation is relevant.

3.5. Remarks

Given any series-parallel graph, its decomposition tree can be found in $O(|V|)$ time [Valdes et al. 1982], and it will have $O(|V|)$ size. Each algorithm in this section has $O(1)$ equivalence classes, and thus runs in $O(|V|)$ time.

There is a rich literature pertaining to the solution of problems on series-parallel graphs [Hare et al. 1987; Richey 1985; Wimer 1987; Wimer and Hedetniemi 1988; Wimer et al. 1985]. By any measure, the class is well studied in general and includes a host of interesting special outcomes. For example, a series-parallel graph has at most one Hamiltonian cycle [Syslo 1983], which implies that solving the *traveling salesman problem* on series-parallel graphs reduces to deciding Hamiltonicity. Also, a series-parallel graph has chromatic number 3 if it is not bipartite; otherwise it has chromatic number 2 (because it has at least one edge). Hence, the chromatic number for a series-parallel graph can be determined in linear time by using depth-first search to simply test for the existence of an odd cycle. There also exist problems that are easy on trees but hard on series-parallel graphs. Notable is a *multiple Steiner subgraph problem* [Richey 1985] that is trivial on trees but $\mathcal{NP}$-complete on series-parallel graphs. Another example is *tree subgraph isomorphism* [Radin and Parker 1986].

4. *K*-TERMINAL GRAPHS

Viewed from the perspective of their recursive constructions, it is evident that trees always employ a single terminal vertex (the root), while series-parallel graphs always employ at most two such terminals (the left and the right terminals). Accordingly, the classes of trees and series-parallel graphs can be referred to as “1-terminal” and “2-terminal” recursive graph classes, respectively. Indeed, early work pertaining to generalizations of these fairly uncomplicated recursive structures (especially early work extending series-parallel graphs) employed precisely this nomenclature by introducing so-called *k-terminal recursive graph classes* [Wimer 1987; Borie 1988].

Though now far less prominent than more modern references such as treewidth, the basic recursive graph template afforded by the k-terminal perspective still has merit in expositions such as this survey, especially in terms of providing intuition and a unifying framework for many of the subsequent, more contemporary classes.

A *k-terminal graph* $G = (V,T,E)$ has a vertex set V, an edge set E, and a set of distinguished terminals $T = \{t_1, t_2, \ldots, t_{|T|}\} \subseteq V$, where $|T| \le k$. A *k-terminal recursively structured graph class* $C(B, R)$ is specified by base graphs B and a finite rule set $R = \{f_1, f_2, \ldots, f_n\}$, where each f_i is a *recursive composition operation*. Typically, for some k, B is the set of connected k-terminal graphs (V, T, E) with $V = T$, that is, all vertices are terminals.

The notion of *composition* typically permitted in the context of k-terminal graphs can be described in a more formal way. For $1 \le i \le m$, let $G_i = (V_i, T_i, E_i)$, such that $V_1, \ldots, V_m$ are mutually disjoint vertex sets. Let $G = (V, T, E)$ as well. Then a *valid vertex mapping* is a function $f : \cup_{1 \le i \le m} V_i \to V$ such that:

—vertices from the same G_i remain distinct: $v_1 \in V_i, v_2 \in V_i$, $f(v_1) = f(v_2)$ implies $v_1 = v_2$;

—only (not necessarily all) terminals map to terminals: $v \in V_i$, $f(v) \in T$ implies $v \in T_i$;

—only terminals can merge: $v_1 \in V_{i_1}, v_2 \in V_{i_2}, i_1 \ne i_2$, $f(v_1) = f(v_2)$ implies $v_1 \in T_{i_1}, v_2 \in T_{i_2}$; and

—edges are preserved: $(\exists i)((v_1, v_2) \in E_i)$ iff $(f(v_1), f(v_2)) \in E$.

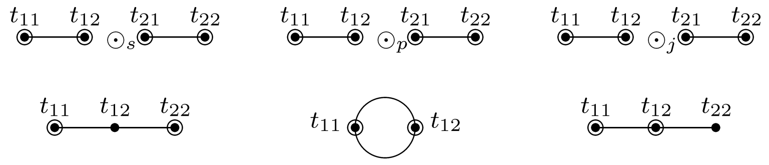


Fig. 22. Series, parallel, and jackknife composition for 2-terminal graphs.

Each composition operation must correspond to a valid vertex mapping. If f is a valid vertex mapping, the corresponding m-ary composition operation (denoted by f) is generally written $f(G_1, \ldots, G_m) = G$. In other words, applying operation f to graphs $G_1, \ldots, G_m$ yields graph G.

To illustrate, consider a case with $k = 2$; the base graphs are edges with both vertices specified as terminals, that is, $B = \{K_2\}$. Let the composition rule set R consist of three binary operations f_1, f_2, f_3 relativized by $\{\odot_s, \odot_p, \odot_j\}$ and that operate on 2-terminal graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Identifying the terminal vertex sets for G_1 and G_2 as $\{t_{11}, t_{12}\}$, and $\{t_{21}, t_{22}\}$, respectively, we have the following.

—The *series* operation $\odot_s$ identifies t_{12} with t_{21}, and constructs a new 2-terminal graph $(V_1 \cup V_2, E_1 \cup E_2)$ with terminal set $\{t_{11}, t_{22}\}$.

—The *parallel* operation $\odot_p$ identifies t_{11} with t_{21}, and also t_{12} with t_{22}, and constructs a new 2-terminal graph $(V_1 \cup V_2, E_1 \cup E_2)$ with terminal set $\{t_{11}, t_{12}\}$ (equivalently, $\{t_{21}, t_{22}\}$).

—The *jackknife* operation $\odot_j$ identifies t_{12} with t_{21}, and constructs a new 2-terminal graph $(V_1 \cup V_2, E_1 \cup E_2)$ with terminal set $\{t_{11}, t_{12}\}$ (equivalently, $\{t_{11}, t_{21}\}$).

Figure 22 illustrates the three composition rules (recall that terminals are identified by circled vertices). A complete composition was previously demonstrated in Figure 16. Indeed, the stated B and R generate precisely the class of series-parallel graphs.

4.1. Algorithms

Consistent with the theme alluded to earlier, it is important to determine a decomposition tree for a given k-terminal graph; that is, to understand how the graph can be constructed via a legal application of a set of composition rules applied to a fixed base graph set. In general, when we are able to do this for some recursive class, it will provide a way to recognize membership in the given class. Naturally, it is important that this test be applied efficiently. (It is not necessary for every member of a recursive graph class to have a unique decomposition tree; if a particular member of a k-terminal graph class has multiple decomposition trees, then it will be sufficient to use any one of these.)

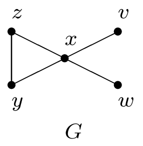
4.1.1. Example. A *proper vertex coloring* for a graph G is an assignment of colors $c_1, c_2, \ldots, c_t$ to the vertices of G such that no pair of adjacent vertices is assigned the same color. If at most k colors are needed to properly color the vertices in some G, then we say that G is *k-colorable*. Deciding 2-colorability or, equivalently, whether or not a graph is bipartite is not interesting for any graph class, since there is an obvious, fast algorithm for deciding the matter; however, deciding 3-colorability is hard [Karp 1972].

Let us count the number of valid 3-colorings of a 2-terminal graph. Denote the colors as **1**,**2**,**3**. There are $|\{\mathbf{1},\mathbf{2},\mathbf{3}\}|^2 = 9$ equivalence classes. Colorings in a class are equivalent in

$\odot_s$	**11**	**12**	**13**	**21**	**22**	**23**	**31**	**32**	**33**
11	**11**	**12**	**13**						
12				**11**	**12**	**13**			
13							**11**	**12**	**13**
21	**21**	**22**	**23**						
22				**21**	**22**	**23**			
23							**21**	**22**	**23**
31	**31**	**32**	**33**						
32				**31**	**32**	**33**			
33							**31**	**32**	**33**

$\odot_j$	**11**	**12**	**13**	**21**	**22**	**23**	**31**	**32**	**33**
11	**11**	**11**	**11**						
12				**12**	**12**	**12**			
13							**13**	**13**	**13**
21	**21**	**21**	**21**						
22				**22**	**22**	**22**			
23							**23**	**23**	**23**
31	**31**	**31**	**31**						
32				**32**	**32**	**32**			
33							**33**	**33**	**33**

Fig. 23. 3-colorings of a 2-terminal graph.



Subgraph	
$G_1 = (z, y)$	$[0, 1, 1, 1, 0, 1, 1, 1, 0]$
$G_2 = (y, x)$	$[0, 1, 1, 1, 0, 1, 1, 1, 0]$
$G_3 = (z, x)$	$[0, 1, 1, 1, 0, 1, 1, 1, 0]$
$G_4 = (x, v)$	$[0, 1, 1, 1, 0, 1, 1, 1, 0]$
$G_5 = (x, w)$	$[0, 1, 1, 1, 0, 1, 1, 1, 0]$
$G_6 = G_1 \odot_s G_2$	$[2, 1, 1, 1, 2, 1, 1, 1, 2]$
$G_7 = G_6 \odot_p G_3$	$[0, 1, 1, 1, 0, 1, 1, 1, 0]$
$G_8 = G_7 \odot_j G_4$	$[0, 2, 2, 2, 0, 2, 2, 2, 0]$
$G_9 = G_8 \odot_j G_5$	$[0, 4, 4, 4, 0, 4, 4, 4, 0]$

Fig. 24. Counting the 3-colorings of a 2-terminal graph.

$G_8 \odot_j G_5$	G_5	0	1	1	1	0	1	1	1	0
G_8	$\odot_j$	**11**	**12**	**13**	**21**	**22**	**23**	**31**	**32**	**33**
0	**11**	**11** 0	**11** 0	**11** 0						
2	**12**				**12** 2	**12** 0	**12** 2			
2	**13**							**13** 2	**13** 2	**13** 0
2	**21**	**21** 0	**21** 2	**21** 2						
0	**22**				**22** 0	**22** 0	**22** 0			
2	**23**							**23** 2	**23** 2	**23** 0
2	**31**	**31** 0	**31** 2	**31** 2						
2	**32**				**32** 2	**32** 0	**32** 2			
0	**33**							**33** 0	**33** 0	**33** 0

Fig. 25. Detailed computation at final step during 3-coloring.

their compatibility with colorings of other subgraphs. In lexicographic order, the classes are denoted **11,12,13,21,22,23,31,32,33**. For the base graphs, the 9-tuple of counting values is [0, 1, 1, 1, 0, 1, 1, 1, 0]. As done previously, we express the composition formulas in the more compact form of multiplication tables, as shown in Figure 23. Rows correspond to G_α and columns correspond to G_β for the composition $G_\alpha \odot G_\beta$. We do not take the space to show the parallel table, as its only nonempty entries are the values **11,12,**. . . ,**33** along the main diagonal.

An entire computation is summarized in Figure 24. The detailed computation at the final step of the composition is given in Figure 25. The smaller case digit in each cell is

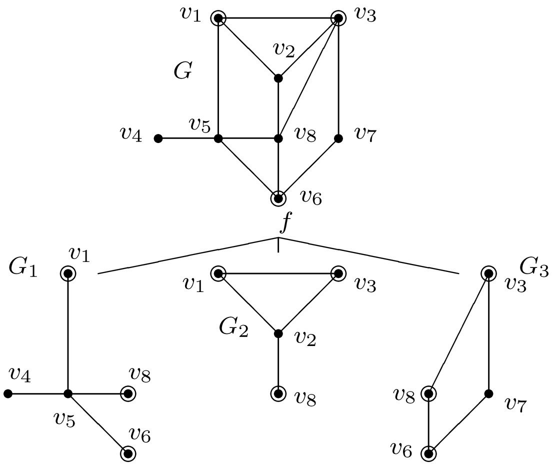


Fig. 26. A 3-terminal graph composition.

the contribution of that cell towards the final 9-tuple [0, 4, 4, 4, 0, 4, 4, 4, 0]. Therefore the total number of valid 3-colorings is 24 for this graph.

4.2. More about *k*-Terminal Graphs

To facilitate a broader understanding, we generalize by considering a case $k \geq 2$ for some k-terminal class and where the composition is other than binary. Naturally, some complication may be introduced with such generalization, but often this is more a matter of technical detail rather than structural. To illustrate, consider a 3-terminal class with composition f shown in Figure 26. We will not take space to state an explicit algorithm nor provide a full solution to a problem; however, the partial illustration shown should suffice to convey the basic scheme in a more general context.

4.2.1. Example. For ease, we use the vertex cover problem to demonstrate. Now, since any composed subgraph has, in this case, at most $k = 3$ terminal vertices, it is sufficient that the computation record optimal values for $2^k = 8$ attributes. Denoting the terminal vertices generically by t_1, t_2, and t_3, we can define these attributes as follows.

$G.\emptyset =$ size of smallest vertex cover containing no terminals.
$G.t_1 =$ size of smallest vertex cover containing t_1 but not t_2 or t_3.
$G.t_2 =$ size of smallest vertex cover containing t_2 but not t_1 or t_3.
$G.t_3 =$ size of smallest vertex cover containing t_3 but not t_1 or t_2.
$G.t_1t_2 =$ size of smallest vertex cover containing t_1 and t_2 but not t_3.
$G.t_1t_3 =$ size of smallest vertex cover containing t_1 and t_3 but not t_2.
$G.t_2t_3 =$ size of smallest vertex cover containing t_2 and t_3 but not t_1.
$G.t_1t_2t_3 =$ size of smallest vertex cover containing t_1 and t_2 and t_3.

Now, the explicit computation for the ternary composition shown in Figure 26 can be formulated as shown in Figure 27. Again, observe that the constants that are subtracted throughout simply adjust for multiple counting when terminal vertices are merged.

Assuming the correct 8-tuples are known for G_1, G_2, and G_3, then upon applying the specified composition formulas, the respective values for the 8-tuple identified with

$$\begin{aligned}
&G.\emptyset \leftarrow \min\{G_1.\emptyset + G_2.\emptyset + G_3.\emptyset, G_1.v_8 + G_2.v_8 + G_3.v_8 - 2\}\\
&G.v_1 \leftarrow \min\{G_1.v_1 + G_2.v_1 + G_3.\emptyset - 1, G_1.v_1v_8 + G_2.v_1v_8 + G_3.v_8 - 3\}\\
&G.v_3 \leftarrow \min\{G_1.\emptyset + G_2.v_3 + G_3.v_3 - 1, G_1.v_8 + G_2.v_3v_8 + G_3.v_3v_8 - 3\}\\
&G.v_6 \leftarrow \min\{G_1.v_6 + G_2.\emptyset + G_3.v_6 - 1, G_1.v_6v_8 + G_2.v_8 + G_3.v_6v_8 - 3\}\\
&G.v_1v_3 \leftarrow \min\{G_1.v_1 + G_2.v_1v_3 + G_3.v_3 - 2, G_1.v_1v_8 + G_2.v_1v_3v_8 + G_3.v_3v_8 - 4\}\\
&G.v_1v_6 \leftarrow \min\{G_1.v_1v_6 + G_2.v_1 + G_3.v_6 - 2, G_1.v_1v_6v_8 + G_2.v_1v_8 + G_3.v_6v_8 - 4\}\\
&G.v_3v_6 \leftarrow \min\{G_1.v_6 + G_2.v_3 + G_3.v_3v_6 - 2, G_1.v_6v_8 + G_2.v_3v_8 + G_3.v_3v_6v_8 - 4\}\\
&G.v_1v_3v_6 \leftarrow \min\{G_1.v_1v_6 + G_2.v_1v_3 + G_3.v_3v_6 - 3, G_1.v_1v_6v_8 + G_2.v_1v_3v_8 + G_3.v_3v_6v_8 - 5\}
\end{aligned}$$

Fig. 27. Formulas for minimum vertex cover in a 3-terminal graph.

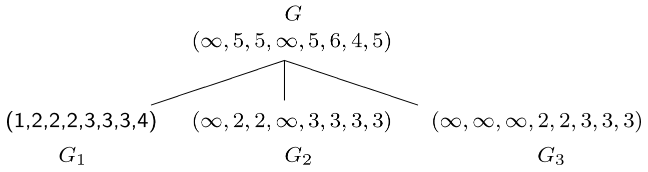

G
$(\infty, 5, 5, \infty, 5, 6, 4, 5)$
(1,2,2,2,3,3,3,4)
G_1
$(\infty, 2, 2, \infty, 3, 3, 3, 3)$
G_2
$(\infty, \infty, \infty, 2, 2, 3, 3, 3)$
G_3

Fig. 28. A 3-terminal computation for vertex cover.

G can be computed. The computation is summarized in Figure 28. For example, the minimum cover in G has size 4 corresponding to the value of $G.v_3v_6$; the relevant cover is given by the set of vertices $\{v_2, v_3, v_5, v_6\}$. Similarly, the value $G.v_6 = \infty$ correctly signifies that no admissible cover that excludes vertices v_1 and v_3 can exist in G.

4.3. Remarks

The time needed to find a decomposition tree for a k-terminal graph depends upon the particular set of k-terminal operations permitted. But in any case, the decomposition tree will have $O(|V|)$ size. Each algorithm in this section has $O(1)$ equivalence classes, so once a decomposition is known, these algorithms will run in $O(|V|)$ time.

It is important to be reminded that simply referring to a structure as a k-terminal graph is too general; in some sense, all graphs are k-terminal graphs for some particular set of k-terminal operations. However, the class is not well defined until valid composition operations are specified. Still, by including this section on general notions derived within the broad context of k-terminal graphs, our intention is to create an easy, conceptual backdrop for what follows with specific graph classes popularly identified by other names. Although only scant reference will be made to these classes in terms of their membership or association with some k-terminal family, the basic tenets of how problems are solved on these popular classes are often generally traceable to the protocol described in this section.

