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 paper 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 k-terminal graphs, trees, series-parallel graphs, k-trees, partial k-trees, Halin graphs, bandwidth-k graphs, pathwidth-k graphs, treewidth-k graphs, branchwidth-k graphs, cographs, cliquewidth-k graphs, k-NLC graphs, k-HB graphs, and rankwidth-k graphs. The definition of each class is provided, after which some typical algorithms are applied to solve problems on instances of each class.

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, Halin graph, pathwidth, rankwidth, series parallel, tree, treewidth

1. INTRODUCTION

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. Each operation involves fusing specific vertices from each subgraph or adding new edges between specific vertices from each subgraph. Each graph in a recursive class has a 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. Solve the problem on the base graphs defined for the given class, and then combine the

R. B. Borie, Department of Computer Science, University of Alabama, Box 870290, Tuscaloosa, AL 35487-0290. 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.

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. 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. 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. Also key is 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.

Organizationally, the paper proceeds as follows. Each section is devoted to a specific graph class, beginning with the general class of k-terminal graphs. In each, the class is defined and some algorithms are formally stated or at least described in sufficient detail. In addition, at least one explicit example is given, i.e., at least one problem (independent set, vertex cover, minimum-maximal matching, etc.) is identified and its solution is demonstrated on an instance from the respective graph class. The only case not consistent with this template is the class of rankwidth-k graphs (c.f. Section 15). In most of our examples, problem selection is confined to cases where the specific problem is hard on arbitrary graphs, i.e., $\mathcal{NP}$-hard/$\mathcal{NP}$-complete. One exception is the maximum matching problem, which can be solved in polynomial time for arbitrary graphs, but which can be solved more efficiently (i.e., in linear time) for many of our graph classes. Note also 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" cases such as trees and series-parallel graphs. 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.

Most sections are concluded with some general comments pertaining to the specific graph class. The detail in this coverage will vary broadly from section to section. In some cases, comments will describe structural, graph-theoretic attributes relevant to the graph class; sometimes, the comments 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 were dealt with earlier as well as ones covered in subsequent sections. Readers may choose to skip such remarks since doing so does not adversely affect the primary content of the tutorial.

In algorithmic descriptions, we shall generally employ notation where $G.x$ denotes an attribute x (corresponding to an equivalence class) of a given graph G. It is crucial that within most algorithms, $G.x$ carries only $O(1)$ pieces of information, rather than a complete description of the equivalence class. When we write, for

example, $G.x$ = *maximum-cardinality independent set*, then $G.x$ carries with it two pieces of information: primarily, the size of a maximum cardinality independent set, 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 pseudo-code 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. K-TERMINAL GRAPHS

It is natural to expect that many classes including ones rarely defined in recursive terms (e.g., trees, Halin graphs) can nonetheless be described by more general and unifying definitions that legitimize their identity as recursively constructed graphs. This will be the case with many of the classes that follow. In this first section, we facilitate this by introducing the notion of k-terminal graphs.

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 $|T| \leq k$. 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 $V = T$, i.e., 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 $1 \leq i \leq m$, 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 $f : \cup_{1 \leq i \leq m} V_i \rightarrow V$ 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: $v_1 \in V_{i_1}, v_2 \in V_{i_2}, i_1 \neq i_2, f(v_1) = f(v_2)$ implies $v_1 \in T_{i_1}, v_2 \in T_{i_2}$; and

— edges are preserved: $(\exists i)(\{v_1, v_2\} \in E_i)$ if and only if $\{f(v_1), f(v_2)\} \in E$.

If f is a valid vertex mapping, the corresponding m-ary composition operation (denoted by f) is generally written $f(G_1, \ldots, G_m) = G$.

To illustrate, consider a case with $k = 2$; the base graphs are edges with both vertices specified as terminals, i.e., $B = \{K_2\}$. 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 *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, $\{t_{11}, t_{21}\}$).

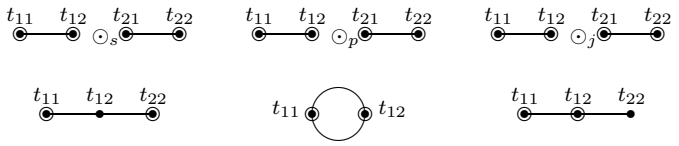


Fig. 1. Series, parallel, and jackknife composition for 2-terminal graphs

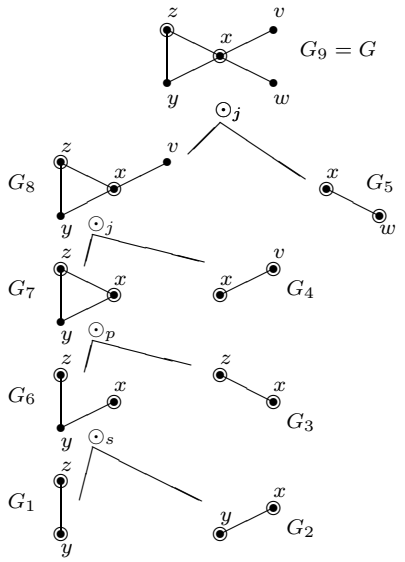


Fig. 2. A 2-terminal graph composition using series, parallel, and jackknife operations

Figure 1 illustrates the three composition rules. As will be the case throughout, terminals are identified by circled vertices.

A specific example is demonstrated in Figure 2. Starting with base graphs (leaves of the tree shown) G_1, G_2, G_3, G_4, and G_5, the successive application of the composition rules $\odot_s, \odot_p$, and $\odot_j$ ultimately produce the graph G_9 as indicated.

2.1 Algorithms

For a given k-terminal graph it is important to determine its decomposition tree; that is, to understand how the graph is constructed by a legal application of a set of composition rules applied to a fixed base graph set. 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. For $G = G_9$ in Figure 2, the tree depicted is a decomposition tree for G. For a graph in some recursive class, a corresponding decomposition tree may not be unique.

Now, as indicated in the introduction, if we are given a decomposition tree for some graph G, it is sufficient that we solve the problem of interest on G by successively merging solutions or potential solutions from the constituent subgraphs as per the stated composition rules for the class, ultimately reaching a solution for G,

Series composition $G \leftarrow G_1 \odot_s G_2$	Parallel composition $G \leftarrow G_1 \odot_p G_2$	Jackknife composition $G \leftarrow G_1 \odot_j G_2$
$G.a \leftarrow (G_1.a \wedge G_2.a) \vee (G_1.b \wedge G_2.c)$	$G.a \leftarrow (G_1.a \wedge G_2.a)$	$G.a \leftarrow (G_1.a \wedge G_2.a) \vee (G_1.a \wedge G_2.b)$
$G.b \leftarrow (G_1.a \wedge G_2.b) \vee (G_1.b \wedge G_2.d)$	$G.b \leftarrow (G_1.b \wedge G_2.b)$	$G.b \leftarrow (G_1.b \wedge G_2.c) \vee (G_1.b \wedge G_2.d)$
$G.c \leftarrow (G_1.c \wedge G_2.a) \vee (G_1.d \wedge G_2.c)$	$G.c \leftarrow (G_1.c \wedge G_2.c)$	$G.c \leftarrow (G_1.c \wedge G_2.a) \vee (G_1.c \wedge G_2.b)$
$G.d \leftarrow (G_1.c \wedge G_2.b) \vee (G_1.d \wedge G_2.d)$	$G.d \leftarrow (G_1.d \wedge G_2.d)$	$G.d \leftarrow (G_1.d \wedge G_2.c) \vee (G_1.d \wedge G_2.d)$

Fig. 3. Formulas for 2-colorability of a 2-terminal graph

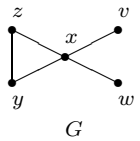
represented by the tree's *root*. Crucial is determining how these subgraph outcomes are merged vis-a-vis the composition rules. We demonstrate the basic concept on the graph in Figure 2 using an easy example from graph coloring.

2.1.1 *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*. Now, for the graph G shown in Figure 2, suppose we seek to decide whether or not it is 2-colorable. Of course, 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 (note that deciding 3-colorability is hard (c.f. [Karp 1972])). That said, we employ this simple case for ease of exposition only; extending the notion for deciding k-colorability with $k \geq 3$ should be natural.

Denote the two colors by c_1 and c_2. Since all constituent subgraphs are 2-terminal graphs, there are only four possible color states for any solution relative to the terminal vertices of any such subgraph; these are given by the pairs $\{c_1, c_2\} \times \{c_1, c_2\}$. The equivalence classes for each subgraph G are given by a 4-tuple $(G.a, G.b, G.c, G.d)$ where each component represents the equivalence class corresponding to the pairs $(c_1, c_1), (c_1, c_2), (c_2, c_1)$, and (c_2, c_2) respectively. Thus, $G.d$ represents the equivalence class of all valid 2-colorings of G that assign color c_2 to both terminals of G.

Since k-coloring is a question of admissability, the primary piece of information required for each $G.x$ is whether the equivalence class is empty or nonempty, that is, whether G admits a valid coloring of type x. Assign values of *true* for admissible and *false* for inadmissible. For base graphs (edges) this 4-tuple is trivially fixed as $(false, true, true, false)$. Now, if two subgraphs are merged under any of the three operations, $\odot_s, \odot_p$, or $\odot_j$, the only admissible outcomes result from pairs where the fused or common vertices are required to have the same color. This leads to the explicit formulation shown in Figure 3. Due to symmetry, it is always true that $G.a \cong G.d$ and $G.b \cong G.c$, but we provide the complete formulas for clarity of exposition.

Beginning with the base graph nodes of the decomposition tree from Figure 2 and applying the computation successively, we can produce a correct set of 4-tuples, one corresponding to each composed graph. If any of the values are *true* in the 4-tuple corresponding to the original graph G (represented at the root of the tree), then G is 2-colorable; otherwise it is not. Naturally, the full computation can be preempted earlier if the 4-tuple for any subgraph consists only of values $false$. The complete



G

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

Fig. 4. Deciding 2-colorability of a 2-terminal graph

computation for the graph of Figure 2 is summarized in Figure 4; obviously, the stated graph G is not 2-colorable.

In the example above there is no need for the secondary information, that is, a particular instance of a valid 2-coloring, because no such coloring exists. When the secondary information is carried, some care must be taken to ensure the algorithm's linear runtime. A naive implementation would store an explicit valid 2-coloring, for each subgraph G, and for each $x \in a, b, c, d$ such that $G.x = true$. The resulting runtime on a 2-colorable n-vertex graph would be $\Omega(n \log n)$, and $\theta(n^2)$ in the worst case. Instead, store a particular coloring in $O(1)$ time by using pointers to previously stored colorings of G's constituent subgraphs, so that the entire algorithm runs in $O(n)$ time.

To illustrate how akin are algorithms for feasibility and counting, we state here the changes in formalisms that would be required in order to determine the number of valid 2-colorings: change $G.x$ from boolean to an integer that carries the cardinality of the equivalence class; accordingly, change the base graph 4-tuple to $(0, 1, 1, 0)$; and replace the $\wedge$ and $\vee$ operators by the $\times$ and $+$ operators respectively.

2.1.2 *Example.* We count the number of valid 3-colorings of the graph in Figure 4. Denote the colors as **1,2,3**. There are $|\{\mathbf{1,2,3}\}^2| = 9$ equivalence classes. Colorings in a class are equivalent in 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 $[0, 1, 1, 1, 0, 1, 1, 1, 0]$. We express the composition formulas in the more compact form of *multiplication tables*, as shown in Figure 5. 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.