5. TREEWIDTH-*K* GRAPHS

The work on treewidth reported in Robertson and Seymour [1986a, 1983] is fundamental in its own right. But it has also played a key role in motivating, indeed establishing, important results in related areas in graph theory. Arguably most notable is the role played by treewidth in their work on graph minors culminating in the proof of Wagner's conjecture [Robertson and Seymour 2004].

A *tree-decomposition* of a graph $G = (V, E)$ is defined by a pair $(\{X_i | i \in I\}, T)$ where $\{X_i | i \in I\}$ is a family of subsets of V, and T is a tree with vertex set I such that:

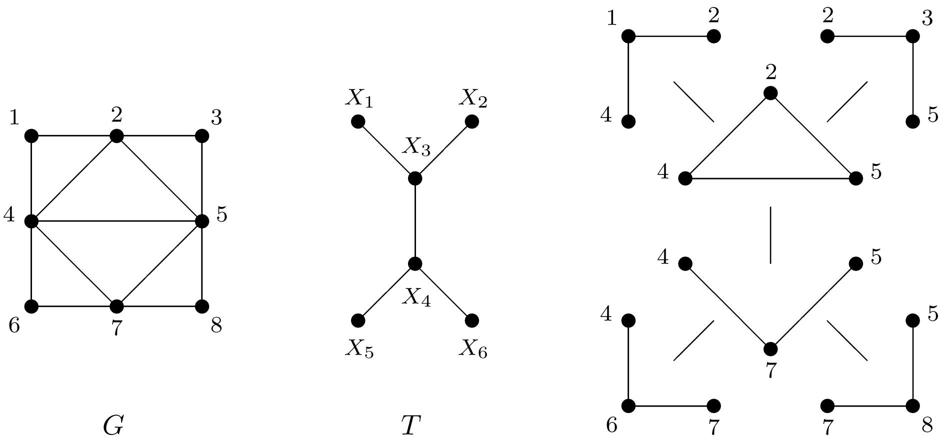


Fig. 29. Tree-decomposition of a treewidth-2 graph.

—$\bigcup_{i \in I} X_i = V$;

—for each edge $(x, y) \in E$ there is an element $i \in I$ with $x, y \in X_i$; and

—for all triples $i, j, k \in I$, if j is on the path from i to k in T, then we have that $X_i \bigcap X_k \subseteq X_j$.

Figure 29 demonstrates with an example.

Trivially, every graph G has a tree decomposition that is defined by a single vertex, representing G itself. On the other hand, we are interested in tree decompositions and hence, their graphs in which the stated X_i are small. Accordingly, the *width* of a given tree decomposition is measured as $\max_{i \in I}\{|X_i| - 1\}$. Then the *treewidth* of a graph G is the minimum width taken over all tree decompositions of G, and a graph G is said to be a *treewidth-k graph* if it has treewidth at most k. The graph in Figure 29 has treewidth 2.

Important algorithmically, it has been shown that every treewidth-k graph has a tree decomposition T where T, is a rooted binary tree [Scheffler 1989]. Recursively, we may write $(G, X) = (G_1, X_1) \otimes (G_2, X_2)$, where $X \subseteq V$ is the set of vertices of G associated with *root* $[T]$, and graphs G_1 and G_2 have tree decompositions given by the left and right subtrees of T. This is sufficient to produce linear-time dynamic programming algorithms for many problems on treewidth-k graphs, since each $|X| \leq k + 1$.

5.1. Algorithm for Minimum-Cardinality Vertex Cover in a Treewidth-*k* Graph

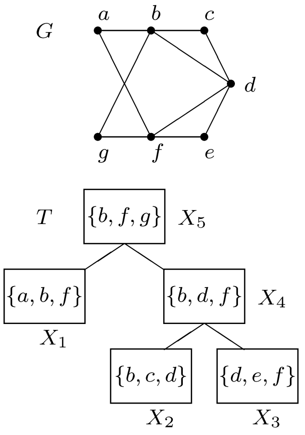
Define

$G[S]$ = min cardinality vertex cover that contains $S \subseteq X$ but not $X - S$
G.min = min cardinality vertex cover

The algorithm is shown in Figure 30. It is demonstrated on the treewidth-2 graph G shown in Figure 31. Specifically, T is the stated binary rooted tree decomposition of G, where for ease, each node in T specifies the respective X_i; each 8-tuple exhibits values for $G[S]$ for each $S \subseteq X$. However, only values smaller than ∞ within each 8-tuple are shown as the computation progresses. The minimum vertex cover has size 3, and the explicit solution is $\{b, d, f\}$.

```
if $X = V$ then
    $\forall_{S\subseteq X}\ G[S] \leftarrow$ if $\exists_{y,z\in X-S}\ (y,z)\in E$ then $\infty$ else $|S|$
else if $(G,\ X) = (G_1,\ X_1) \otimes (G_2,\ X_2)$ then
    $\forall_{S\subseteq X}\ G[S] \leftarrow$ if $\exists_{y,z\in X-S}\ (y,z)\in E$ then $\infty$
        else min $\{G_1[S_1] + G_2[S_2] - |S_1 \cap S| - |S_2 \cap S| + |S|$ :
            $S_1 \subseteq X_1,\ S_2 \subseteq X_2,\ S_1 \cap X = S \cap X_1,\ S_2 \cap X = S \cap X_2\}$
$G.min \leftarrow$ min $\{G[S] : S \subseteq X\}$
```

Fig. 30. Algorithm for minimum cardinality vertex cover in a treewidth-k graph.



Subgraph

$(G_1, X_1 = \{a, b, f\})$
$[\{a\} \rightarrow 1, \{a, b\} \rightarrow 2, \{a, f\} \rightarrow 2, \{b, f\} \rightarrow 2, \{a, b, f\} \rightarrow 3]$

$(G_2, X_2 = \{b, c, d\})$
$[\{b, c\} \rightarrow 2, \{b, d\} \rightarrow 2, \{c, d\} \rightarrow 2, \{b, c, d\} \rightarrow 3]$

$(G_3, X_3 = \{d, e, f\})$
$[\{d, e\} \rightarrow 2, \{d, f\} \rightarrow 2, \{e, f\} \rightarrow 2, \{d, e, f\} \rightarrow 3]$

$(G_4, X_4 = \{b, d, f\}) = G_2 \otimes G_3$
$[\{d\} \rightarrow 3, \{b, d\} \rightarrow 3, \{b, f\} \rightarrow 4, \{d, f\} \rightarrow 3, \{b, d, f\} \rightarrow 3]$

$(G_5, X_5 = \{b, f, g\}) = G_1 \otimes G_4$
$[\{g\} \rightarrow 5, \{b, f\} \rightarrow 3, \{b, g\} \rightarrow 5, \{f, g\} \rightarrow 5, \{b, f, g\} \rightarrow 4]$

Fig. 31. Minimum cardinality vertex cover in a treewidth-2 graph.

```
if $X = V$ then
    $\forall_{S,T\subseteq X}\ G[S,T] \leftarrow$ if $T = S \cup N_G(S)$ then $|S|$ else $\infty$
else if $(G,\ X) = (G_1,\ X_1) \otimes (G_2,\ X_2)$ then
    $\forall_{S,T\subseteq X}\ G[S,T] \leftarrow$ min $\{G_1[S_1,T_1] + G_2[S_2,T_2]\ - |S_1 \cap S| - |S_2 \cap S| + |S|$ :
        $S_1 \subseteq X_1,\ S_2 \subseteq X_2,\ S_1 \cap X = S \cap X_1,\ S_2 \cap X = S \cap X_2,$
        $T_1 \subseteq X_1,\ T_2 \subseteq X_2,\ T = X \cap (T_1 \cup T_2 \cup S \cup N_G(S))\}$
$G.min \leftarrow$ min $\{G[S,V] : S \subseteq X\}$
```

Fig. 32. Algorithm for minimum cardinality dominating set in a treewidth-k graph.

5.2. Algorithm for Minimum-Cardinality Dominating Set in a Treewidth-*k* Graph

Define:

$G[S, T]$ = min cardinality dominating or almost-dominating set that contains $S \subseteq X$ but not $X - S$, and that dominates $T \subseteq X$ but not $X - T$
G.min = min cardinality dominating set

The algorithm is shown in Figure 32. Recall that $N_G(S)$ denotes the set of neighbors of vertices S in graph G.

Assuming a tree decomposition is known, the algorithms in Figures 30 and 32 each run in $O(|V|)$ time. The next subsection discusses an algorithm adapted from Bodlaender [1990] that runs in polynomial, but not linear, time.

5.3. Algorithm for Chromatic Index in a Treewidth-*k* Graph

For a given graph G a *proper edge-coloring* is an assignment of colors to the edges of G such that no pair of adjacent edges have the same color. The smallest number of colors

```
if $X = V$ then
      compute $G.D$ by brute force // for example, $G.D = \{(1, 0, 0, 0)\}$ when $k$=2
else if $(G, X) = (G_1, X_1) \otimes (G_2, X_2)$ then
      $G.D \leftarrow$ closure($G_1.D$, $G_2.D$)
$G.index \leftarrow$ min $\{\Sigma_{S \subseteq X}\ c_S : C \in G.D\}$
```

Fig. 33. Algorithm for chromatic index in a treewidth-k graph.

```
closure($D'$, $D''$) {
      $Z \leftarrow D' \times D'' \times \{(0, \ldots, 0)\}$; // each element of $Z$ is a 3-tuple of $2^k$-tuples
      while (more tuples can be added to $Z$) do {
            choose any tuple $(C', C'', C) \in Z$;
            choose any $S' \subseteq X_1$ and $S'' \subseteq X_2$ such that $S' \cap S'' = \emptyset$ and $c'_{S'} + c''_{S''} > 0$;
            if $c'_{S'} = 0$ then $S' \leftarrow \emptyset$;
            else if $c''_{S''} = 0$ then $S'' \leftarrow \emptyset$;
            choose any $S \subseteq X$ such that $S' \cap X = S \cap X_1$ and $S'' \cap X = S \cap X_2$;
            $c'_{S'} \leftarrow \max\{0, c'_{S'} - 1\}$;
            $c''_{S''} \leftarrow \max\{0, c''_{S''} - 1\}$;
            $c_S \leftarrow c_S + 1$;
            add tuple $(C', C'', C)$ to $Z$;
      }
      $D \leftarrow \{C : ((0, \ldots, 0), (0, \ldots, 0), C) \in Z\}$;
      return $D$;
}
```

Fig. 34. Closure function for chromatic index algorithm.

which produces a proper edge coloring is called the *chromatic index* of G. Define

$c_S =$ number of colors that are incident upon $S \subseteq X$ but not $X - S$
$C = (c_X, \ldots, c_S, \ldots, c_\emptyset)$ is a valid 2^k-tuple with compatible c_S values
$G.D = \{C\}$ is the set of all valid 2^k-tuples
$G.index =$ chromatic index

The algorithm is shown in Figure 33; it invokes the closure function which is given in Figure 34.

Vizing's theorem [Vizing 1964] states that the chromatic index of any graph is either Δ or $\Delta + 1$, where Δ is the maximum vertex degree. Thus the chromatic index is at most $|V|$. Hence c_S in the chromatic index algorithm of Figure 33 is bounded as $0 \leq c_S \leq |V|$ for all $S \subseteq X$. In fact, the sum of the entries in each 2^k-tuple is at most $|E|$. So each graph has a polynomial number of valid 2^k-tuples.

Next, the size of each set Z in the closure function of Figure 34 is $O(|V|^{3 \cdot 2^k})$. Therefore the chromatic index algorithm runs in polynomial time with appropriate data structures. The running time is shown in Bodlaender [1990] to be $O(|V|^{2^{2k+2}+1})$.

A more recent improvement shows a linear-time algorithm for this chromatic index problem on treewidth-k graphs Zhou et al. [1996, 1993].

5.4. Monadic Second-Order Logic (MSOL)

Though differing structurally, the graph classes described thus far and importantly, the algorithms created, expose a consistent theme. But this is, in some natural sense, an essential expectation with recursive graph classes; the phenomenon will persist in subsequent classes presented in this article. Moreover, this consistency translates directly to the problem solving realm. Most of the problems solved thus far (variations of vertex cover, dominating set, independent set, m-vertex coloring for arbitrary fixed

$$\begin{aligned}
&P \rightarrow Q \Leftrightarrow \neg P \vee Q\\
&P \leftrightarrow Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)\\
&e_i = e_j \Leftrightarrow (\forall v_1)\ (\text{Incident}(v_1,\ e_i) \leftrightarrow \text{Incident}(v_1,\ e_j))\\
&\text{Adjacent}(v_i,\ v_j) \Leftrightarrow \neg(v_i = v_j) \wedge (\exists e_1)\ (\text{Incident}(v_i,\ e_1) \wedge \text{Incident}(v_j,\ e_1))\\
&V_1 \cup V_2 = V_3 \Leftrightarrow (\forall v_1)((v_1 \in V_1 \vee v_1 \in V_2) \leftrightarrow v_1 \in V_3)\\
&V_1 \cap V_2 = V_3 \Leftrightarrow (\forall v_1)((v_1 \in V_1 \wedge v_1 \in V_2) \leftrightarrow v_1 \in V_3)\\
&\text{Partition}(V_1,\ldots,V_m) \Leftrightarrow (V_1 \cup \ldots \cup V_m = V) \wedge \textstyle\bigwedge_{1 \le i < j \le m} (V_i \cap V_j = \emptyset)
\end{aligned}$$

Fig. 35. Some useful MSOL predicates.

$$\begin{aligned}
&\text{VertexCover}(V_1) \Leftrightarrow (\forall e_2)(\exists v_3)(v_3 \in V_1 \wedge \text{Incident}\ (v_3,\ e_2))\\
&\text{IndependentSet}(V_1) \Leftrightarrow (\forall v_2)\ (\forall v_3)\ ((v_2 \in V_1 \wedge v_3 \in V_1) \rightarrow \neg\ \text{Adjacent}(v_2,\ v_3))\\
&\text{Clique}(V_1) \Leftrightarrow (\forall v_2)\ (\forall v_3)\ ((v_2 \in V_1 \wedge v_3 \in V_1) \rightarrow \text{Adjacent}(v_2,\ v_3))\\
&\text{DominatingSet}(V_1) \Leftrightarrow (\forall v_2)\ (v_2 \in V_1 \vee (\exists v_3)\ (v_3 \in V_1 \wedge \text{Adjacent}(v_2,\ v_3))\\
&\text{VertexColoring}_m(V_1,\ldots,V_m) \Leftrightarrow \text{Partition}(V_1,\ldots,V_m) \wedge \textstyle\bigwedge_{1 \le i \le m} \text{IndependentSet}(V_i)\\
&\text{CliquePartition}_m(V_1,\ldots,V_m) \Leftrightarrow \text{Partition}(V_1,\ldots,V_m) \wedge \textstyle\bigwedge_{1 \le i \le m} \text{Clique}(V_i)\\
&\text{Matching}(E_1) \Leftrightarrow (\forall e_2)\ (\forall e_3)\ ((e_2 \in E_1 \wedge e_3 \in E_1 \wedge \neg\ (e_2 = e_3)) \rightarrow\\
&\qquad\qquad \neg(\exists v_4)\ (\text{Incident}(v_4,\ e_2) \wedge \text{Incident}(v_4,\ e_3)))\\
&\text{Connected}(E_1) \Leftrightarrow (\forall V_2)\ (\forall V_3)\ (\neg\ (\exists v_4)\ (v_4 \in V_2) \vee \neg\ (\exists v_5)\ (v_5 \in V_3) \vee\\
&\qquad\qquad (\exists v_6)\ (\neg\ (v_6 \in V_2) \wedge \neg\ (v_6 \in V_3)) \vee\\
&\qquad\qquad (\exists e_7)\ (\exists v_8)\ (\exists v_9)\ (e_7 \in E_1 \wedge v_8 \in V_2 \wedge v_9 \in V_3 \wedge\\
&\qquad\qquad \text{Incident}(v_8,\ e_7) \wedge \text{Incident}(v_9,\ e_7)))\\
&\text{HamCycle}(E_1) \Leftrightarrow \text{Connected}(E_1) \wedge (\forall v_2)\ (\exists e_3)\ (\exists e_4)\ (e_3 \in E_1 \wedge e_4 \in E_1 \wedge\\
&\qquad\qquad \neg(e_3 = e_4) \wedge \text{Incident}(v_2,\ e_3) \wedge \text{Incident}(v_2,\ e_4) \wedge\\
&\qquad\qquad (\forall e_5)\ ((e_5 \in E_1 \wedge \text{Incident}(v_2,\ e_5)) \rightarrow (e_5 = e_3 \vee e_5 = e_4)))\\
&\text{HamPath}(E_1) \Leftrightarrow \text{Connected}(E_1) \wedge (\forall v_2)\ (\exists e_3)\ (\exists e_4)\ (e_3 \in E_1 \wedge e_4 \in E_1 \wedge\\
&\qquad\qquad \text{Incident}(v_2,\ e_3) \wedge \text{Incident}(v_2,\ e_4) \wedge\\
&\qquad\qquad (\forall e_5)\ ((e_5 \in E_1 \wedge \text{Incident}(v_2,\ e_5)) \rightarrow (e_5 = e_3 \vee e_5 = e_4)) \wedge\\
&\qquad\qquad (\exists v_6)\ (\exists e_7)\ (\forall e_8)\ ((e_8 \in E_1 \wedge \text{Incident}(v_6,\ e_8)) \rightarrow e_8 = e_7))
\end{aligned}$$

Fig. 36. Graph problems expressed in MSOL.

m, matching, and Hamiltonian cycle/path problems) and many others can be solved by linear-time algorithms on recursive classes, which in turn provokes an obvious question: Is there a formalism that describes this outcome; a formalism under which the existence of linear-time algorithms is anticipated?