The entire computation is summarized in Figure 6. The detailed computation at the final step of the composition is given in Figure 7. The smaller case digit in each cell is the contribution of that cell towards the final 9-tuple.

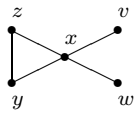
2.2 More about k-terminal graphs

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

$\odot_s$	**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. 5. 3-colorings of a 2-terminal graph



G

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. 6. 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. 7. Detailed computation at final step during 3-coloring

3-terminal class with composition f shown in Figure 8. We will not take space to state an explicit algorithm nor provide a full solution to a problem; however, the partial illustration shown in the next example should suffice to convey the basic scheme in a more general context.

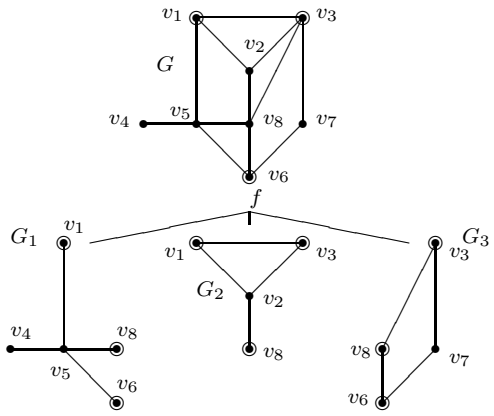


Fig. 8. A 3-terminal graph composition

2.2.1 *Example.* 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 member vertices in S. A minimum vertex cover for G is a smallest S with this property. Now, since a vertex is either in a cover or it is not, and 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 attributes. Denoting the terminal vertices generically by t_1, t_2, and t_3, we can define these attributes as follows:

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

A minimum vertex cover for a composed graph derives from the smallest of these eight values; specifically, the cardinality of a smallest cover is the minimum value in the 8-tuple, while the vertices in the actual cover with this size may be found by a routine backtrack through prior computation.

Now, the explicit computation for the ternary composition shown in Figure 8 can be formulated as shown in Figure 9. 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. Upon applying the specified composition formulas, the respective values for the 8-tuple identified with G can be computed. The computation is summarized in Figure 10. 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.

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

Fig. 9. Formulas for minimum vertex cover in a 3-terminal graph

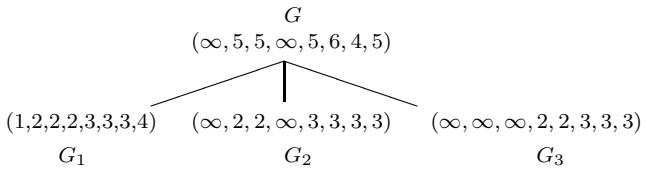


Fig. 10. A 3-terminal computation for vertex cover

2.3 Remarks

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 initiating this tutorial with general notions derived within the broad context of k-terminal graphs, our intention is to create an easy, conceptual backdrop for what now 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.

3. TREES

A *tree* is a connected, acyclic graph. Trees can also be defined as 2-terminal graphs in which the only operation is jackknife. 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 $r = r_1$. In the spirit of k-terminal nomenclature, trees could be regarded as 1-terminal graphs. Figure 11 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, several algorithms for problems on trees are stated. For the first algorithm, we provide some elementary exposition to underscore the basic concepts. For subsequent cases, we will simply state the specific algorithms.

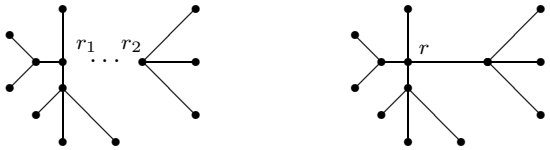


Fig. 11. Recursive construction of a tree

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

Fig. 12. Multiplication table for independent set in a tree

3.1 Algorithm for maximum cardinality independent set in a tree

For some $G = (V, E)$, a subset $S \subseteq V$ is an *independent set* (or *stable set*) if no two vertices in S are adjacent in G. Now, given any tree (or subtree), with a designated root vertex which we now denote by *root* $[G]$, it must be that any admissible independent set of vertices either includes *root* $[G]$ or does not. Whether or not independent sets of two trees G_1, G_2 can be combined into an independent set of the tree $G_1 \oplus G_2$ depends *only* on these inclusions. Hence, let the equivalence classes be:

$G.a$ = max cardinality independent set that includes *root* $[G]$
$G.b$ = max cardinality independent set that excludes *root* $[G]$
$G.c$ = max cardinality independent set

It is convenient to describe the computation by a multiplication table as shown in Figure 12, i.e., all possible outcomes from the composition $G_1 \oplus G_2$ are thus enumerable. Observe that in the table, we have suppressed notation slightly, *i.e.,* rather than listing $G.a$ and $G.b$ we simply specify **a** and **b**. The row by column product assumes the convention where subgraphs (i.e., 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, i.e., 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 $G.a$ while $G.b$ may be produced from a pair of possible products; the maximum of these yields the desired $G.b$. 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 13.

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

Fig. 13. Algorithm for maximum cardinality independent set in a tree

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

Fig. 14. Algorithm for maximum weighted independent set in a tree

3.2 Algorithm for maximum weighted independent set in a tree

Following the approach used to find the maximum cardinality independent set, switching to a weighted version of the independent set problem is straightforward. Let:

$G.d$ = max weight independent set that includes *root* $[G]$
$G.e$ = max weight independent set that excludes *root* $[G]$
$G.f$ = max weight independent set

The algorithm is shown in Figure 14. The stated expression for $G.e$ results from this simplification:

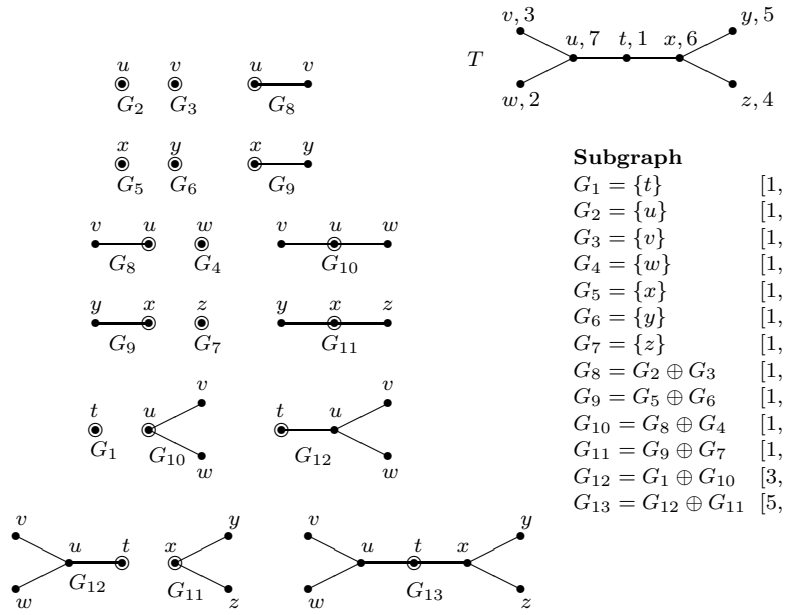
$$\max\{G_1.e + G_2.d, G_1.e + G_2.e\} = G_1.e + \max\{G_2.d, G_2.e\} = G_1.e + G_2.f.$$

3.2.1 *Example.* Consider the tree denoted by T in Figure 15. Vertices are labelled $t, u, \ldots, z$ and next to each label is a vertex weight. The algorithms of Figures 13 and 14 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 $[G.a, G.b, G.c, G.d, G.e, G.f]$. The maximum cardinality and maximum weight of any independent set is $G.c = 5$ and $G.f = 16$ respectively; these are read from the computation for G_{13}. Standard backtracking can be applied to determine that the explicit solutions are sets $\{t, v, w, y, z\}$ and $\{u, y, z\}$ respectively.

3.3 Algorithm for minimum cardinality maximum weight independent set in a tree

Among all maximum weight independent sets, suppose we wish to determine the minimum possible cardinality. Define:

$G.g$ = min cardinality set that yields $G.d$



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, 7]
$G_9 = G_5 \oplus G_6$	[1, 1, 1, 6, 5, 6]
$G_{10} = G_8 \oplus G_4$	[1, 2, 2, 7, 5, 7]
$G_{11} = G_9 \oplus G_7$	[1, 2, 2, 6, 9, 9]
$G_{12} = G_1 \oplus G_{10}$	[3, 2, 3, 6, 7, 7]
$G_{13} = G_{12} \oplus G_{11}$	[5, 4, 5, 15, 16, 16]

Fig. 15. Maximum cardinality and maximum weight independent sets in a tree

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

Fig. 16. Algorithm for minimum cardinality maximum weight independent set in a tree

$G.h$ = min cardinality set that yields $G.e$
$G.i$ = min cardinality max weight independent set

The algorithm is shown in Figure 16.

3.4 Algorithm for bottleneck independent set in a tree

Consider a bottleneck version of the previous weighted independent set problem. That is, among all maximum weight independent sets, find the minimum possible maximum weight of any vertex in such a set. 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 max weight independent set

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

Fig. 17. Algorithm for bottleneck independent set in a tree

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

Fig. 18. Counting the maximum weighted independent sets in a tree

The algorithm is shown in Figure 17.

3.5 Algorithm for counting the maximum weighted independent sets in a tree

Next we consider a counting version of the previous weighted independent set 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 independent sets with maximum weight

The algorithm is shown in Figure 18.

3.6 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 independent set, solutions in each subgraph to be composed need not be admissible, i.e., 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

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

Fig. 19. Multiplication table for dominating set in a tree

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. 20. Algorithm for minimum cardinality dominating set in a tree

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

Fig. 21. Multiplication table for longest path in a tree

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 19. Specifically then we have the algorithm shown in Figure 20.

3.7 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

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. That is, $\emptyset$ signifies a path with no edges (hence, always has value 0). A multiplication table is shown in Figure 21. Then, we can write the algorithm as shown in Figure 22.

3.8 Algorithm for maximum weight matching in a tree

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. We have:

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. 22. Algorithm for longest cardinality path in a tree

$\oplus$	**A**	**B**
A	A	A
B	B	A,B

Fig. 23. Multiplication table for maximum weight matching in a tree

if $|V|$=1 then
  $G.A \leftarrow -\infty$
  $G.B \leftarrow 0$
else if $G = G_1 \oplus G_2$ then
  $G.A \leftarrow \max\{G_1.A + G_2.C, G_1.B + G_2.B +$ weight(*root* $[G_1]$, *root* $[G_2]$)$\}$
  $G.B \leftarrow G_1.B + G_2.C$
$G.C \leftarrow \max\{G.A, G.B\}$

Fig. 24. Algorithm for maximum weight matching in a tree

$G.A$ = max weight matching with *root* $[G]$ saturated
$G.B$ = max weight matching with *root* $[G]$ free
$G.C$ = max weight matching

The associated table is shown in Figure 23. Observe that one location has a multiple entry because the outcome depends upon whether or not the edge (*root* $[G_1]$, *root* $[G_2]$) is placed in the matching. The corresponding algorithm is shown in Figure 24.

3.9 Algorithm for minimum cardinality maximal matching in a tree

A matching $M \subseteq E$ is maximal if there exists no proper superset $N \supset M$ that is also a matching. We now have:

$G.J$ = min maximal matching with *root* $[G]$ saturated
$G.K$ = min maximal matching with *root* $[G]$ free
$G.L$ = min almost-maximal matching where *root* $[G]$ needs an edge
$G.M$ = min maximal matching

The associated table is shown in Figure 25. Again, some locations have multiple entries depending upon whether or not the edge (*root* $[G_1]$, *root* $[G_2]$) is placed in the matching.

$\oplus$	**J**	**K**	**L**
J	J	J	
K	K	J,L	J
L	L	J,L	J

Fig. 25. Multiplication table for min-max matching in a tree

if $|V|$=1 then
  $G.J \leftarrow \infty$
  $G.K \leftarrow 0$
  $G.L \leftarrow \infty$
else if $G = G_1 \oplus G_2$ then
  $G.J \leftarrow \min\{G_1.J + G_2.M, G_1.N + G_2.N + 1\}$
  $G.K \leftarrow G_1.K + G_2.J$
  $G.L \leftarrow \min\{G_1.K + G_2.K, G_1.L + G_2.M\}$
$G.M \leftarrow \min\{G.J, G.K\}$
$G.N \leftarrow \min\{G.K, G.L\}$

Fig. 26. Algorithm for min-max matching in a tree

Next we add an extra parameter $G.N$ merely to express the formulas more concisely:

$G.N$ = min almost-maximal matching or maximal matching with *root* $[G]$ free

The algorithm is shown in Figure 26.

3.10 Remarks

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 example of dependence on the problem, consider the following generalization of the 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 many other problems could well have been selected to represent the basic recursive method on trees. Of course, many problems are trivial on trees (e.g., maximum clique, chromatic number, etc.) and even more interesting problems such as matching, vertex covering, 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 resistent. For example, the minimum bandwidth problem remains hard on trees [Garey et al. 1978].

4. SERIES-PARALLEL GRAPHS

A graph is *series-parallel* if it has no subgraph homeomorphic to K_4 [Duffin 1965]. Series-parallel graphs are also members of a 2-terminal class defined by base graphs that are single edges and binary composition operators that were described earlier in Section 2. 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 (c.f., [Valdes et al. 1982]).

The algorithms stated below will deal only with cardinality examples; weighted or counting versions are straightforward and follow as previously illustrated by the algorithms of Figures 14 and 18 for trees. When designing algorithms for trees, only a single point of composition involving constituent subtrees was relevant, but 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.

4.1 Algorithm for minimum cardinality vertex cover in a series-parallel graph

The vertex cover problem was defined earlier. From the 2-terminal 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 independent set problem 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 27. The explicit algorithm is shown in Figure 28.

$\odot_s$	**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. 27. 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
  $G.a \leftarrow G_1.a + \min\{G_2.a, G_2.b\} - 1$
  $G.b \leftarrow G_1.b + \min\{G_2.c, G_2.d\}$
  $G.c \leftarrow G_1.c + \min\{G_2.a, G_2.b\} - 1$
  $G.d \leftarrow G_1.d + \min\{G_2.c, G_2.d\}$
$G.e \leftarrow \min\{G.a, G.b, G.c, G.d\}$

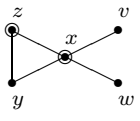
Fig. 28. Algorithm for minimum cardinality vertex cover in a series-parallel graph

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

4.1.1 *Example.* The algorithm of Figure 28 is demonstrated using the series-parallel graph G shown in Figure 29. Vertices are labelled in G as indicated, and to the right the explicit computation is summarized according to the decomposition tree previously shown in Figure 2. The 4-tuples exhibit values for $G.a, G.b, G.c$, and $G.d$. 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 $G.c = 4$ 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 $G.d = \infty$.

4.2 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



G

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. 29. Minimum cardinality vertex cover in a series-parallel graph

$\odot_s$	**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. 30. Multiplication tables for dominating set in a series-parallel graph

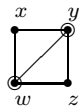
corresponding multiplication tables are shown in Figure 30.

$G.i$ = 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 $G.r \leftarrow \min\{G.i, G.j, G.k, G.l\}$.



G

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. 31. Minimum cardinality dominating set in a series-parallel graph

$\odot_s$	**s**	**t**	**u**	**v**
s			s	t
t	s	t	s	t
u			u	v
v	u	v	u	v

$\odot_p$	**s**	**t**	**u**	**v**
s				s
t			s	t
u		s		u
v	s	t	u	v

$\odot_j$	**s**	**t**	**u**	**v**
s			s	s
t	s	s	t	t
u			u	u
v	u	u	v	v

Fig. 32. Multiplication tables for maximum weight matching in a series-parallel graph

4.2.1 *Example.* The algorithm implied by the tables in Figure 30 is applied to the graph G shown in Figure 31. As before, the constituent subgraphs are indicated where $G_9 = G$; the computation is summarized by the 9-tuples on the extreme right. 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.

4.3 Algorithm for maximum cardinality matching in a series-parallel graph

The weighted version of this problem was previously solved on trees in Figure 24. For series-parallel graphs we have the following classes:

$G.s$ = max cardinality matching with both *left* $[G]$ and *right* $[G]$ saturated
$G.t$ = max cardinality matching with *left* $[G]$ saturated and *right* $[G]$ free
$G.u$ = max cardinality matching with *right* $[G]$ saturated and *left* $[G]$ free
$G.v$ = max cardinality matching with both *left* $[G]$ and *right* $[G]$ free
$G.w$ = max cardinality matching

The corresponding multiplication tables are shown in Figure 32. Initialization when $|E| = 1$ requires that $[G.s, G.t, G.u, G.v] \leftarrow [1, -\infty, -\infty, 0]$. Upon completing the computation, a maximum cardinality matching is obtained by $G.w \leftarrow \max\{G.s, G.t, G.u, G.v\}$.