Monadic second-order logic (MSOL) for a graph $G = (V, E)$ is a predicate calculus language in which predicates are constructed recursively as follows. Let variables v_i denote a vertex with domain V, e_i denote an edge with domain E, V_i denote a vertex set with domain 2^V, and E_i denote an edge set with domain 2^E. MSOL contains primitive predicates such as $v_i = v_j$, Incident(v_i, e_j), $v_i \in V_j$, and $e_i \in E_j$. If P and Q are MSOL predicates then each of ($\neg$ P), (P $\wedge$ Q), and (P $\vee$ Q) is also an MSOL predicate. Finally, if P is an MSOL predicate and x is any variable, then ($\exists$x)(P) and ($\forall$x)(P) are also MSOL predicates.

Several useful MSOL predicates are shown in Figure 35. Of particular interest is the creation of legal expressions that formalize the full expression of important graph problems, such as those listed in Figure 36. This is a very short list and interested readers are directed to the literature for a more expansive compilation [Borie 1988].

Important is the following result, established independently and reported in various sources. Specifically, it has been shown that every MSOL-expressible problem can be solved in linear time on treewidth-k graphs [Borie 1988; Arnborg et al. 1991; Borie et al. 1992; Courcelle 1990]. This statement also holds for many variations of each MSOL problem including existence, minimum or maximum cardinality, minimum or maximum total weight, minimum-maximal or maximum-minimal sets, bottleneck weight, and counting. But since treewidth relates, in a parameterized sense, to many of the classes covered in this article, the statement's true power becomes most evident; if a problem is MSOL-expressible, then it would be solvable in linear time on any of a broad variety of recursive graph classes.

5.5. Remarks

Approaches to recognizing treewidth-k graphs and constructing their tree decompositions will be addressed in Section 6.

Once a problem is expressed in MSOL, a linear-time dynamic programming algorithm can be created mechanically to solve the problem on treewidth-k graphs [Arnborg et al. 1991; Borie et al. 1992]. Though interesting, this is an existential outcome and the algorithms generated accordingly do not produce practical procedures directly. In other words, these automatically generated algorithms will run in linear time with respect to the graph size, but the hidden constants may be superexponential in parameter k.

The proof of Wagner's conjecture [Robertson and Seymour 2004] guarantees for fixed k that treewidth-k graphs are characterizable by a finite obstruction set. Also, for each fixed graph H there exists an MSOL expression for stating that an input graph G contains minor H. Therefore, for each k an MSOL expression exists for stating that G has treewidth at most k. However, this result is nonconstructive; indeed, the obstruction set and corresponding MSOL expression are currently only known when $k \leq 3$.

For some problems, an MSOL expression cannot be written and so a linear-time algorithm cannot be derived automatically. In some of these cases it might still be possible to develop a linear-time algorithm via an extension to MSOL [Arnborg et al. 1993, 1991; Borie et al. 1992; Courcelle and Mosbash 1993].

In other cases it may be possible to develop a polynomial-time algorithm. Polynomiality is achieved by constructing a polynomial-size data structure that corresponds to each node in the tree decomposition. This was seen previously for the chromatic index algorithm of Figure 33. Additional polynomial-time algorithms for problems on treewidth-k graphs may be found in Borie [1995], Courcelle et al. [2001], and Telle and Proskurowski [1997, 1993] as well as the generalized coloring algorithms from Isobe et al. [1999], Ito et al. [2003], Kashem et al. [2000], and Zhou et al. [2000a, 2000b].

Trees are treewidth-1 graphs, and series-parallel graphs are treewidth-2 graphs. For every k, treewidth-k graphs are equivalent to the class of partial k-trees, which will be discussed in Section 7; other subclasses of treewidth-k graphs will then be discussed in subsequent sections. Also, treewidth-k graphs are precisely the subgraphs of chordal graphs with maximum clique size at most $k + 1$.

6. PRACTICAL ISSUES

We insert this section as a break from the rather theoretically-oriented catalog presentation of recursive graph classes to briefly discuss some more practical considerations: applying these graph classes to solve real-world problems, converting MSOL expressions into efficient algorithms, and quickly determining a graph's treewidth.

6.1. Applications

Treewidth-k graphs are useful in a variety of real-world applications. Among these are the following examples.

—Control-flow graphs of structured computer programs generally have small treewidth [Thorup 1998]. For example, the control-flow graph of a Pascal program always has treewidth at most 3, or at most 2 if short-circuiting is not used for evaluating Boolean expressions. C programs without goto statements always yield treewidth at most 6, and often at most 3. The same holds for Ada programs without labeled loops, and for Java programs without labeled break or continue statements. Important is that such results permit compilers to perform various analysis and optimization tasks efficiently, for example, register allocation [Bodlaender et al. 1998].

—Electrical circuits are often formed by connecting components using series and parallel operations. Such circuits are precisely treewidth-2 graphs, and there are well-known formulas for computing the total resistance, capacitance, or inductance across these circuits using the series-parallel decomposition. It is also sometimes useful to build circuits that correspond to treewidth-3 graphs. This can be done by permitting an additional transformation called "Y-delta" or "star-triangle". Again, there are well-known formulas for computing the resistance, capacitance, or inductance across such a circuit by converting each "Y" into a "delta" (i.e., by removing degree-3 vertices) and then applying the series and parallel rules.

—In the area of artificial intelligence, knowledge-based systems must perform uncertain reasoning. Probabilistic domain knowledge can be modeled as a directed graph called a belief network. This belief network is then transformed into a related undirected graph called a junction tree. This junction tree is similar to a clique tree, and often yields a treewidth-k graph for small k. If so, then various probability distributions can be computed in linear time [Lauritzen and Spiegelhalter 1988; Jensen et al. 1990].

6.2. Translating MSOL Efficiently

MSOL expressions guarantee the existence of linear-time algorithms for the corresponding problems on any class of treewidth-k graphs; however, the hidden constants can be huge. In what follows we mention some approaches to reducing the size of such constants.

—The Mona system is a tool that translates an MSOL-like expression into a finite-state machine whose states completely represent the set of all feasible solutions. This system is powerful and efficient enough to implement MSOL on trees, and by properly encoding tree decompositions, Mona can be used to implement MSOL on treewidth-k graphs for small k [Klarlund 1998; Klarlund et al. 2002].

—It has been shown [Kassios 2001] how to translate expressions into mutumorphisms [Sasano et al. 2000]. Mutumorphisms are based on functional programming, and have the advantage that quantifiers within an expression do not cause exponential increases in the size of the hidden constants.

6.3. Determining the Treewidth

Given a graph G and positive integer k, it is $\mathcal{NP}$-complete to recognize whether G is a treewidth-k graph [Arnborg et al. 1987]. Next we mention some approaches for determining the treewidth of a graph; most also construct a corresponding tree decomposition.

—For any fixed k, there exists a linear-time algorithm that determines whether a given input graph has treewidth at most k, and if so then produces a treewidth-k decomposition [Bodlaender 1996]. While theoretically nice, this algorithm runs too slowly in practice.

—When $k \leq 4$, faster linear-time recognition procedures are known. For $k = 2$ see Valdes et al. [1982]; for $k = 3$ see Matousek and Thomas [1991]; and for $k = 4$ see Sander [1996, 1993].

—The treewidth of a graph can be computed using dynamic programming algorithms [Arnborg et al. 1987; Bodlaender et al. 2006]. Such algorithms can take exponential time, but may still run faster in practice than linear-time algorithms with huge constants.

—The treewidth can also be computed using branch-and-bound algorithms [Bachoore and Bodlaender 2006; Gogate and Dechter 2004]. These algorithms might take exponential time in the worst case, but often converge rapidly to the precise treewidth.

—Treewidth can be estimated using approximation algorithms with guaranteed constant approximation ratio. The running times of such algorithms are typically polynomial in the size of the graph, but exponential in the treewidth. For example, the approach in Robertson and Seymour [1995] yields an approximation ratio of 4, the one in Becker and Geiger [2001] yields a ratio of 11/3, and Amir [2001] provides a faster algorithm with ratio 9/2.

—Treewidth can be estimated using polynomial-time approximation algorithms with nonconstant approximation ratio. For example, an algorithm in Bouchitté et al. [2004] yields an $O(\lg k)$ ratio, and an algorithm in Feige et al. [2005] yields an $O(\sqrt{\lg k})$ ratio, where k denotes the treewidth of the graph.

—Treewidth can also be estimated using heuristic approaches without any guaranteed approximation ratios. Some algorithms promise only upper bounds on the treewidth [Clautiaux et al. 2004; Koster 1999], and other algorithms promise only lower bounds [Bodlaender et al. 2006; Lucena 2003]. Some papers deliver both upper and lower bound results [Clautiaux et al. 2003].

—Finally, additional techniques can frequently be used to improve the results obtained by some of the preceding methods. Some techniques involve preprocessing of the input graph [Bodlaender and Koster 2006; Bodlaender et al. 2005], while other techniques involve postprocessing of a tree decomposition [Blair et al. 2001].

7. *K*-TREES AND PARTIAL *K*-TREES

First, K_k is a k-tree. Then a k-tree with $n > k$ vertices is constructed from a k-tree on $n - 1$ vertices by adding a vertex adjacent to all vertices of one of its K_k subgraphs, and only to those vertices. Note that 1-trees are just ordinary unrooted trees. Now, a *partial k-tree* is a subgraph of a k-tree. In a given construction of a k-tree, the original K_k subgraph is referred to as its *basis*. A 3-tree is depicted in Figure 37; the basis graph is the complete graph on $\{a, b, c\}$. Then, below the graph the actual construction is described by specifying the added vertex and those (forming a K_3) to which it is adjacent in the prior graph. Any subgraph of the structure shown is a partial 3-tree.

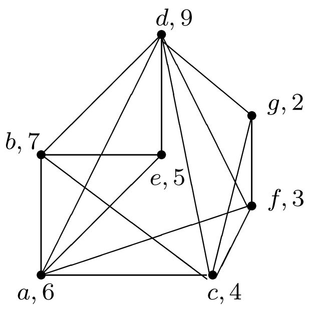
7.1. Remarks

Partial k-trees are equivalent to the class of treewidth-k graphs, so all algorithms for problems on treewidth-k graphs may also be used for the class of partial k-trees.

Also, all k-trees (but not partial k-trees) are chordal graphs, and hence also perfect graphs.

8. *K*-JACKKNIFE GRAPHS

The *k-jackknife class* of k-terminal graphs was originally described in Wimer [1987]. The base graphs in this class are graphs where all vertices are terminals; that is $\{(V, T, E) : |V| \leq k \text{ and } T = V\}$. Then, for the *generalized k-jackknife composition operation* select two k-terminal graphs, (V_1, T_1, E_1) and (V_2, T_2, E_2) where in each, $T_i =< t_{i1}, \ldots, t_{i|T_i|} >$. Now, given ordered sets $X_1 \subseteq \{1, \ldots, k\}$ and $X_2 \subseteq \{1, \ldots, k\}$ such



Basis includes a, b, and c
d adjacent to a, b, and c
e adjacent to a, b, and d
f adjacent to a, c, and d
g adjacent to c, d, and f

Fig. 37. A 3-tree.

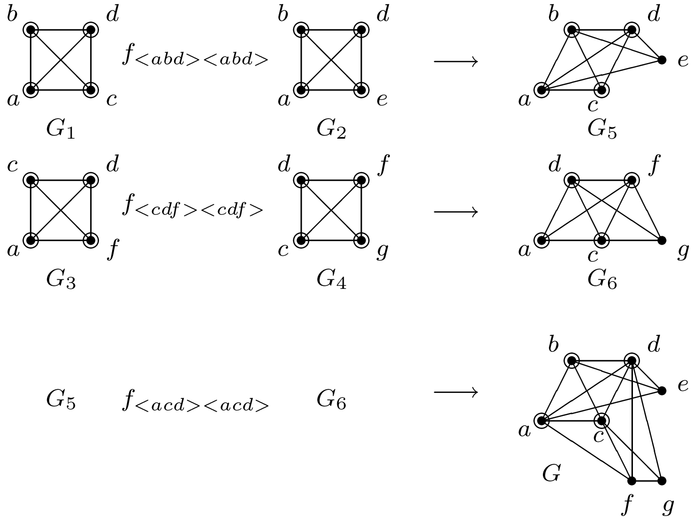


Fig. 38. Construction of a 4-jackknife graph.

that $|X_1| = |X_2| = a$ and each $X_i =< x_{i1}, \ldots, x_{ia} >$, the generalized k-jackknife operation merges pairs $t_{1x_{1j}}$ and $t_{2x_{2j}}$ so that these terminals are the only points at which the graphs (V_1, T_1, E_1) and (V_2, T_2, E_2) meet. We signify the composition using the notation $f_{<t_{1x_{11}},\ldots,t_{1x_{1a}}>,<t_{2x_{21}},\ldots,t_{2x_{2a}}>}$, and the resulting graph is given by $(V_1 \cup V_2, T_1, E_1 \cup E_2)$. The construction of a 4-jackknife graph is demonstrated in Figure 38.

8.1. Remarks

Note that the 3-tree in Figure 37 is isomorphic to the 4-jackknife graph G formed in Figure 38. In fact, this outcome is not coincidental, since there exists a general relationship between k-jackknife graphs and partial k-trees. Specifically, a graph (V, E) is a partial k-tree iff there is a subset $T \subseteq V$ such that (V, T, E) is a $(k + 1)$-jackknife

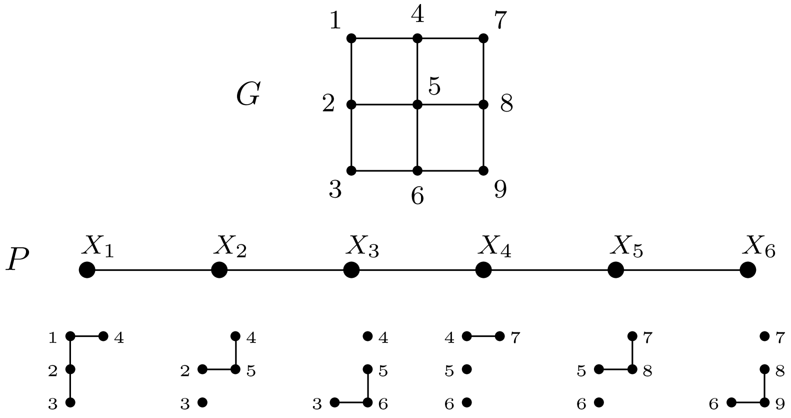


Fig. 39. Path-decomposition of a pathwidth-3 graph.

graph. So every problem that can be solved on partial k-trees, or equivalently treewidth-k graphs, can also be solved on k-jackknife graphs.

9. PATHWIDTH-*K* GRAPHS

Like the class of treewidth-k graphs, the class of pathwidth-k graphs owes its origin to the seminal work in Robertson and Seymour [1986a, 1983]. Indeed, pathwidth-k graphs are but a special case of treewidth-k graphs.

A *path decomposition* of a graph $G = (V, E)$ is defined by a path or sequence denoted by $P = (X_1, X_2, \ldots, X_t)$ such that:

—each $X_i \subseteq V$;
—$\bigcup_{1 \leq i \leq t} X_i = V$;
—for each edge $(x, y) \in E$ there exists some i, $1 \leq i \leq t$, such that $x, y \in X_i$; and
—whenever $1 \leq i \leq j \leq k \leq t$, we have that $X_i \bigcap X_k \subseteq X_j$.

Figure 39 provides an illustration. Now, the *width* of a path decomposition is $\max_{1 \leq i \leq t}\{|X_i| - 1\}$ and the *pathwidth* of a graph G is the smallest width taken over all path decompositions of the stated G. A graph G is said to be a *pathwidth-k graph* if it has pathwidth at most k. The graph in Figure 39 has pathwidth 3.

9.1. Remarks

Obviously, every pathwidth-k graph is also a treewidth-k graph, and hence every problem that can be solved on treewidth-k graphs can also be solved on pathwidth-k graphs.

Pathwidth has a nice interpretation as a search by cops on the vertices of G for an invisible robber who can move at arbitrarily high speed from vertex to adjacent vertex. Each cop is initially placed at a vertex, and may in unit time be transported to any other vertex. The robber may not occupy or traverse a vertex that is occupied by a cop. Thus the robber is caught when a cop is placed at its present vertex, if each adjacent vertex is already occupied by a cop. The pathwidth of G is one less than the minimum number of police needed to ensure capture. If the robber is visible, the minimum number of police required is one more than the treewidth [Seymour and Thomas 1993].

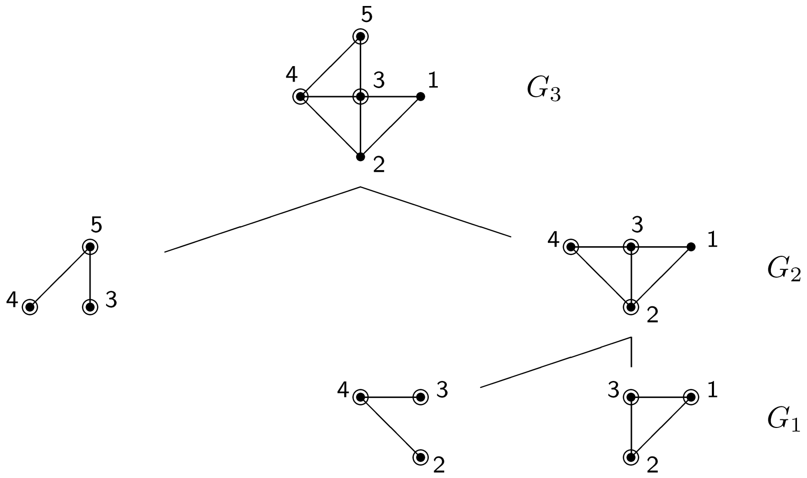


Fig. 40. A bandwidth-2 graph.

Pathwidth-k graphs are precisely the subgraphs of interval graphs with maximum clique size at most $k + 1$.

10. BANDWIDTH-*K* GRAPHS

A graph $G = (V, E)$ is a *bandwidth-k graph* if there exists a labeling of its vertices $h : V \rightarrow \{1, 2, \ldots |V|\}$ such that $(u, v) \in E$ implies $|h(u) - h(v)| \leq k$. A bandwidth-2 graph is shown in Figure 40. In general, bandwidth-k graphs are contained in a $(k+1)$-terminal recursive class that is obtained by letting B be the set of $(k + 1)$-terminal graphs with no nonterminal vertices, and $R = \{b_k\}$ where b_k is defined as follows: For $1 \leq i \leq 2$ let $G_i = (V_i, T_i, E_i)$ where $T_i =< t_{i,1}, \ldots, t_{i,k+1} >$. Then $b_k(G_1, G_2)$ merges $t_{1,j+1}$ with $t_{2,j}$ for $1 \leq j \leq k$ and creates a new graph $(V_1 \cup V_2, T_1, E_1 \cup E_2)$.

10.1. Remarks

Every bandwidth-k graph is a pathwidth-k graph, and hence also a treewidth-k graph. So every problem that can be solved on pathwidth-k graphs or treewidth-k graphs can also be solved on bandwidth-k graphs.

Bandwidth-k graphs are precisely the subgraphs of proper interval graphs with maximum clique size at most $k + 1$.

11. CUTWIDTH-*K* GRAPHS

A graph $G = (V, E)$ is a *cutwidth-k graph* if there exists a labeling of its vertices $h : V \rightarrow \{1, 2, \ldots |V|\}$ such that for all j, $|\{(u, v) \in E : f(u) \leq j < f(v)\}| \leq k$. A cutwidth-2 graph is shown in Figure 41. As with bandwidth-k graphs, cutwidth-k graphs are also contained in a $(k + 1)$-terminal recursive class.

11.1. Remarks

The classes of cutwidth-k graphs and bandwidth-k graphs are incomparable. The graph in Figure 40 is not cutwidth-2, and the graph in Figure 41 is not bandwidth-2. However, similar to bandwidth-k graphs, every cutwidth-k graph is also both pathwidth-k and

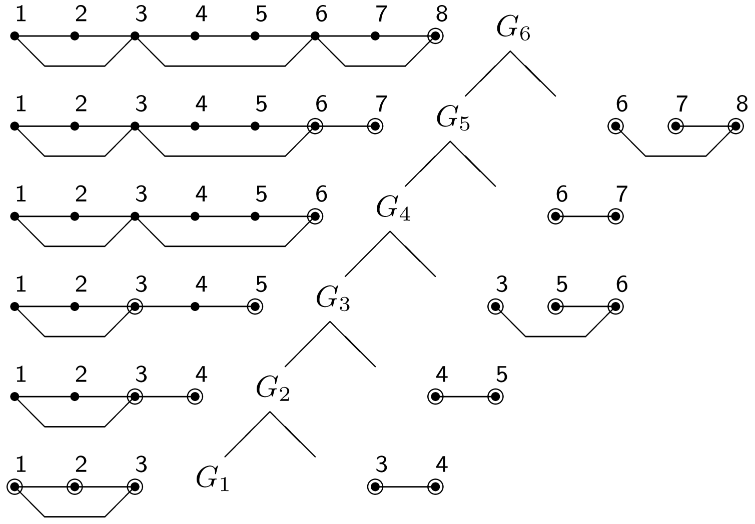


Fig. 41. A cutwidth-2 graph.

treewidth-k. So every problem which is solvable on pathwidth-k graphs or treewidth-k graphs is also solvable on cutwidth-k graphs.

12. BRANCHWIDTH-*K* GRAPHS

The concept of branchwidth was introduced in Robertson and Seymour [1995, 1991]. A *branch decomposition* of a graph $G = (V, E)$ is a pair (T, f), where T is an unrooted ternary tree (vertices have degree either 1 or 3) and f is a bijection from the leaves of T to E. If instead the degree of every nonleaf vertex in T is *at least* 3, the pair (T, f) is called a *partial branch-decomposition*.

Now, the *order* of an edge e of T in a branch decomposition is the number of vertices v in V such that there exist leaves l_1 and l_2 of T residing in different components of $T - e$, where $f(l_1)$ and $f(l_2)$ are both incident to v. In other words, the order of e is the number of vertices v in G that have incident edges corresponding to leaves in both components of $T - e$. The *width* of (T, f) is the maximum order of the edges of T and the *branchwidth* of G is the minimum width taken over all branch decompositions of G. A graph G is a *branchwidth-k graph* if it has branchwidth at most k. Figure 42 shows a branchwidth-2 graph as well as its branch decomposition; as an example, the order of edge (g, j) is 2 due to vertices 1 and 5.

12.1. Remarks

The following results are shown in Robertson and Seymour [1991]. The class of branchwidth-2 graphs is identical to the class of treewidth-2 graphs, but this same relationship is not known to hold for any $k \neq 2$. Every treewidth-k graph is a branchwidth-$(k + 1)$ graph. For $k < 2$ every branchwidth-k graph is a treewidth-1 graph, and for $k \geq 2$ every branchwidth-k graph is a treewidth-$(\lfloor \frac{3k}{2} \rfloor - 1)$ graph.

13. HALIN GRAPHS

A *Halin graph* is a planar graph having the property that its edge set E can be partitioned as $E = \langle T, C \rangle$, where T is a tree with no vertex of degree 2 and C is a cycle

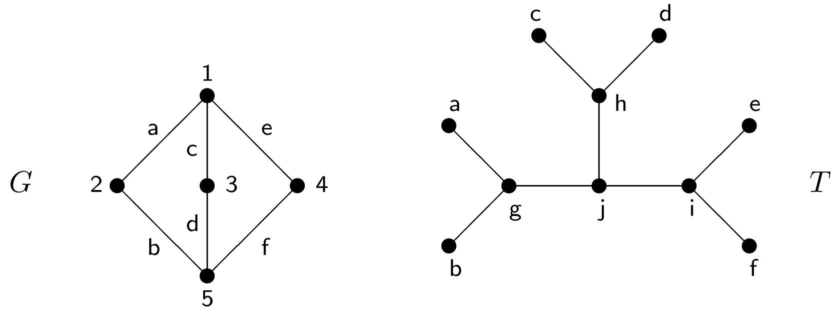


Fig. 42. Branch decomposition of a branchwidth-2 graph.

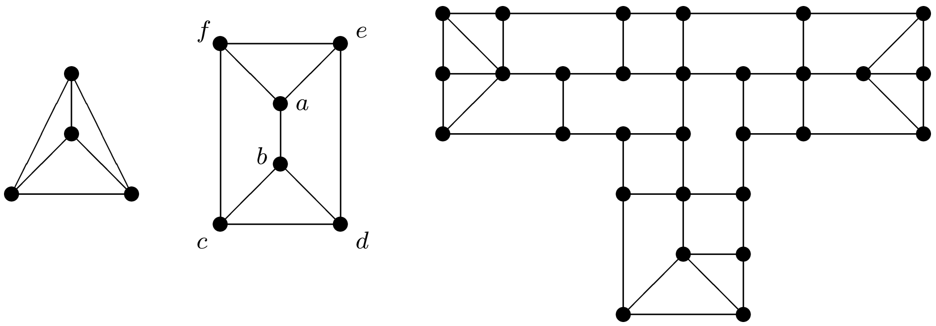


Fig. 43. Some Halin graphs.

including only and all leaves (i.e., degree-1 vertices) of T. A selection of Halin graphs is shown in Figure 43. The drawings in all cases are such that the cycle component of the respective partitions lies on the outer face. Removal of these edges should leave a tree with the correct vertex degrees.

Halin graphs are edge minimal, planar, 3-connected graphs and are easily recognizable by ad hoc means: First, check if planar graph G is 3-connected, then embed it in the plane and seek a face in the embedded graph such that the edges defining the face, if removed, leave a tree without degree-2 vertices. More importantly, we can easily show constructively that these graphs can always be decomposed as 3-terminal graphs. Consider the middle Halin graph in Figure 43 and fix an embedding of the graph as shown in Figure 44. The terminals are circled and labeled by t_1, t_2, and t_3 as indicated. It is important that t_1 is a nonleaf vertex of the identifying tree subgraph, that t_2 is the leftmost leaf, and that t_3 is the rightmost leaf. After the edge (t_2, t_3) is removed, the right descendant subgraph results as shown. Now, to correctly decompose this graph further, simply identify the leaf vertex that is closest to t_3 and that can be reached by a path from t_2 that passes neither through other leaf vertices nor through t_1. In this simple example, this corresponds to vertex d and we are led to the next pair of descendent subgraphs as indicated. Then it should be clear how decomposition proceeds from this point, culminating in base graphs with all vertices terminal. Though described using an example, the process demonstrated is sufficient to yield a template that is valid for the 3-terminal decomposition of *any* Halin graph.

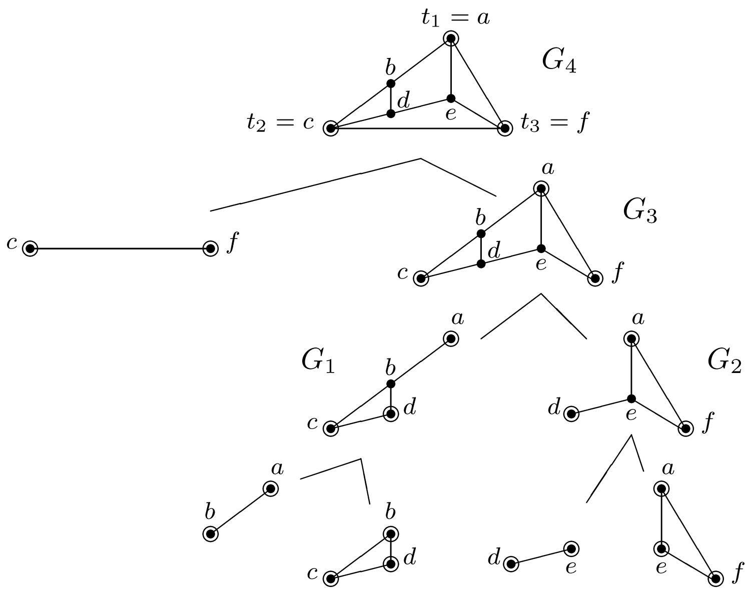


Fig. 44. Halin graph decomposition.

13.1. Remarks

A number of interesting properties are distinctly associated with Halin graphs. From the thematic perspective of this tutorial, their membership in an efficiently recognizable 3-terminal graph class is important to note. In addition, all Halin graphs are treewidth-3 graphs, so every problem that can be solved on treewidth-*k* graphs can also be solved on Halin graphs.

It is also interesting to observe some purely graph-theoretic properties of Halin graphs. For example, all Halin graphs are Hamiltonian but no Halin graphs are bipartite. Halin graphs are *1-Hamiltonian* in that if any vertex is removed the resulting graph remains Hamiltonian. All Halin graphs of even order are *bicritical* in that if any two vertices are deleted, the resulting graph possesses a 1-factor. Halin graphs are also so-called *class-1* graphs in that their chromatic index is always the same as the maximum vertex degree. And finally, if a Halin graph possesses more than one correct bipartiton, say $\langle T_i, C_i \rangle$ and $\langle T_j, C_j \rangle$, then T_i and T_j are isomorphic. More detailed descriptions of these and other properties of Halin graphs can be found in Horton et al. [1992].

14. COGRAPHS

Properties of *cographs*, also known as *complement reducible graphs*, were largely developed in Corneil et al. [1985, 1984, 1981]. A graph with a single vertex is a cograph. Now, suppose G_1 and G_2 are cographs. Then the disjoint union $G_1 \cup G_2$ is a cograph. Also, the cross-product $G_1 \times G_2$ is a cograph, which is formed by taking the union of G_1 and G_2 and adding all edges (v_1, v_2) where v_1 is in G_1 and v_2 is in G_2. Figure 45 demonstrates the construction.

Importantly, every cograph can be composed from single vertices using only the operations $G = G_1 \cup G_2$ and $G = G_1 \times G_2$. This is sufficient to produce linear-time dynamic programming algorithms for many problems restricted to cographs. A sample of algorithms follow.

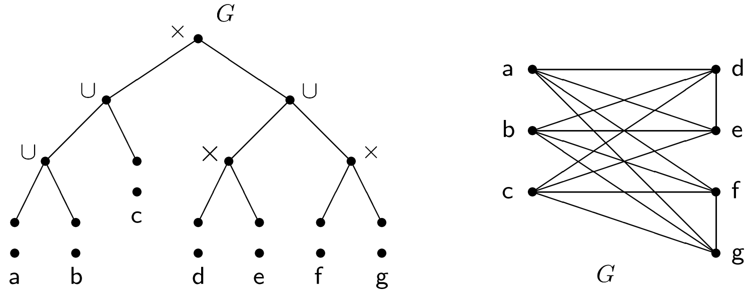


Fig. 45. Recursive construction of a cograph.

```
if $|V| = 1$ then
    $G.v \leftarrow 0$
else if $G = G_1 \cup G_2$ then
    $G.v \leftarrow G_1.v + G_2.v$
else if $G = G_1 \times G_2$ then
    $G.v \leftarrow$ min $\{G_1.v + |V_2|, G_2.v + |V_1|\}$
```

Fig. 46. Algorithm for minimum cardinality vertex cover in a cograph.

```
if $|V| = 1$ then
    $G.c \leftarrow 1$
else if $G = G_1 \cup G_2$ then
    $G.c \leftarrow$ max $\{G_1.c, G_2.c\}$
else if $G = G_1 \times G_2$ then
    $G.c \leftarrow G_1.c + G_2.c$
```

Fig. 47. Algorithm for maximum cardinality clique and chromatic number in a cograph.

```
if $|V| = 1$ then
    $G.d \leftarrow 1$
else if $G = G_1 \cup G_2$ then
    $G.d \leftarrow G_1.d + G_2.d$
else if $G = G_1 \times G_2$ then
    $G.d \leftarrow$ min $\{G_1.d, G_2.d, 2\}$
```

Fig. 48. Algorithm for minimum-cardinality dominating set in a cograph.

14.1. Algorithm for Minimum Cardinality Vertex Cover in a Cograph

The algorithm is shown in Figure 46.

14.2. Algorithm for Maximum Cardinality Clique and Chromatic Number in a Cograph

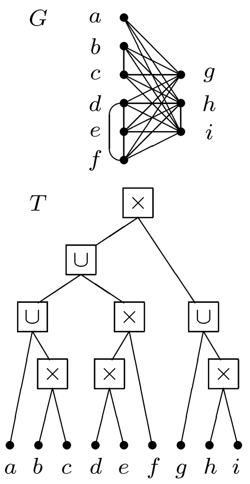
The maximum clique size is always identical to the chromatic number for any cograph, so both these parameters can be computed via the same procedure. The algorithm is shown in Figure 47.

14.3. Algorithm for Minimum Cardinality Dominating Set in a Cograph

The algorithm is shown in Figure 48.

```
if $|V| = 1$ then
    $G.m \leftarrow 0$
else if $G = G_1 \cup G_2$ then
    $G.m \leftarrow G_1.m + G_2.m$
else if $G = G_1 \times G_2$ then
    $G.m \leftarrow \min\{G_1.m + |V_2|, G_2.m + |V_1|, \lfloor(|V_1| + |V_2|)/2\rfloor\}$
```

Fig. 49. Algorithm for maximum-cardinality matching in a cograph.



Subgraph	$[v, c, d, m]$
$G_1 = \{a\}$	$[0, 1, 1, 0]$
$G_2 = \{b\}$	$[0, 1, 1, 0]$
$G_3 = \{c\}$	$[0, 1, 1, 0]$
$G_4 = \{d\}$	$[0, 1, 1, 0]$
$G_5 = \{e\}$	$[0, 1, 1, 0]$
$G_6 = \{f\}$	$[0, 1, 1, 0]$
$G_7 = \{g\}$	$[0, 1, 1, 0]$
$G_8 = \{h\}$	$[0, 1, 1, 0]$
$G_9 = \{i\}$	$[0, 1, 1, 0]$
$G_{10} = G_2 \times G_3$	$[1, 2, 1, 1]$
$G_{11} = G_4 \times G_5$	$[1, 2, 1, 1]$
$G_{12} = G_8 \times G_9$	$[1, 2, 1, 1]$
$G_{13} = G_1 \cup G_{10}$	$[1, 2, 2, 1]$
$G_{14} = G_{11} \times G_6$	$[2, 3, 1, 1]$
$G_{15} = G_7 \cup G_{12}$	$[1, 2, 2, 1]$
$G_{16} = G_{13} \cup G_{14}$	$[3, 3, 3, 2]$
$G_{17} = G_{16} \times G_{15}$	$[6, 5, 2, 4]$

Fig. 50. Optimum vertex cover, clique, dominating set, and matching in a cograph.

14.4. Algorithm for Maximum Cardinality Matching in a Cograph

The algorithm is shown in Figure 49. The last formula in this algorithm is obtained by simplifying the following more straightforward but less efficient expression

$$\max\left\{k + \min\left\{G_1.m, \left\lfloor\frac{|V_1| - k}{2}\right\rfloor\right\} + \min\left\{G_2.m, \left\lfloor\frac{|V_2| - k}{2}\right\rfloor\right\} : \right.$$
$$\left. 0 \le k \le \min\{|V_1|, |V_2|\}\right\},$$

where k denotes the number of matching edges with one endpoint in each of the subgraphs G_1 and G_2.