4.4 Algorithm for minimum cardinality maximal matching in a series-parallel graph

This problem was previously solved on trees in Figure 26. For series-parallel graphs we have the following classes:

$G.A$ = min maximal matching with both *left* $[G]$ and *right* $[G]$ saturated
$G.B$ = min maximal matching with *left* $[G]$ saturated and *right* $[G]$ free

$\odot_s$	**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. 33. Multiplication tables for min-max matching in a series-parallel graph

$G.C$ = min maximal matching with *right* $[G]$ saturated and *left* $[G]$ free
$G.D$ = 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 33. The construction of the jackknife table is left to the reader. Initialization when $|E| = 1$ requires that

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

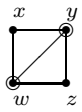
Upon completing the computation, a minimum cardinality maximal matching is obtained by $G.L \leftarrow \min\{G.A, G.B, G.C, G.D\}$.

4.4.1 *Example.* In Figure 34 we reuse the same graph from the previous example to demonstrate the application of the algorithm implied by the tables of Figure 33. The final 10-tuple shown in Figure 34 indicates that a smallest maximal matching consists of a single edge, i.e., $\{w, y\}$.

4.5 Algorithm for Hamiltonian cycle and Hamiltonian path in a series-parallel graph

A *Hamiltonian cycle* is a cycle that visits every vertex exactly once, and a *Hamiltonian path* is a path that visits every vertex exactly once. Define the 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

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. 34. 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. 35. Multiplication tables for Hamiltonian cycle and path in a series-parallel graph

$G.V$ = 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 35. Again, the construction of the jackknife table is left to the reader. Initialization when $|E| = 1$ requires that

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

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 non-biconnected graph, and no such graph can have a Hamiltonian cycle. Accordingly, if the aim is deciding the existence of a Hamiltonian cycle, 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.

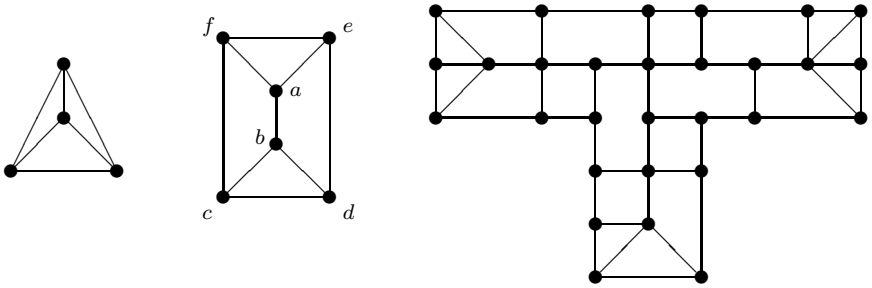


Fig. 36. Some Halin graphs

4.6 Remarks

There is a rich literature pertaining to the solution of problems on series-parallel graphs (c.f. [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 (c.f., [Syslo 1983]) which implies that solving the *travelling 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 is *tree subgraph isomorphism* [Rardin and Parker 1986].

5. 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 including only and all leaves (i.e., degree-1 vertices) of T. A selection of Halin graphs is shown in Figure 36. 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 important, we can easily show constructively that these graphs can always be decomposed as 3-terminal graphs. Consider the middle Halin graph in Figure 36 and fix an embedding of the graph as shown in Figure 37. The terminals are circled and labelled by t_1, t_2, and t_3 as indicated. It is important that t_1 is a non-leaf 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

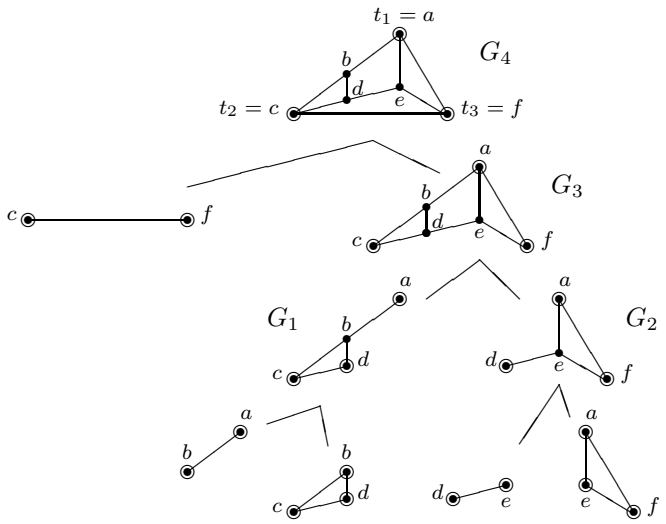


Fig. 37. Halin graph decomposition

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.

Each Halin graph $G = (V, E)$ is a 3-terminal graph with terminals $X \subseteq V$ where $|X| \leq 3$.

5.1 Algorithm for maximum cardinality independent set in a Halin graph

Define:

$G[S]$ = max cardinality independent set that contains $S \subseteq X$ but not $X - S$
$G.max$ = max cardinality independent set

The algorithm is shown in Figure 38. It is illustrated in Figure 39 for the non-base graphs of the Halin graph decomposition shown in Figure 37. The maximum independent sets have size 2; they are $\{b, e\}$, $\{a, d\}$, $\{c, e\}$, $\{b, f\}$, $\{d, f\}$, and $\{a, c\}$.

5.2 Algorithm for minimum cardinality dominating set in a Halin 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 40. Note that $N_G(S)$ denotes the set of neighbors of vertices S in graph G.