14.4.1. Example. The algorithms of Figures 46 through 49 are demonstrated on the cograph G shown in Figure 50. T denotes the tree decomposition of G, and each 4-tuple exhibits values for $G.v$, $G.c$, $G.d$, and $G.m$. The minimum vertex cover has size 6, for example $\{b, d, e, g, h, i\}$. The maximum clique has size 5, given by $\{d, e, f, h, i\}$. The minimum dominating set has size 2, for example $\{a, g\}$. Lastly, the maximum matching has size 4, for example $\{(a, g), (b, h), (c, i), (d, e)\}$.

```
if $|V| = 1$ then
    $[G.dp,\ G.hp,\ G.hc] \leftarrow [1,\ true,\ false]$
else if $G = G_1 \cup G_2$ then
    $[G.dp,\ G.hp,\ G.hc] \leftarrow [G_1.dp + G_2.dp,\ false,\ false]$
else if $G = G_1 \times G_2$ then
    $G.dp \leftarrow \max\ \{1,\ G_1.dp - |V_2|,\ G_2.dp - |V_1|\}$
    $G.hp \leftarrow (G.dp = 1)$
    $G.hc \leftarrow (|V| \geq 3)$ and $(G_1.dp \leq |V_2|)$ and $(G_2.dp \leq |V_1|)$
```

Fig. 51. Algorithm for Hamiltonian path and Hamiltonian cycle in a cograph.

14.5. Algorithm for Hamiltonian Path and Hamiltonian Cycle in a Cograph

Define

G.dp = minimum number of disjoint paths that cover all the vertices
G.hp = Hamiltonian path exists
G.hc = Hamiltonian cycle exists

The algorithm is shown in Figure 51.

14.6. Remarks

Given any cograph, its decomposition tree can be found in $O(|V| + |E|)$ time [Corneil et al. 1985], and it will have $O(|V|)$ size. Each algorithm in this section has $O(1)$ equivalence classes, and thus runs in $O(|V|)$ time.

From the graph-theoretic perspective, there are numerous interesting properties of cographs. For example, the complement of any cograph is also a cograph, all cographs are perfect, and cographs are characterized by the absence of an induced P_4 subgraph. Algorithmically, weighted versions of the vertex cover, clique, and dominating set problems are solvable in linear time on cographs by making straightforward modifications to the algorithms of Figures 46, 47, and 48.

However, it is not known whether the weighted versions of problems such as matching or Hamiltonian path or cycle can be solved in linear time on cographs. Intuitively, the algorithms of Figures 49 and 51 do not obviously generalize to solve the weighted problems because the cross-product operation adds many edges where each edge potentially has a different weight. (But it is known that weighted versions of the matching and Hamiltonian path/cycle problems are solvable in polynomial time on cographs.)

15. CLIQUEWIDTH-*K* GRAPHS

The graph parameter *cliquewidth* was introduced in Courcelle et al. [1993] and formed a seminal concept in linking research in graph theory and logic. Let $[k]$ denote the set of integers $\{1, \ldots, k\}$. Then any graph with only one vertex v having label $l(v) \in [k]$ is a *cliquewidth-k graph*. Now let G_1 and G_2 be cliquewidth-*k* graphs and let $i, j \in [k]$, $i \neq j$. Then the disjoint union $G_1 \oplus G_2$ is a cliquewidth-*k* graph. Also, the graph $\eta_{i,j}(G_1)$ is a cliquewidth-*k* graph, which is formed from G_1 by adding all edges (v_1, v_2) where $l(v_1) = i$ and $l(v_2) = j$. Finally, the graph $\rho_{i \to j}(G_1)$ is a cliquewidth-*k* graph, which is formed from G_1 by switching all vertices with label i to label j.

Recursively, every cliquewidth-*k* graph can be composed from single vertices with labels in $[k]$ using only the operations $G = G_1 \oplus G_2$, $G = \eta_{i,j}(G_1)$, and $G = \rho_{i \to j}(G_1)$. The *cliquewidth* of a graph G is the smallest value of k such that G is a cliquewidth-*k* graph.

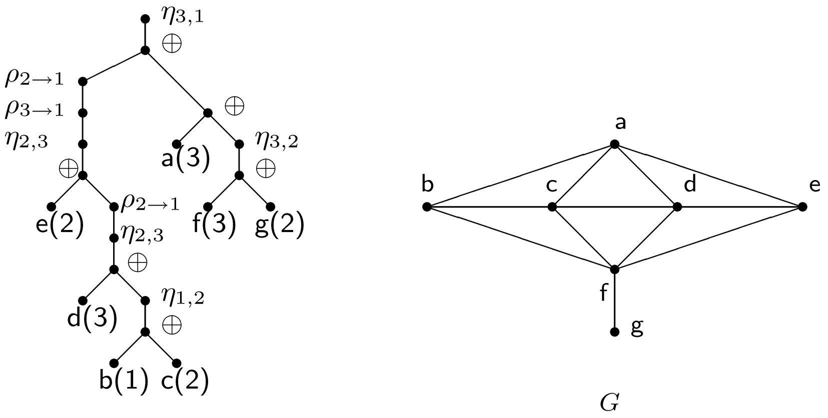


Fig. 52. Recursive construction of a cliquewidth-3 graph.

```
if $|V| = 1$ then
      let $v \in V$
      $\forall_{S\subseteq[k]}\ G[S] \leftarrow$ if label($v$) $\in S$ then 1 else 0
else if $G = G_1 \oplus G_2$ then
      $\forall_{S\subseteq[k]}\ G[S] \leftarrow G_1[S] + G_2[S]$
else if $G = \eta_{i,j}(G_1)$ then
      $\forall_{S\subseteq[k]}\ G[S] \leftarrow$ min $\{G_1[S\cup\{i\}],\ G_1[S\cup\{j\}]\}$
else if $G = \rho_{i\to j}(G_1)$ then
      $\forall_{S\subseteq[k]}\ G[S] \leftarrow$ if $j \in S$ then $G_1[S\cup\{i\}]$ else $G_1[S-\{i\}]$
$G.min \leftarrow$ min $\{G[S] : S \subseteq [k]\}$
```

Fig. 53. Algorithm for minimum-cardinality vertex cover in a cliquewidth-k graph.

A *cliquewidth decomposition* for a graph is a rooted tree such that the root corresponds to G, each leaf corresponds to a labeled, single-vertex graph, and each nonleaf node of the tree is obtained by applying one of the operations $\oplus$, $\eta_{i,j}$, or $\rho_{i\to j}$ to its child or children. A cliquewidth-3 decomposition is demonstrated in Figure 52.

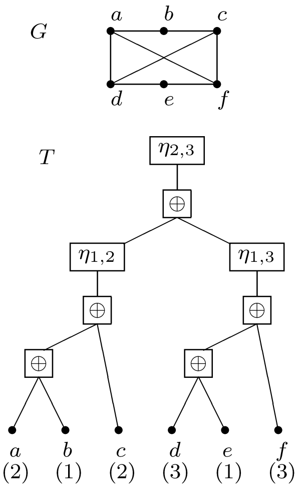
15.1. Algorithm for Minimum Cardinality Vertex Cover in a Cliquewidth-*k* Graph

Define

$G[S]$ = min cardinality vertex cover that contains every vertex with label in $S \subseteq [k]$
$G.min$ = min cardinality vertex cover

The algorithm is shown in Figure 53.

15.1.1. Example. We demonstrate the algorithm of Figure 53 on the cliquewidth-3 graph G shown in Figure 54. T denotes the tree decomposition of G. Each 8-tuple consists of $G[\emptyset]$, $G[\{1\}]$, $G[\{2\}]$, $G[\{3\}]$, $G[\{1,2\}]$, $G[\{1,3\}]$, $G[\{2,3\}]$, and $G[\{1,2,3\}]$. The minimum vertex cover has size 3, given by either {*a,c,e*} or {*b,d,f*}.



Subgraph	
$G_1 = \{a\}$	$[0, 0, 1, 0, 1, 0, 1, 1]$
$G_2 = \{b\}$	$[0, 1, 0, 0, 1, 1, 0, 1]$
$G_3 = \{c\}$	$[0, 0, 1, 0, 1, 0, 1, 1]$
$G_4 = \{d\}$	$[0, 0, 0, 1, 0, 1, 1, 1]$
$G_5 = \{e\}$	$[0, 1, 0, 0, 1, 1, 0, 1]$
$G_6 = \{f\}$	$[0, 0, 0, 1, 0, 1, 1, 1]$
$G_7 = G_1 \oplus G_2$	$[0, 1, 1, 0, 2, 1, 1, 2]$
$G_8 = G_4 \oplus G_5$	$[0, 1, 0, 1, 1, 2, 1, 2]$
$G_9 = G_3 \oplus G_7$	$[0, 1, 2, 0, 3, 1, 2, 3]$
$G_{10} = G_6 \oplus G_8$	$[0, 1, 0, 2, 1, 3, 2, 3]$
$G_{11} = \eta_{1,2}(G_9)$	$[1, 1, 2, 1, 3, 1, 2, 3]$
$G_{12} = \eta_{1,3}(G_{10})$	$[1, 1, 1, 2, 1, 3, 2, 3]$
$G_{13} = G_{11} \oplus G_{12}$	$[2, 2, 3, 3, 4, 4, 4, 6]$
$G_{14} = \eta_{2,3}(G_{13})$	$[3, 4, 3, 3, 4, 4, 4, 6]$

Fig. 54. Minimum cardinality vertex cover in a cliquewidth-3 graph.

```
if $|V| = 1$ then
    let $v \in V$
    $\forall_{S,T \subseteq [k]}$ $G[S,T] \leftarrow$ if $S$=$\{$label$(v)\}$ and $T = [k]$ then 1
        else if $S$=$\emptyset$ and $T$=$[k]-\{$label$(v)\}$ then 0
        else $\infty$
else if $G = G_1 \oplus G_2$ then
    $\forall_{S,T \subseteq [k]}$ $G[S,T] \leftarrow$ min $\{G_1[S_1,T_1] + G_2[S_2,T_2] : S = S_1 \cup S_2, T = T_1 \cap T_2\}$
else if $G = \eta_{i,j}(G_1)$ then
    $\forall_{S,T \subseteq [k]}$ $G[S,T] \leftarrow$ min $\{G_1[S,T_1] : T = T_1 \cup \{i : j \in S\} \cup \{ j : i \in S\}\}$
else if $G = \rho_{i \to j}(G_1)$ then
    $\forall_{S,T \subseteq [k]}$ $G[S,T] \leftarrow$ min $\{G_1[S_1,T_1] : S = S_1 - \{i\} \cup \{j : i \in S_1\},$
            $T = T_1 \cup \{i\} - \{j : i \notin T_1\}\}$
$G.min \leftarrow$ min $\{G[S,[k]] : S \subseteq [k]\}$
```

Fig. 55. Algorithm for minimum cardinality dominating set in a cliquewidth-*k* graph.

15.2. Algorithm for Minimum Cardinality Dominating Set in a Cliquewidth-*k* Graph

Define

$G[S,T]$ = min cardinality dominating or almost-dominating set that contains exactly the labels in $S \subseteq [k]$, and that dominates all vertices with labels in $T \subseteq [k]$

$G.min$ = min cardinality dominating set

The algorithm is shown in Figure 55.

15.3. More about Cliquewidth

It is sometimes interesting to understand the nomenclature associated with graph classes, and this is the case for cliquewidth-*k* graphs. As every tree is a treewidth-1 graph, we can view treewidth as a measure of how much a graph varies from a tree. Similarly, every clique is a cliquewidth-2 graph, so cliquewidth is a measure of how much a graph varies from a clique, hence the basis for coining the term *cliquewidth* [Courcelle and Olariu 2000].

The algorithms in Figures 53 and 55 each run in $O(|V|)$ time, assuming a decomposition tree is already known. Many other problems including variations of independent set, clique, and m-vertex colorability (for fixed m) can also be solved in linear time on cliquewidth-k graphs, provided that a decomposition tree is known. These problems are all expressible in a particular language that we will call MSOL′.

MSOL′ for a graph $G = (V, E)$ denotes a subset of MSOL restricted to variables v_i with domain V, and V_i with domain 2^V. The language contains primitive predicates such as $\mathrm{v}_i = \mathrm{v}_j$, Adjacent($\mathrm{v}_i$, v_j), and $\mathrm{v}_i \in \mathrm{V}_j$. MSOL′ also permits the logical operators ($\neg$, $\wedge$, $\vee$) and quantifiers ($\exists$, $\forall$). In other words, MSOL′ is essentially the same as MSOL without the edge variables e_i and edge set variables E_j.

Every MSOL′-expressible problem can be solved in linear time on any class of cliquewidth-k graphs [Courcelle et al. 2000], provided that either there exists a linear time decomposition algorithm for the class (as for cographs) or a decomposition tree is provided as part of the input. This statement holds for variations of each MSOL′ problem that involve existence, optimum cardinality, optimum total weight, counting the number of solutions, etc. Once a problem is expressed in MSOL′, a linear-time dynamic programming algorithm can also be created mechanically.

Note that our MSOL expressions for problems VertexCover, IndependentSet, Clique, DominatingSet, VertexColoring$_m$, and CliquePartition$_m$ given previously in Figure 36 are also MSOL′ expressions. On the other hand, our MSOL expressions for Matching, Connected, HamCycle, and HamPath in Figure 36 are not in MSOL′.

Some problems such as variations of matching and Hamiltonicity do not appear to be expressible in MSOL′, and it is not known whether these problems can be solved in linear time on cliquewidth-k graphs. However, such problems can often be solved in polynomial time, given the decomposition tree. Polynomial time is achieved by constructing a polynomial-size data structure corresponding to each node in the tree decomposition, as illustrated by the following algorithm adapted from Wanke [1994] and Espelage et al. [2001].

15.4. Algorithm for Hamiltonian Path in a Cliquewidth-*k* Graph

Define

p_{ij} $= p_{ji} =$ count of the number of disjoint paths whose endpoints have labels i and j
P $= (p_{11}, \ldots, p_{ij}, \ldots, p_{kk})$ is a valid $\frac{k(k+1)}{2}$-tuple with compatible p_{ij} for $1 \le i \le j \le k$
$G.Q$ $= \{P\}$ is the set of all valid $\frac{k(k+1)}{2}$-tuples
$G.hp$ $=$ Hamiltonian path exists

The algorithm is shown in Figure 56; it invokes the closure function which is given in Figure 57. Note that $0 \le p_{ij} \le |V|$ for all labels i and j, and indeed $1 \le \Sigma_{i,j}\ p_{ij} \le |V|$ for every valid tuple. Hence the size of each set Q is $O(|V|^{k(k+1)/2})$. Therefore the algorithm runs in polynomial time with suitable data structures.

15.5. Remarks

Additional algorithms on cliquewidth-k graphs can be found in Kobler and Rotics [2003], Makowsky et al. [2006], and Gurski and Wanke [2006, 2004].

It is not currently known whether cliquewidth-k graphs can be recognized in polynomial time. However, Section 18 will discuss some polynomial-time algorithms for obtaining an approximate decomposition tree [Oum and Seymour 2006; Oum 2005a].

```
if $|V| = 1$ then
    let $v \in V$
    $G.Q \leftarrow \{P$ : if $i =$ label($v$) and $j =$ label($v$) then $p_{ij} = 1$ else $p_{ij} = 0\}$
else if $G = G_1 \oplus G_2$ then
    $G.Q \leftarrow \{P$ : there exists $P' \in G_1.Q$ and $P'' \in G_2.Q$ such that
        $p_{ij} = p'_{ij} + p''_{ij}$ for each $i$ and $j\}$
else if $G = \eta_{i,j}(G_1)$ then
    $G.Q \leftarrow$ closure($G_1.Q$, $i$, $j$)
else if $G = \rho_{i \to j}(G_1)$ then
    $G.Q \leftarrow \{P$ : there exists $P' \in G_1.Q$ such that
        $p_{hi} = 0$ for all $h$, $p_{hj} = p'_{hi} + p'_{hj}$ for all $h \neq i$,
        and $p_{hl} = p'_{hl}$ for all $h$ and $l$ such that $\{h, l\} \cap \{i, j\} = \emptyset\}$
$G.hp \leftarrow \bigvee \{1 = \Sigma_{1 \leq i \leq j \leq k}\ p_{ij} : P \in G.Q\}$
```

Fig. 56. Algorithm for Hamiltonian path in a cliquewidth-*k* graph.

```
closure($Q'$, $i$, $j$) {
    $Q \leftarrow Q'$;
    while (more tuples can be added to $Q$) do {
        choose any tuple $P' \in Q$ and labels $h$, $l$ such that $p'_{hi} > 0$ and $p'_{jl} > 0$;
        $P \leftarrow P'$;
        $p_{hi} \leftarrow p_{hi} - 1$;
        $p_{jl} \leftarrow p_{jl} - 1$;
        $p_{hl} \leftarrow p_{hl} + 1$;
        add tuple $P$ to $Q$;
    }
    return $Q$;
}
```

Fig. 57. Closure function for Hamiltonian path algorithm.

The class of cliquewidth-2 graphs is identical to the class of cographs, and every treewidth-*k* graph is a cliquewidth-($3 \cdot 2^{k-1}$) graph [Corneil and Rotics 2005]. Also, for all *k*, there exist cliquewidth-*k* graphs with arbitrarily large treewidth. For example, the complete graph K_n has cliquewidth 2 and treewidth $n - 1$. Finally, every cliquewidth-*k* graph that does not contain a complete bipartite subgraph $K_{t,t}$ is a treewidth-($3kt - 3k - 1$) graph [Gurski and Wanke 2000].