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

Fig. 38. Algorithm for maximum cardinality independent set in a Halin graph

$G_1[\emptyset] = 1$	$G_2[\emptyset] = 1$	$G_3[\emptyset] = 2$	$G_4[\emptyset] = 2$
$G_1[\{a\}] = 1$	$G_2[\{a\}] = 1$	$G_3[\{a\}] = 2$	$G_4[\{a\}] = 2$
$G_1[\{c\}] = 1$	$G_2[\{d\}] = 1$	$G_3[\{c\}] = 2$	$G_4[\{c\}] = 2$
$G_1[\{d\}] = 1$	$G_2[\{f\}] = 1$	$G_3[\{f\}] = 2$	$G_4[\{f\}] = 2$
$G_1[\{a,c\}] = 2$	$G_2[\{a,d\}] = 2$	$G_3[\{a,c\}] = 2$	$G_4[\{a,c\}] = 2$
$G_1[\{a,d\}] = 2$	$G_2[\{a,f\}] = -\infty$	$G_3[\{a,f\}] = -\infty$	$G_4[\{a,f\}] = -\infty$
$G_1[\{c,d\}] = -\infty$	$G_2[\{d,f\}] = 2$	$G_3[\{c,f\}] = 2$	$G_4[\{c,f\}] = -\infty$
$G_1[\{a,c,d\}] = -\infty$	$G_2[\{a,d,f\}] = -\infty$	$G_3[\{a,c,f\}] = -\infty$	$G_4[\{a,c,f\}] = -\infty$

Fig. 39. Maximum cardinality independent set in a Halin graph

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

Fig. 40. Algorithm for minimum cardinality dominating set in a Halin graph

5.3 Remarks

There are also a number of interesting properties 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 partial 3-trees where the latter recursive class is defined in the next section. But it is also the 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].

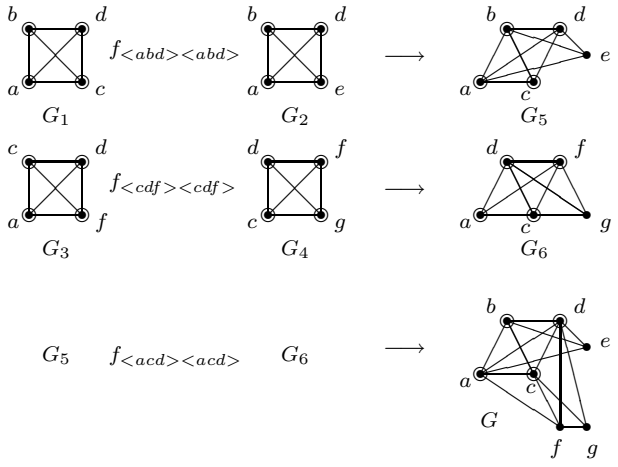


Fig. 41. Construction of a 4-jackknife graph

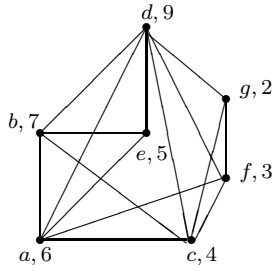
6. K-JACKKNIFE GRAPHS, K-TREES AND PARTIAL K-TREES

The *k-jackknife class* of k-terminal graphs has been described previously in various sources (c.f. [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 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 41.

We could legitimately have incorporated coverage of the k-jackknife class in Section 2. However, it is purposely reserved for this section since it plays an instructive, connecting role with other, more popularly identifiable graph classes, namely k-trees and partial k-trees the definitions of which follow.

First, K_k is a k-tree. Then a k-tree with $n+1$ vertices ($n \geq k$) is constructed from a k-tree on n 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 42; 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.

Note that the 3-tree in Figure 42 is isomorphic to the 4-jackknife graph G formed in Figure 41. In fact, this outcome is not coincidental, since there exists a general relationship between k-jackknife graphs and k-trees/partial k-trees. Specifically, a graph (V,E) is a partial k-tree if and only if there is a subset $T \subseteq V$ such that



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. 42. A 3-tree

initialize $S \leftarrow \emptyset$
for each vertex w in reverse order of how added when the k-tree was built do
    if w is not adjacent to any vertex belonging to S then
        $S \leftarrow S \cup \{w\}$
return $|S|$

Fig. 43. Greedy algorithm for maximum cardinality independent set in a k-tree

(V, T, E) is a $(k+1)$-jackknife graph. Furthermore, a 2-terminal graph (V, W, E) is series-parallel if and only if there is a superset $T \supseteq W$ for which (V, T, E) is a 3-jackknife graph, so the series-parallel graphs and the partial 2-trees form equivalent classes.

6.1 Algorithm for maximum cardinality independent set in a k-tree

We first consider only the solution of the independent set problem on k-trees; we treat cardinality and weighted versions separately. That we do so follows since the cardinality case is solved by a simple greedy procedure while the weighted case requires the dynamic programming approach. The greedy algorithm for the cardinality case is shown in Figure 43.

The 3-tree shown in Figure 42 has several maximum cardinality independent sets of size 2, but the algorithm of Figure 43 constructs only the set $S = \{g, e\}$. To see this, first consider vertex g and add it to set S. Next, we consider vertex f for inclusion but the choice is rejected since f is adjacent to g. Continuing, vertex e can be added to S but this is the last admissible inclusion since d, c, b, and a are adjacent to g and/or e.

6.2 Algorithm for maximum weighted independent set in a k-tree

Now, in stating and demonstrating the dynamic programming algorithm for the weighted case, we shall refer to the $(k+1)$-jackknife representation of a k-tree, which always exists as indicated earlier. Let X denote the set of terminals, where

if $X = V$ then
  $G[\emptyset] \leftarrow 0$
  $\forall_{t \in X}\ G[t] \leftarrow \text{weight}(t)$
else if $(G,\ X)$ is composed from $(G_1,\ X)$ and $(G_2,\ X')$ then
  $G[\emptyset] \leftarrow G_1[\emptyset] + \max\ \{G_2[\emptyset]\} \cup \{G_2[u] : u \in X' - X\}$
  $\forall_{t \in X - X'}\ G[t] \leftarrow G_1[t] + \max\ \{G_2[\emptyset]\} \cup \{G_2[u] : u \in X' - X\}$
  $\forall_{t \in X \cap X'}\ G[t] \leftarrow G_1[t] + G_2[t] - \text{weight}(t)$
$G.max \leftarrow \max\ \{G[\emptyset]\} \cup \{G[t] : t \in X\}$