16. *K*-NLC GRAPHS

The underlying feature distinguishing graphs in this class is their construction from isolated vertices by employing a bipartite join and certain vertex relabeling operations. Again, let $[k]$ denote the set of integers $\{1, \ldots, k\}$. Any graph with a single vertex v such that $l(v) \in [k]$ is a *k-NLC (node label controlled) graph*. Now, let G_1 and G_2 be k-NLC graphs and let $i, j \in [k]$. Let B denote a bipartite graph on $[k] \times [k]$ having edge set E_B. The join $G_1 \times_B G_2$ is a k-NLC graph, which is formed from the disjoint union $G_1 \oplus G_2$ by adding all edges (v_1, v_2) such that v_1 is in V_1, $l(v_1) = i$, v_2 is in V_2, $l(v_2) = j$, and (i, j) is an edge in E_B. Finally, the graph $\rho_{i \to j}(G_1)$ is a k-NLC graph, which is formed from G_1 by switching all vertices with label i to label j [Wanke 1994].

Recursively, every k-NLC graph can be composed from single vertices with labels in $[k]$ using only the operations $G = G_1 \times_B G_2$ and $G = \rho_{i \to j}(G_1)$. The *NLC-width* of a graph G is the smallest value of k such that G is a k-NLC graph. A 2-NLC construction is demonstrated in Figure 58.

Algorithms on k-NLC graphs are similar to those for cliquewidth-k graphs.

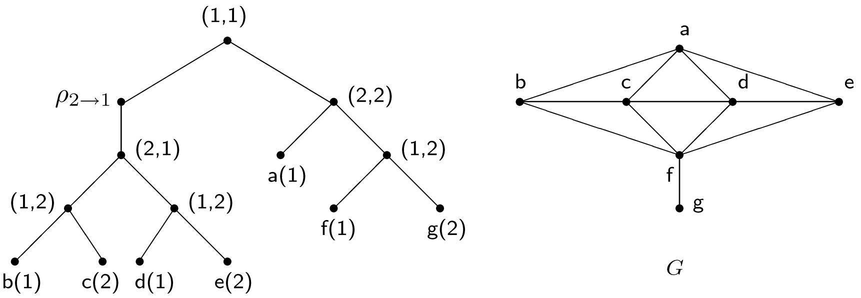
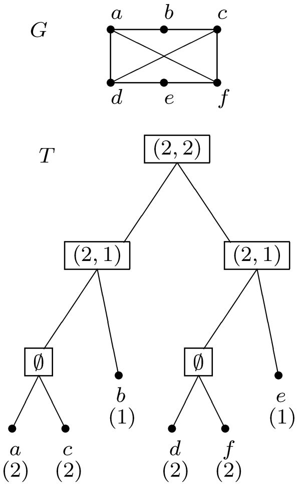


Fig. 58. Recursive construction of a 2-NLC graph.

```
if $|V| = 1$ then
    let $v \in V$
    $\forall_{S\subseteq[k]}\ G[S] \leftarrow$ if label$(v) \in S$ then 1 else 0
else if $G = G_1 \times_B G_2$ then
    $\forall_{S\subseteq[k]}\ G[S] \leftarrow$ min $\{G_1[S_1] + G_2[S_2] : S \subseteq S_1 \cap S_2,$
                                   $E_B \subseteq (S_1 \times [k]) \cup ([k] \times S_2)\}$
else if $G = \rho_{i\to j}(G_1)$ then
    $\forall_{S\subseteq[k]}\ G[S] \leftarrow$ if $j \in S$ then $G_1[S\cup\{i\}]$ else $G_1[S-\{i\}]$
$G.min \leftarrow$ min $\{G[S] : S \subseteq [k]\}$
```

Fig. 59. Algorithm for minimum cardinality vertex cover in a k-NLC graph.



Subgraph	
$G_1 = \{a\}$	$[0, 0, 1, 1]$
$G_2 = \{c\}$	$[0, 0, 1, 1]$
$G_3 = \{b\}$	$[0, 1, 0, 1]$
$G_4 = \{d\}$	$[0, 0, 1, 1]$
$G_5 = \{f\}$	$[0, 0, 1, 1]$
$G_6 = \{e\}$	$[0, 1, 0, 1]$
$G_7 = G_1 \times_\emptyset G_2$	$[0, 0, 2, 2]$
$G_8 = G_4 \times_\emptyset G_5$	$[0, 0, 2, 2]$
$G_9 = G_7 \times_{(2,1)} G_3$	$[1, 1, 2, 3]$
$G_{10} = G_8 \times_{(2,1)} G_6$	$[1, 1, 2, 3]$
$G_{11} = G_9 \times_{(2,2)} G_{10}$	$[3, 4, 4, 6]$

Fig. 60. Minimum cardinality vertex cover in a 2-NLC graph.

16.1. Algorithm for Minimum-Cardinality Vertex Cover in a *k*-NLC Graph

Define

$G[S]$ = min cardinality vertex cover that contains every vertex with label in $S \subseteq [k]$

$G.min$ = min cardinality vertex cover

The algorithm is shown in Figure 59. It is demonstrated on the 2-NLC graph G shown in Figure 60. Specifially, T denotes the tree decomposition of G. Each 4-tuple consists of $G[\emptyset]$, $G[\{1\}]$, $G[\{2\}]$, and $G[\{1, 2\}]$. The optimal solution has cardinality 3.

```
if $|V| = 1$ then
    $G[S] \leftarrow |S|$
else if $G = G_1 \times_{B,h} G_2$ then
    $G[S] \leftarrow \min \{G_1[T] + G_2[U] : X \subseteq h(V_1), Y \subseteq h(V_2),$
                  $E_B \subseteq (X \times Z_2) \cup (Z_1 \times Y),$
                  $T = S \cap h^{-1}(X), U = S \cap h^{-1}(Y)\}$
$G.min \leftarrow \min \{G[S]\}$
```

Fig. 61. Algorithm for minimum cardinality vertex cover in a k-HB graph.

16.2. Remarks

The algorithm in Figure 59 runs in $O(|V|)$ time, assuming a decomposition tree is already known. Note that this algorithm could easily be modified to find a maximum independent set.

The complement of any k-NLC graph is also a k-NLC graph. Hence the algorithm of Figure 59 could also be adapted to find a maximum cardinality clique.

The class of 1-NLC graphs is identical to the class of cographs. Every treewidth-k graph is a $(2^{k+1} - 1)$-NLC graph [Wanke 1994]. In addition, every cliquewidth-k graph is a k-NLC graph, and every k-NLC graph is a cliquewidth-$2k$ graph [Johansson 1998].

17. *K*-HB GRAPHS

k-HB (homogeneous balanced) graphs are graphs such that a particular $O(|V|^{k+2})$-time top-down decomposition algorithm constructs a *balanced* $(k, 2^k)$*-pseudo-NLC decomposition*. Every k-HB graph can be composed from single vertices using only the operation $G = G_1 \times_{B,h} G_2$. Here G_1 and G_2 denote child subgraphs, each $|V_i| \leq \frac{2|V|}{3}$, $B = (V_B, E_B)$ is a bipartite graph with $V_B = Z_1 \cup Z_2$ and $E_B \subseteq Z_1 \times Z_2$, $|Z_1| \leq k$, $|Z_2| \leq 2^k$, h: $V \rightarrow V_B$ is a mapping with each $h(V_i) \subseteq Z_i$, and $(x, y) \in E$ iff $(h(x), h(y)) \in E_B$ for each $x \in V_1$ and $y \in V_2$.

This k-HB decomposition leads to polynomial-time algorithms for many problems on k-HB graphs, using ordinary recursion (top-down computation, without memoization) rather than dynamic programming techniques. Each algorithm's running time is polynomial because at each node of the decomposition it evaluates $O(1)$ parameters, each of which produces $O(1)$ recursive calls on smaller subproblems. Also, the decomposition has $O(\lg |V|)$ height, hence $|V|^{O(1)}$ nodes [Johnson 2003; Borie et al. 2002].

17.1. Algorithm for Minimum-Cardinality Vertex Cover in a *k*-HB Graph

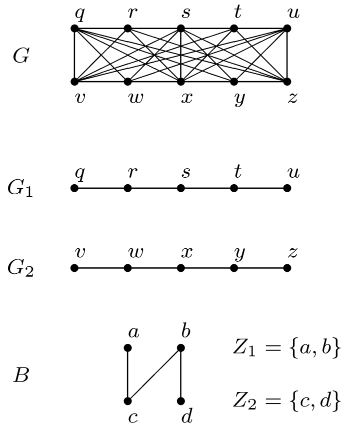
Define

$G[S]$ = min cardinality vertex cover that contains all vertices of $S \subseteq V$
$G.min$ = min cardinality vertex cover

The algorithm is shown in Figure 61. We demonstrate this algorithm on the 2-HB graph G as shown in Figure 62. Note that $G = G_1 \times_{B,h} G_2$, where G_1, G_2, B, and h are as shown. The top-level computations are summarized on the right. The minimum vertex cover has size 6, and the explicit solution is $\{q, s, u, v, x, z\}$.

17.2. Remarks

Top-down decomposition refers to a recognition algorithm that places a candidate graph at the root of a tree, and then decomposes this graph into smaller subgraphs that become its children in the tree, and so on recursively until reaching the leaves of the tree. A balanced decomposition is a decomposition tree that has height $O(\lg |V|)$. A



h

$h(r) = h(t) = a$
$h(q) = h(s) = h(u) = b$
$h(v) = h(x) = h(z) = c$
$h(w) = h(y) = d$

Recursive calls

$X = \emptyset \Rightarrow T = \emptyset \Rightarrow G_1[T] = 2$
$X = \{a\} \Rightarrow T = \{r, t\} \Rightarrow G_1[T] = 2$
$X = \{b\} \Rightarrow T = \{q, s, u\} \Rightarrow G_1[T] = 3$
$X = \{a, b\} \Rightarrow T = \{q, r, s, t, u\} \Rightarrow G_1[T] = 5$

$Y = \emptyset \Rightarrow U = \emptyset \Rightarrow G_2[U] = 2$
$Y = \{c\} \Rightarrow U = \{v, x, z\} \Rightarrow G_2[U] = 3$
$Y = \{d\} \Rightarrow U = \{w, y\} \Rightarrow G_2[U] = 2$
$Y = \{c, d\} \Rightarrow U = \{v, w, x, y, z\} \Rightarrow G_2[U] = 5$

Compatible subsets

$X = \emptyset, Y = \{c, d\} \Rightarrow G_1[T] + G_2[U] = 2 + 5 = 7$
$X = \{a\}, Y = \{c, d\} \Rightarrow G_1[T] + G_2[U] = 2 + 5 = 7$
$X = \{b\}, Y = \{c\} \Rightarrow G_1[T] + G_2[U] = 3 + 3 = 6$
$X = \{b\}, Y = \{c, d\} \Rightarrow G_1[T] + G_2[U] = 3 + 5 = 8$
$X = \{a, b\}, Y = \emptyset \Rightarrow G_1[T] + G_2[U] = 5 + 2 = 7$
$X = \{a, b\}, Y = \{c\} \Rightarrow G_1[T] + G_2[U] = 5 + 3 = 8$
$X = \{a, b\}, Y = \{d\} \Rightarrow G_1[T] + G_2[U] = 5 + 2 = 7$
$X = \{a, b\}, Y = \{c, d\} \Rightarrow G_1[T] + G_2[U] = 5 + 5 = 10$

Fig. 62. Minimum cardinality vertex cover in a 2-HB graph.

pseudo-NLC decomposition is similar to the standard NLC decomposition, except that the vertex labels used at one node in the tree are not enforced at other nodes. The requirement that the decomposition must be balanced is more restrictive, while simultaneously the pseudo-NLC condition is less restrictive. This trade-off yields the class of k-HB graphs. For more details see Johnson [2003] or Borie et al. [2002] .

Note that k-HB graphs are an ambiguously defined class due to the nondeterministic nature of this decomposition algorithm. On the other hand, the decomposition is guaranteed to succeed for every cliquewidth-k graph despite this nondeterminism, so every cliquewidth-k graph is a k-HB graph.

The k-HB decomposition algorithm also succeeds on some noncliquewidth-k graphs, and in fact the k-HB graphs do not have bounded cliquewidth. For example, construct a graph $G_{k,r}$ with vertex set $\{0, 1, \ldots, (2k)^r - 1\}$ and edge set as follows. Denote each vertex x as an r-digit number $x_1 x_2 \ldots x_r$ in base $2k$, and let the edges of $G_{k,r}$ be the vertex pairs (x, y) such that there exists an index j, $1 \leq j \leq r$, where $|x_j - y_j| = k$ and also $x_i < k$ iff $y_i < k$ for each index $i < j$. The resulting graph G has a k-HB decomposition with height $O(r) = O(\lg |V|)$, where each bipartite graph B is a perfect matching on $[k] \times [k]$; intuitively, the idea is to split the graph top-down, first using the most significant digit, then the second most significant digit, etc. But for $k \geq 2$, the cliquewidth of this graph $G_{k,r}$ grows with r, and so is unbounded for fixed k.

The chromatic number, dominating set, and Hamiltonian problems are not known to be solvable in polynomial time on k-HB graphs. Maximum matching is of course solvable in polynomial time on k-HB graphs, but it is not known whether this can be done more efficiently than for arbitrary graphs. Most problems that are known solvable in polynomial time for k-HB graphs are expressible in a particular language that we will call MSOL″.

MSOL″ for a graph G = (V, E) denotes a subset of MSOL′ restricted to variables v_i with domain V, and variables V_i with domain 2^V, and contains primitive predicates such as Adjacent(v_i, v_j) and $\mathrm{v}_i \in \mathrm{V}_j$. MSOL″ also permits the logical operators ($\neg$, $\wedge$, $\vee$) and quantifiers ($\exists$, $\forall$). However, these primitives and connectors cannot be combined in

$$\begin{aligned}&(\exists \mathsf{V}_1)\ldots(\exists \mathsf{V}_m)\ ((\forall \mathsf{v}_1)\ \mathsf{F}_0(\mathsf{v}_1{\in}\mathsf{V}_1,\ \ldots,\ \mathsf{v}_1{\in}\mathsf{V}_m)\\ &\qquad \wedge(\forall \mathsf{v}_2)\ (\forall \mathsf{v}_3)\ (\mathsf{Adjacent}(\mathsf{v}_2,\ \mathsf{v}_3) \rightarrow \wedge_{1\le i\le j\le m}\ \mathsf{F}_{ij}(\mathsf{v}_2{\in}\mathsf{V}_i,\ \mathsf{v}_3{\in}\mathsf{V}_j))\\ &\qquad \wedge(\forall \mathsf{v}_4)\ (\forall \mathsf{v}_5)\ (\neg\mathsf{Adjacent}(\mathsf{v}_4,\ \mathsf{v}_5) \rightarrow \wedge_{1\le i\le j\le m}\ \mathsf{F'}_{ij}(\mathsf{v}_4{\in}\mathsf{V}_i,\ \mathsf{v}_5{\in}\mathsf{V}_j)))\end{aligned}$$

Fig. 63. Format of MSOL″ predicates.

$$\begin{aligned}&\mathsf{VertexCover} \Leftrightarrow (\exists \mathsf{V}_1)\ (\forall \mathsf{v}_2)\ (\forall \mathsf{v}_3)\ (\mathsf{Adjacent}(\mathsf{v}_2,\ \mathsf{v}_3) \rightarrow (\mathsf{v}_2 \in \mathsf{V}_1 \vee \mathsf{v}_3 \in \mathsf{V}_1))\\ &\mathsf{IndependentSet} \Leftrightarrow (\exists \mathsf{V}_1)\ (\forall \mathsf{v}_2)\ (\forall \mathsf{v}_3)\ (\mathsf{Adjacent}(\mathsf{v}_2,\ \mathsf{v}_3) \rightarrow \neg\ (\mathsf{v}_2 \in \mathsf{V}_1 \wedge \mathsf{v}_3 \in \mathsf{V}_1))\\ &\mathsf{Clique} \Leftrightarrow (\exists \mathsf{V}_1)\ (\forall \mathsf{v}_4)\ (\forall \mathsf{v}_5)\ (\neg\ \mathsf{Adjacent}(\mathsf{v}_4,\ \mathsf{v}_5) \rightarrow \neg\ (\mathsf{v}_4 \in \mathsf{V}_1 \wedge \mathsf{v}_5 \in \mathsf{V}_1))\\ &\mathsf{VertexColoring}_m \Leftrightarrow (\exists \mathsf{V}_1)\ldots(\exists \mathsf{V}_m)\ ((\forall \mathsf{v}_1)\ (\mathsf{v}_1 \in \mathsf{V}_1 \vee \ldots \vee \mathsf{v}_1 \in \mathsf{V}_m)\\ &\qquad \wedge(\forall \mathsf{v}_2)\ (\forall \mathsf{v}_3)\ (\mathsf{Adjacent}(\mathsf{v}_2,\ \mathsf{v}_3) \rightarrow \wedge_{1\le i\le m}\ \neg\ (\mathsf{v}_2 \in \mathsf{V}_i \wedge \mathsf{v}_3 \in \mathsf{V}_i)))\\ &\mathsf{CliquePartition}_m \Leftrightarrow (\exists \mathsf{V}_1)\ldots(\exists \mathsf{V}_m)\ ((\forall \mathsf{v}_1)\ (\mathsf{v}_1 \in \mathsf{V}_1 \vee \ldots \vee \mathsf{v}_1 \in \mathsf{V}_m)\\ &\qquad \wedge(\forall \mathsf{v}_4)\ (\forall \mathsf{v}_5)\ (\neg\ \mathsf{Adjacent}(\mathsf{v}_4,\ \mathsf{v}_5) \rightarrow \wedge_{1\le i\le m}\ \neg\ (\mathsf{v}_4 \in \mathsf{V}_i \wedge \mathsf{v}_5 \in \mathsf{V}_i)))\end{aligned}$$

Fig. 64. Some MSOL″ predicates.

any arbitrary way; rather every MSOL″ expression must possess the format shown in Figure 63.

Here each F_0, each F_{ij}, and each $\mathrm{F'}_{ij}$ is an arbitrary formula that combines the indicated primitive predicates using operators ¬, ∧, and ∨. If any of these formulas is identically true, it may be omitted. An illustration follows.

As an example, the previous MSOL expressions for VertexCover, IndependentSet, Clique, VertexColoring$_m$, and CliquePartition$_m$ given in Figure 36 can be rewritten as equivalent MSOL″ expressions as shown in Figure 64. However, other MSOL expressions such as DominatingSet from Figure 36 do not appear to be expressible in MSOL″.

Every MSOL″-expressible problem can be solved in polynomial time when the input graph is restricted to any class of k-HB graphs [Johnson 2003; Borie et al. 2002]. This includes every cliquewidth-k graph, even if its decomposition tree is not provided as part of the input. Once a problem is expressed in MSOL″, the polynomial-time recursive algorithm can be created mechanically.

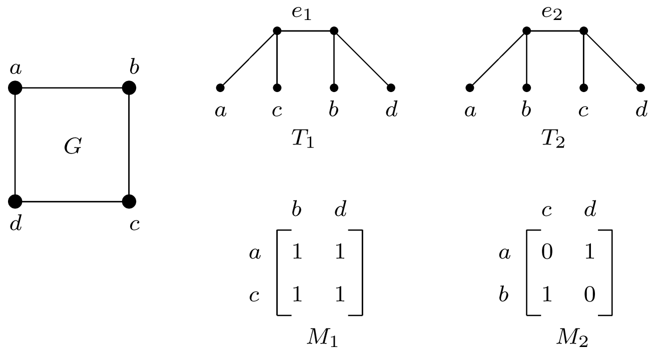
18. RANKWIDTH-*K* GRAPHS

Rankwidth is a relatively new graph parameter that is defined and studied in Oum and Seymour [2006] and Oum [2005a, 2005b]. It is similar to the concept of branchwidth, and it also closely relates to the concept of cliquewidth.

Let $G = (V, E)$ be a graph and consider disjoint vertex subsets $X, Y \subseteq V$. Let $M_{X,Y}(G)$ be a submatrix of the adjacency matrix of G, where rows correspond to X and columns to Y. Define $\mathrm{cutrank}_G(X, Y)$ to be the *rank* of the matrix $M_{X,Y}(G)$, that is, the dimension of the subspace spanned by its vectors. As a special case, define $\mathrm{cutrank}_G(X) = \mathrm{cutrank}_G(X, V - X)$.

Let T denote an unrooted ternary tree having $|V|$ leaves, and let f denote a bijection from the leaves of T to V. Then (T, f) is called a *rank decomposition* of G, which can be regarded as a kind of branch decomposition as follows. For each edge e of T, the connected components of $T - e$ induce a partition (X, Y) on the leaves of T. The *order* of edge e is $\mathrm{cutrank}_G(f(X))$, and the *width* of rank decomposition (T, f) is the maximum order over all edges of T. The *rankwidth* of G is the minimum width over all possible rank decompositions, and graph G is called a *rankwidth-k graph* if it has rankwidth at most k.

An example is shown in Figure 65. Graph G has rankwidth 1 as exhibited by tree T_1. Edge e_1 has order 1 because the matrix $M_1 = M_{\{a,c\},\{b,d\}}(G)$ has rank 1. Each other edge in tree T_1 obviously has order 1. However, not every decomposition produces the minimum width. For example, consider tree T_2 which yields width 2. Edge e_2 has order 2 because the matrix $M_2 = M_{\{a,b\},\{c,d\}}(G)$ has rank 2.



a
b
G
d
c
e_1
a c b d
T_1
e_2
a b c d
T_2
$$\begin{array}{c|cc} & b & d \\ \hline a & 1 & 1 \\ c & 1 & 1 \end{array}$$
M_1
$$\begin{array}{c|cc} & c & d \\ \hline a & 0 & 1 \\ b & 1 & 0 \end{array}$$
M_2

Fig. 65. A rankwidth-1 graph.

It is shown in Oum and Seymour [2006] that every cliquewidth-k graph is a rankwidth-k graph, and that every rankwidth-k graph is a cliquewidth-$(2^{k+1}-1)$ graph. Polynomial-time $O(|V|^9 \lg |V|)$ algorithms are also described that can find a $(3k+1)$ rank decomposition for any rankwidth-k graph, and a $(2^{3k+2}-1)$ cliquewidth decomposition for any cliquewidth-k graph. If such decompositions cannot be found for a given graph, then the algorithms correctly determine that the graph is not rankwidth-k or cliquewidth-k, respectively.

In Oum [2005a] new algorithms are presented that improve the running time from $O(|V|^9 \lg |V|)$ to $O(|V|^4)$. Also, faster $O(|V|^3)$ algorithms are presented, but there is a trade-off: The rankwidth to $24k$ and the cliquewidth to $(2^{24k+1}-1)$.

So, although the status of the recognition problem for cliquewidth-k graphs remains open, the results on rankwidth-k graphs imply significant practical progress. Specifically, all MSOL′ problems are now solvable in polynomial time on cliquewidth-k graphs (and also rankwidth-k graphs), even if the decompositions are not explicitly given along with the input.

18.1. Remarks

The class of rankwidth-1 graphs is identical to the class of distance-hereditary graphs, and thus rankwidth-1 graphs are also perfect graphs. So many problems can be solved in polynomial time on rankwidth-1 graphs.

However, it is not currently known how to design efficient algorithms for solving problems directly on rankwidth-k graphs. Of course, we could reduce a problem on rankwidth-k graphs to a problem on cliquewidth-k' graphs, where k' grows exponentially with k. But it remains an interesting open question how to solve directly on rankwidth-k graphs.

19. SUMMARY AND CONCLUDING REMARKS

In Figure 66, we present a schematic depicting key relationships involving all of the graph classes covered in this tutorial. In the schematic, an arrow directed from a node depicting graph class x to another node depicting class y and labelled by k' is meant to denote that a well-defined relationship exists between the pair such that a class x graph with generic parameter k is a class y graph with parameter k'. For example, every treewidth-k graph is a branchwidth-$(k+1)$ graph; cographs are cliquewidth-2 graphs, etc. A double-headed arrow indicates an equivalence between particular classes; for example, partial k-trees and treewidth-k graphs.

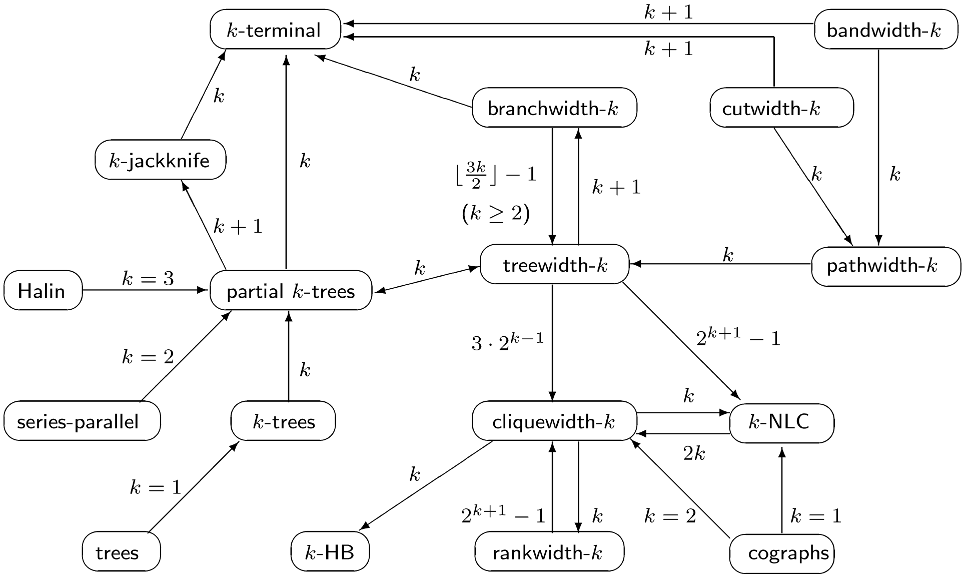


Fig. 66. Relationships between recursive graph classes.

It should be evident that the schematic in Figure 66 is not, strictly speaking, complete in that a number of relationships are not explicitly exhibited. However, most such relations can be deduced from the figure by applying the given relations transitively. For example, a directed arc from trees to treewidth-k with label $k = 1$ is not shown, but it can be determined from the arcs shown in the figure that every tree is a treewidth-1 graph.

Next, it may be instructive to remark why no arcs exit the node identified with k-terminal graphs in Figure 66. As indicated previously in this article, this follows since k-terminal graphs are not well defined until composition operations are precisely specified; that is, every graph is essentially a k-terminal graph for *some* set of k-terminal operations. For example, any K_p with $p \geq 4$ can be correctly viewed as a 2-terminal graph where an operation is defined that splits K_p into $\frac{p(p-1)}{2}$ K_2 subgraphs. However, such K_p are neither 2-jackknife nor branchwidth-2 graphs, for example. On the other hand, if we had restricted our attention to k-terminal graph classes that only permit *binary* operations (rather than arbitrary m-ary operations), then such arcs could have been added to Figure 66. Also note that any particular class of k-terminal graphs *is* a subset of the treewidth-k' graphs for some k', but such k' depends on the particular set of recursive k-terminal composition operations defined for this class, so k' cannot be expressed as a function of k.

In concluding, it is important to reiterate our intent that this tutorial should at least serve to corroborate the explosive growth in the literature pertaining to recursively constructible graph classes, especially over the last two decades. But this growth has also been uneven and, arguably, it has been somewhat disparate as well. While many results have proven interesting and important in their own right, others have been particularly profound, facilitating the discovery of elegant results in other, more general areas in graph theory. Much of this work has derived from a variety of underlying themes, motivations, and impetuses. From an algorithmic perspective, however, it is the case that the literature has lagged and has not been nearly so rich in exposing how graph problems are actually solved on many of these classes. Further, the extent to which such work has been produced reveals that the preponderance of literature has favored results associated with algorithms on more primitive classes such as series-parallel graphs. Thus, the main point of this tutorial is to rectify this condition somewhat by not only providing some consolidation of the myriad results that have been produced in the last 20 to 25 years, but to explicitly demonstrate how algorithms are created to solve problems on a greater expanse of these interesting classes.

Chief among any flaws in this coverage would probably be an unevenness relative to the choice and breadth of illustrations employed across the numerous graph classes included. However, there was a warning at the outset that this option would be taken. Most certainly, it would have been desirable to include as many sample algorithms for, say, the classes of treewidth-k graphs, partial k-trees, cliquewidth-k graphs, and others, as, say, for trees or series-parallel graphs; however, what is already a lengthy document would have swelled substantially. In short, we opted to err on the side of the simpler classes where algorithms were somewhat less detailed but where the fundamental structure of algorithm creation was meaningful insofar as their extension to more sophisticated classes was concerned.

So, in any extended version of this tutorial, it is natural then to imagine a work displaying an even greater variety of algorithms and explicit illustrations. In addition, the notion of looking more critically at various formalisms that have been structured for problems on these recursive graph classes would also seem to be worthwhile. Our coverage in this regard was very brief and only marginally demonstrated the richness of what are some interesting results involving formal models. A deeper exposition of some of these results would have merit as well.

ACKNOWLEDGMENTS

We wish to thank the anonymous referees for their many helpful suggestions.

REFERENCES

AMIR, E. 2001. Efficient approximation for triangulation of minimum treewidth. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*, 7–15.

ARNBORG, S. 1985. Efficient algorithms for combinatorial problems on graphs with bounded decomposibility: A survey. *Bit 25*, 2–23.

ARNBORG, S., CORNEIL, D., AND PROSKUROWSKI, A. 1987. Complexity of finding embeddings in a k-tree. *SIAM J. Algebr. Discrete Methods 8*, 277–284.

ARNBORG, S., COURCELLE, B., PROSKUROWSKI, A., AND SEESE, D. 1993. An algebraic theory of graph reductions. *J. ACM 40*, 1134–1164.

ARNBORG, S., HEDETNIEMI, S., AND PROSKUROWSKI, A. 1994. Efficient algorithms and partial k-trees. *Discrete Appl. Math. 54*, 2–3. Guest editors of special issue.

ARNBORG, S., LAGERGREN, J., AND SEESE, D. 1991. Easy problems for tree-decomposable graphs. *J. Algor. 12*, 308–340.

ARNBORG, S. AND PROSKUROWSKI, A. 1985. Characterization and recognition of partial k-trees. *Congressus Numer. 47*, 69–75.

ARNBORG, S. AND PROSKUROWSKI, A. 1986. Characterization and recognition of partial 3-trees. *SIAM J. Algebr. Discrete Methods 7*, 305–314.

ARNBORG, S. AND PROSKUROWSKI, A. 1989. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Appl. Math. 23*, 11–24.

ARNBORG, S., PROSKUROWSKI, A., AND CORNEIL, D. 1990. Forbidden minors characterization of partial 3-trees. *Discrete Math. 80*, 1–19.

BABEL, L. AND OLARIU, S. 1998. On the structure of graphs with few P_4s. *Discrete Appl. Math. 84*, 1–13.

BACHOORE, E. AND BODLAENDER, H. 2006. A branch-and-bound algorithm for exact, upper, and lower bounds on treewidth. In *Proceedings of the 2nd International Conference on Algorithmic Aspects in Information and Management*. Lecture Notes in Computer Science, vol. 4041. Springer, 255–266.

BECKER, A. AND GEIGER, D. 2001. A sufficiently fast algorithm for finding close to optimal clique trees. *Artif. Intell. 125*, 3–17.

BEINEKE, L. AND PIPPERT, R. 1971. Properties and characterizations of k-trees. *Math. 18*, 141–151.

BERN, M., LAWLER, E., AND WONG, A. 1987. Linear time computation of optimal subgraphs of decomposable graphs. *J. Algor. 8*, 216–235.

BIENSTOCK, D., ROBERTSON, N., SEYMOUR, P., AND THOMAS, R. 1991. Quickly excluding a forest. *J. Combinatorial Theory Series B 52*, 274–283.

BLAIR, J., HEGGERNES, P., AND TELLE, J. 2001. A practical algorithm for making filled graphs minimal. *Theor. Comput. Sci. 250*, 125–141.

BODLAENDER, H. 1987. Dynamic programming on graphs with bounded tree-width. Tech. Rep., Massachusetts Institute of Technology.

BODLAENDER, H. 1988. Dynamic programming algorithms on graphs with bounded tree-width. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 317. Springer, 105–109.

BODLAENDER, H. 1990. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *J. Algor. 11*, 631–643.

BODLAENDER, H. 1993. A tourist guide through treewidth. *Acta Cybernetica 11*, 1–23.

BODLAENDER, H. 1996. A linear-time algorithm for finding tree decompositions of small treewidth. *SIAM J. Comput. 25*, 1305–1317.

BODLAENDER, H. 1998. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci. 209*, 1–45.

BODLAENDER, H., FOMIN, F., KOSTER, A., KRATSCH, D., AND THILIKOS, D. 2006. On exact algorithms for treewidth. In *Proceedings of the 14th Annual European Symposium on Algorithms*. Lecture Notes in Computer Science, vol. 4168. Springer, 672–683.

BODLAENDER, H., GILBERT, J., HAFSTEINSSON, H., AND KLOKS, T. 1995. Approximating treewidth, pathwidth, and minimum elimination tree height. *J. Algor. 18*, 238–255.

BODLAENDER, H., GUSTEDT, J., AND TELLE, J. 1998. Linear-Time register allocation for a fixed number of registers and no stack variables. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, 574–583.

BODLAENDER, H. AND KLOKS, T. 1996. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algor. 21*, 358–402.

BODLAENDER, H. AND KOSTER, A. 2006. Safe separators for treewidth. *Discrete Math. 306*, 337–350.

BODLAENDER, H., KOSTER, A., AND VAN DEN EIJKHOF, F. 2005. Pre-Processing rules for triangulation of probabilistic networks. *Comput. Intell. 21*, 286–305.

BODLAENDER, H., KOSTER, A., AND WOLLE, T. 2006. Contraction and treewidth lower bounds. *J. Graph Algor. Appl. 10*, 5–49.

BODLAENDER, H. AND MOHRING, R. 1993. The pathwidth and treewidth of cographs. *SIAM J. Discrete Math. 6*, 181–186.

BORIE, R. 1988. Recursively constructed graph families: Membership and linear algorithms. Ph.D. thesis, Georgia Institute of Technology.

BORIE, R. 1995. Generation of polynomial-time algorithms for some optimization problems on tree-decomposable graphs. *Algorithmica 14*, 123–137.

BORIE, R., JOHNSON, J., RAGHAVAN, V., AND SPINRAD, J. 2002. Robust polynomial time algorithms on cliquewidth-k graphs. Manuscript.

BORIE, R., PARKER, R., AND TOVEY, C. 1991a. Algorithms for recognition of regular properties and decomposition of recursive graph families. *Annals Oper. Res. 33*, 127–149.

BORIE, R., PARKER, R., AND TOVEY, C. 1991b. Deterministic decomposition of recursive graph classes. *SIAM J. Discrete Math. 4*, 481–501.

BORIE, R., PARKER, R., AND TOVEY, C. 1992. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica 7*, 555–581.

BORIE, R., PARKER, R., AND TOVEY, C. 2004. Algorithms on recursively constructed graphs. In *Handbook of Graph Theory*, J. Gross and J. Yellen, eds. CRC Press, Chapter 10.4, 1046–1066.

BOUCHITTÉ, V., KRATSCH, D., MÜLLER, H., AND TODINCA, I. 2004. On treewidth approximations. *Discrete Appl. Math. 136*, 183–196.

BRANDSTADT, A., LE, V., AND SPINRAD, J. 1999. *Graph Classes: A Survey*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, PA.

Clautiaux, F., Carlier, J., Moukrim, A., and Negre, S. 2003. New lower and upper bounds for graph treewidth. In *Proceedings of the 2nd International Workshop on Experimental and Efficient Algorithms*. Lecture Notes in Computer Science, vol. 2647. Springer, 70–80.

Clautiaux, F., Moukrim, A., Negre, S., and Carlier, J. 2004. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Oper. Res. 38*, 13–26.

Corneil, D., Habib, M., Lanlignel, J., Reed, B., and Rotics, U. 2000. Polynomial-Time recognition of clique-width ≤ 3 graphs. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*. Lecture Notes in Computer Science, vol. 1776. Springer, 126–134.

Corneil, D. and Kirkpatrick, D. 1983. Families of recursively defined perfect graphs. *Congressus Numer. 39*, 237–246.

Corneil, D., Lerchs, H., and Burlington, L. 1981. Complement reducible graphs. *Discrete Appl. Math. 3*, 163–174.

Corneil, D., Perl, Y., and Stewart, L. 1984. Cographs: Recognition, applications and algorithms. *Congressus Numer. 43*, 249–258.

Corneil, D., Perl, Y., and Stewart, L. 1985. A linear recognition algorithm for cographs. *SIAM Journal on Comput. 14*, 926–934.

Corneil, D. and Rotics, U. 2005. On the relationship between clique-width and treewidth. *SIAM J. Comput. 34*, 825–847.

Cornuejols, G., Naddef, D., and Pulleyblank, W. 1983. Halin graphs and the travelling salesman problem. *Math. Program. 26*, 287–294.

Courcelle, B. 1990. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Inf. Comput. 85*, 12–75.

Courcelle, B. 1992. The monadic second-order logic of graphs III: Tree-Decompositions, minors, and complexity issues. *Theor. Inf. Appl. 26*, 257–286.

Courcelle, B. 1995. The monadic second-order logic of graphs VIII: Orientations. *Annals Pure Appl. Logic 72*, 103–143.

Courcelle, B. 1996. The monadic second-order logic of graphs X: Linear orderings. *Theor. Comput. Sci. 160*, 87–143.

Courcelle, B., Engelfriet, J., and Rozenberg, G. 1993. Handle-Rewriting hypergraph grammars. *J. Comput. Syst. Sci. 46*, 218–270.

Courcelle, B., Makowsky, J., and Rotics, U. 2000. Linear time solvable optimization problems on graphs of bounded clique width. *Theory Comput. Syst. 33*, 125–150.

Courcelle, B., Makowsky, J., and Rotics, U. 2001. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Appl. Math. 108*, 23–52.

Courcelle, B. and Mosbah, M. 1993. Monadic second-order evaluations on tree-decomposable graphs. *Theor. Comput. Sci. 109*, 49–82.

Courcelle, B. and Olariu, S. 2000. Upper bounds to the clique-width of graphs. *Discrete Appl. Math. 101*, 77–114.

de Fluiter, B. 1997. Algorithms for graphs of small treewidth. Ph.D. thesis, University of Utrecht.

Duffin, R. 1965. Topology of series-parallel networks. *J. Math. Anal. Appl. 10*, 303–318.

Edmonds, J. 1965a. Maximum matching and polyhedron of 0,1 vertices. *J. Res. National Bureau Standards 69B*, 125–130.

Edmonds, J. 1965b. Paths, trees, and flowers. *Canadian J. Math. 17*, 449–467.

Egervary, E. 1931. On combinatorial properties of matrices. *Math. Phys. Pages 38*, 16–28.

El-Mallah, E. and Colbourn, C. 1988. Partial k-tree algorithms. *Congressus Numer. 64*, 105–119.

Espelage, W., Gurski, F., and Wanke, E. 2001. How to solve NP-hard graph problems on clique-width bounded graphs in polynomial time. In *Proceedings of the 27th International Workshop on Graph Theory*. Lecture Notes in Computer Science, vol. 2204. Springer, 117–128.

Feige, U., Hajiaghayi, M., and Lee, J. 2005. Improved approximation algorithms for minimum-weight vertex separators. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, 563–572.

Garey, M., Graham, R., Johnson, D., and Knuth, D. 1978. Complexity results for bandwidth minimization. *SIAM J. Appl. Math. 34*, 477–495.

Gavril, F. 1972. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput. 1*, 180–187.

Gogate, V. and Dechter, R. 2004. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence*, 201–208.

Golumbic, M. and Rotics, U. 1999. On the clique-width of perfect graph classes. In *Proceedings of the 25th International Workshop on Graph Theory*. Lecture Notes in Computer Science, vol. 1665. Springer, 135–147.

Granot, D. and Skorin-Kapov, D. 1991. NC algorithms for recognizing partial 2-trees and 3-trees. *SIAM J. Algebr. Discrete Methods 4*, 342–354.

Gurski, F. and Wanke, E. 2000. The tree-width of clique-width bounded graphs without $K_{n,n}$. In *Proceedings of the 26th International Workshop on Graph Theory*. Lecture Notes in Computer Science, vol. 1928. Springer, 196–205.

Gurski, F. and Wanke, E. 2004. Vertex disjoint paths on clique-width bounded graphs. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics*. Lecture Notes in Computer Science, vol. 2976. Springer, 119–128.

Gurski, F. and Wanke, E. 2006. Vertex disjoint paths on clique-width bounded graphs. *Theor. Comput. Sci. 359*, 188–199.

Hare, E., Hedetniemi, S., Laskar, R., Peters, K., and Wimer, T. 1987. Linear-time computability of combinatorial problems on generalized series-parallel graphs. *Discrete Algor. Complexity 14*, 437–457.

He, X. and Yesha, Y. 1987. Parallel recognition and decomposition of two-terminal series-parallel graphs. *Inf. Comput. 75*, 15–38.

Hicks, I., Koster, A., and Kolotoğlu, E. 2005. Branch and tree decomposition techniques for discrete optimization. In *TutORials 2005*, J. Smith, ed. Tutorials in Operations Research. INFORMS, New Orleans, LA, 1–29.

Horton, S., Parker, R., and Borie, R. 1992. On some results pertaining to Halin graphs. *Congressus Numer. 93*, 65–86.

Isobe, S., Zhou, X., and Nishizeki, T. 1999. A polynomial-time algorithm for finding total colorings of partial k-trees. *Int. J. Foundat. Comput. Sci. 10*, 171–194.

Ito, T., Nishizeki, T., and Zhou, X. 2003. Algorithms for multicolorings of partial k-trees. *IEICE Trans. Inf. Syst. E86-D*, 191–200.

Jamison, B. and Olariu, S. 1995. Linear time optimization algorithms for P_4-sparse graphs. *Discrete Appl. Math. 61*, 155–175.

Jensen, F., Lauritzen, S., and Olesen, K. 1990. Bayesian updating in recursive graphical models by local computation. *Computat. Statist. Q. 4*, 269–282.

Johansson, O. 1998. Clique-Decomposition, NLC-decomposition, and modular decomposition relationships and results for random graphs. *Congressus Numer. 132*, 39–60.

Johansson, O. 2000. NLC 2-decomposition in polynomial time. *Int. J. Foundat. Comput. Sci. 11*, 373–395.

Johnson, J. 2003. Polynomial time recognition and optimization algorithms on special classes of graphs. Ph.D. thesis, Vanderbilt University.

Kajitani, Y., Ishizuka, A., and Ueno, S. 1985. A characterization of the partial k-tree in terms of certain structures. In *Proceedings of the International Symposium on Circuits and Systems*. IEEE, 1179–1182.

Karp, R. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. Plenum Press, New York, 85–103.

Kashem, M., Zhou, X., and Nishizeki, T. 2000. Algorithms for generalized vertex-rankings of partial k-trees. *Theoret. Comput. Sci. 240*, 407–427.

Kassios, I. 2001. Translating Borie-Parker-Tovey calculus into mutumorphisms. Manuscript.

Klarlund, N. 1998. Mona and Fido: The logic-automaton connection in practice. In *Computer Science Logic 1997*. Lecture Notes in Computer Science, vol. 1414. Springer, 311–326.

Klarlund, N., Moller, A., and Schwartzbach, M. 2002. Mona implementation secrets. *Int. J. Foundat. Comput. Sci. 13*, 571–586.

Kloks, T. 1994. *Treewidth, Computations and Approximations*. Lecture Notes in Computer Science, vol. 842. Springer.

Kloks, T. and Kratsch, D. 1995. Treewidth of chordal bipartite graphs. *J. Algor. 19*, 266–281.

Kobler, D. and Rotics, U. 2003. Edge dominating set and colorings on graphs with fixed clique-width. *Discrete Appl. Math. 126*, 197–221.

Koster, A. 1999. Frequency assignment—Models and algorithms. Ph.D. thesis, Maastricht University.

Lauritzen, S. and Spiegelhalter, D. 1988. Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Statist. Soc. Series B 50*, 157–224.

Lucena, B. 2003. A new lower bound for tree-width using maximum cardinality search. *SIAM J. Discrete Math. 16*, 345–353.

Makowsky, J., Rotics, U., Averbouch, I., and Godlin, B. 2006. Computing graph polynomials on graphs of bounded clique-width. In *Proceedings of the 32nd International Workshop on Graph Theory*. Lecture Notes in Computer Science, vol. 4271. Springer, 191–204.

Matousek, J. and Thomas, R. 1991. Algorithms for finding tree decompositions of graphs. *J. Algor. 12*, 1–22.

Oum, S. 2005a. Approximating rank-width and clique-width quickly. In *Proceedings of the 31st International Workshop on Graph Theory*. Lecture Notes in Computer Science, vol. 3787. Springer, 49–58.

Oum, S. 2005b. Graphs of bounded rank-width. Ph.D. thesis, Princeton University.

Oum, S. and Seymour, P. 2006. Approximating clique-width and branch-width. *J. Combinatorial Theory Series B 96*, 514–528.

Proskurowski, A. 1993. Graph reductions, and techniques for finding minimal forbidden minors. *Graph Structure Theory 147*, 591–600.

Rardin, R. and Parker, R. 1986. Subgraph isomorphism on partial 2-trees. Tech. Rep., Georgia Institute of Technology.

Reed, B. 1992. Finding approximate separators and computing treewidth quickly. In *Proceedings of the 24th Annual Symposium on Theory of Computing*. ACM, 221–228.

Reed, B. 1997. Treewidth and tangles: A new connectivity measure and some applications. In *Surveys in Combinatorics*. London Mathematical Society Lecture Note Series, vol. 241. Cambridge University Press, London, 87–162. Invited papers from 16th British Combinatorial Conference.

Richey, M. 1985. Combinatorial optimization on series-parallel graphs: Algorithms and complexity. Ph.D. thesis, Georgia Institute of Technology.

Robertson, N. and Seymour, P. 1983. Graph minors I: Excluding a forest. *J. Combinatorial Theory Series B 35*, 39–61.

Robertson, N. and Seymour, P. 1984. Graph minors III: Planar treewidth. *J. Combinatorial Theory Series B 36*, 49–64.

Robertson, N. and Seymour, P. 1986a. Graph minors II: Algorithmic aspects of treewidth. *J. Algor. 7*, 309–322.

Robertson, N. and Seymour, P. 1986b. Graph minors V: Excluding a planar graph. *J. Combinatorial Theory Series B 41*, 92–114.

Robertson, N. and Seymour, P. 1986c. Graph minors VI: Disjoint paths across a disc. *J. Combinatorial Theory Series B 41*, 115–138.

Robertson, N. and Seymour, P. 1988. Graph minors VII: Disjoint paths on a surface. *J. Combinatorial Theory Series B 45*, 212–254.

Robertson, N. and Seymour, P. 1990a. Graph minors IV: Treewidth and well-quasi-ordering. *J. Combinatorial Theory Series B 48*, 227–254.

Robertson, N. and Seymour, P. 1990b. Graph minors IX: Disjoint crossed paths. *J. Combinatorial Theory Series B 49*, 40–77.

Robertson, N. and Seymour, P. 1990c. Graph minors VIII: A Kuratowski theorem for general surfaces. *J. Combinatorial Theory Series B 48*, 255–288.

Robertson, N. and Seymour, P. 1991. Graph minors X: Obstructions to tree decompositions. *J. Combinatorial Theory Series B 52*, 153–190.

Robertson, N. and Seymour, P. 1992. Graph minors XXII: Irrelevant vertices in linkage problems. Manuscript.

Robertson, N. and Seymour, P. 1994. Graph minors XI: Distance on a surface. *J. Combinatorial Theory Series B 60*, 72–106.

Robertson, N. and Seymour, P. 1995. Graph minors XIII: The disjoint paths problem. *J. Combinatorial Theory Series B 63*, 65–110.

Robertson, N. and Seymour, P. 2003. Graph minors XVI: Excluding a non-planar graph. *J. Combinatorial Theory Series B 89*, 43–76.

Robertson, N. and Seymour, P. 2004. Graph minors XX: Wagner's conjecture. *J. Combinatorial Theory Series B 92*, 325–357.

Robertson, N., Seymour, P., and Thomas, R. 1994. Quickly excluding a planar graph. *J. Combinatorial Theory Series B 62*, 323–348.

Rose, D. 1974. On simple characterization of k-trees. *Discrete Math. 7*, 317–322.

Sanders, D. 1993. Linear algorithms for graphs of tree-width at most four. Ph.D. thesis, Georgia Institute of Technology.

Sanders, D. 1996. On linear recognition of tree-width at most four. *SIAM J. Discrete Mathematics 9*, 101–117.

SASANO, I., HU, Z., TAKEICHI, M., AND OGAWA, M. 2000. Make it practical: A generic linear-time algorithm for solving maximum-weightsum problems. *ACM SIGPLAN Not. 35*, 137–149.

SATYANARAYANA, A. AND TUNG, L. 1990. A characterization of partial 3-trees. *Netw. 20*, 299–322.

SCHEFFLER, P. 1987. Linear-Time algorithms for NP-complete problems restricted to partial k-trees. Tech. Rep. R-MATH-03/87, Akademie der Wissenschaften der DDR.

SCHEFFLER, P. 1988. What graphs have bounded treewidth? In *Fischland Colloquium on Discrete Mathematics and Applications*.

SCHEFFLER, P. 1989. The treewidth of graphs as a measure for the complexity of algorithmic problems. Ph.D. thesis, German Academy of Sciences Berlin.

SCHEFFLER, P. AND SEESE, D. 1986. Graphs of bounded tree-width and linear-time algorithms for NP-complete problems. In *Bilateral Seminar*.

SCHEFFLER, P. AND SEESE, D. 1988. A combinatorial and logical approach to linear-time computability. In *Proceedings of the European Conference on Computer Algebra*. Lecture Notes in Computer Science, vol. 378. Springer, 379–380.

SEYMOUR, P. AND THOMAS, R. 1993. Graph searching and a min-max theorem for treewidth. *J. Combinatorial Theory Series B 58*, 22–33.

SEYMOUR, P. AND THOMAS, R. 1994. Call routing and the ratcatcher. *Combinatorica 14*, 217–241.

SPINRAD, J. 2003. *Efficient Graph Representations*. Fields Institute Monographs. AMS, Brooklyn, NY.

SYSLO, M. 1983. NP-Complete problems on some tree-structured graphs: A review. In *Proceedings of the 9th Workshop on Graph-Theoretic Concepts in Computer Science*, 342–353.

TAKAMIZAWA, K., NISHIZEKI, T., AND SAITO, N. 1982. Linear-Time computability of combinatorial problems on series-parallel graphs. *J. ACM 29*, 623–641.

TELLE, J. AND PROSKUROWSKI, A. 1993. Efficient sets in partial k-trees. *Discrete Appl. Math. 44*, 109–117.

TELLE, J. AND PROSKUROWSKI, A. 1997. Algorithms for vertex partitioning problems on partial k-trees. *SIAM J. Discrete Math. 10*, 529–550.

THORUP, M. 1998. All structured programs have small tree-width and good register allocation. *Inf. Comput. 142*, 159–181.

VALDES, J., TARJAN, R., AND LAWLER, E. 1982. The recognition of series parallel digraphs. *SIAM J. Comput. 11*, 298–313.

VIZING, V. 1964. On an estimate of the chromatic class of a p-graph. *Discrete Anal. 3*, 25–30.

WANKE, E. 1994. k-NLC graphs and polynomial algorithms. *Discrete Appl. Math. 54*, 251–266. Later revised with new co-author F. Gurski.

WIMER, T. 1987. Linear algorithms on k-terminal recursive graphs. Ph.D. thesis, Clemson University.

WIMER, T. AND HEDETNIEMI, S. 1988. K-terminal recursive families of graphs. *Congressus Numer. 63*, 161–176.

WIMER, T., HEDETNIEMI, S., AND LASKAR, R. 1985. A methodology for constructing linear graph algorithms. *Congressus Numer. 50*, 43–60.

ZHOU, X., FUSE, K., AND NISHIZEKI, T. 2000. A linear algorithm for finding [g,f]-colorings of partial k-trees. *Algorithmica 27*, 227–243.

ZHOU, X., KANARI, Y., AND NISHIZEKI, T. 2000. Generalized vertex-colorings of partial k-trees. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E83-A*, 671–678.

ZHOU, X., NAKANO, S., AND NISHIZEKI, T. 1993. A linear algorithm for edge-coloring partial k-trees. In *Proceedings of the 1st Annual European Symposium on Algorithms*. Lecture Notes in Computer Science, vol. 726. Springer, 409–418.

ZHOU, X., NAKANO, S., AND NISHIZEKI, T. 1996. Edge-Coloring partial k-trees. *J. Algor. 21*, 598–617.

Received July 2007; revised October 2007; accepted January 2008