Fig. 44. Algorithm for maximum weighted independent set in a k-tree

$G_1[\emptyset] = 0$	$G_2[\emptyset] = 0$	$G_3[\emptyset] = 0$	$G_4[\emptyset] = 0$	$G_5[\emptyset] = 5$	$G_6[\emptyset] = 2$	$G[\emptyset] = 8$
$G_1[a] = 6$	$G_2[a] = 6$	$G_3[a] = 6$	$G_4[c] = 4$	$G_5[a] = 6$	$G_6[a] = 8$	$G[a] = 8$
$G_1[b] = 7$	$G_2[b] = 7$	$G_3[c] = 4$	$G_4[d] = 9$	$G_5[b] = 7$	$G_6[c] = 4$	$G[b] = 10$
$G_1[c] = 4$	$G_2[d] = 9$	$G_3[d] = 9$	$G_4[f] = 3$	$G_5[c] = 9$	$G_6[d] = 9$	$G[c] = 9$
$G_1[d] = 9$	$G_2[e] = 5$	$G_3[f] = 3$	$G_4[g] = 2$	$G_5[d] = 9$	$G_6[f] = 3$	$G[d] = 9$

Fig. 45. Maximum weight independent set in a 3-tree

$|X| = k + 1$. Because the graph is a k-tree, the vertices in X form a clique, and so at most one terminal can belong to any independent set. Define:

$G[\emptyset]$ = max weight independent set that contains no terminal vertex
$G[t]$ = max weight independent set that contains the terminal t
$G.max$ = max weight independent set

The algorithm is shown in Figure 44. Trivially, this algorithm is also applicable to the cardinality case by setting all vertex weights to 1.

6.2.1 *Example.* Figure 45 illustrates the algorithm of Figure 44 for the 3-tree shown in Figure 42, and whose 4-jackknife decomposition is given in Figure 41. Weights are indicated next to each vertex label in Figure 42. The maximum weight independent set is $\{b, f\}$ which has total weight 10.

6.3 Remarks

Arguably, the most celebrated class among the three identified in this section is the class of partial k-trees. There is a reason why we chose to demonstrate with algorithms using the other classes however. Importantly, partial k-trees can be recognized and their decomposition trees produced in polynomial time (for fixed k). This attribute follows since partial k-trees are equivalent to the class of treewidth-k graphs which are discussed in a following section. Indeed, an algorithm for a problem on treewidth-k graphs is applicable for the associated class of partial k-trees. Moreover, the corresponding polynomial recognition/decomposition tree result, for arbitrary k, is a generalized outcome requiring $O(n^{k+2})$ effort. That said, when $k \leq 4$, linear-time recognition procedures are known ([Matousek and Thomas 1991] for $k = 3$; [Sanders 1996] when $k = 4$). Note finally that if k is not fixed but rather, is allowed to be part of the problem instance, then recognition of partial k-trees (and the aforementioned equivalent classes) is $\mathcal{NP}$-complete.

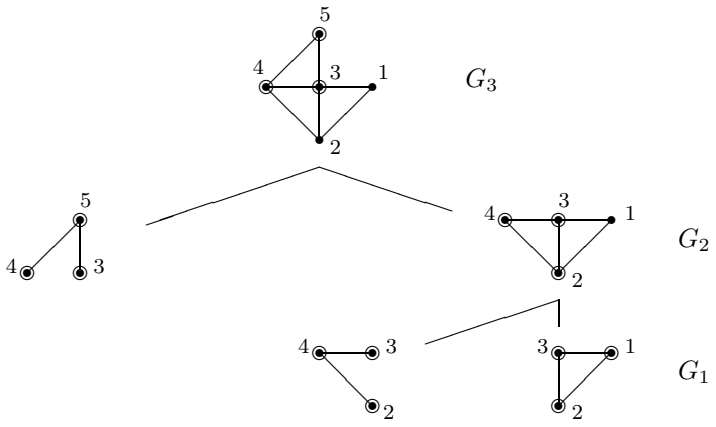


Fig. 46. A bandwidth-2 graph

7. BANDWIDTH-K GRAPHS

A graph $G = (V, E)$ is a *bandwidth-k graph* if there exists a labelling of its vertices $h : V \to \{1, 2, \ldots |V|\}$ such that $\{u, v\} \in E \Rightarrow |h(u) - h(v)| \leq k$. A bandwidth-2 graph is shown in Figure 46. 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 $(V_1 \cup V_2, T_1, E_1 \cup E_2)$.

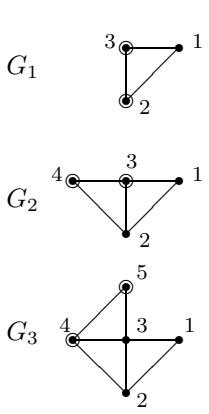
7.1 Algorithms

Algorithms for problems on bandwidth-k graphs typically first solve the problem on the subgraph induced by the first $k+1$ vertices, and then add one vertex at a time, maintaining just enough information to solve the larger problem at each step. For example, consider the independent set problem on a graph $G = (V, E)$. For subsets $X, Y \subseteq V$ define $G[X, Y]$ to be the largest size of any independent set of G that includes all the vertices of X but no vertices from Y. The key to computational efficiency is that for bandwidth-k graphs, the algorithm only needs to consider cases where $X \cup Y$ consists of the k highest numbered vertices, since these are the only vertices that can interact with the remainder of the graph.

7.1.1 *Example.* Figure 47 illustrates the independent set algorithm computations on the bandwidth-2 graph from Figure 46. The optimum solution has size 2, given by any of the sets $\{4, 1\}$, $\{5, 2\}$ or $\{5, 1\}$.

7.2 Remarks

Repeating the theme that was articulated in the case of partial k-trees, bandwidth-k graphs are related to important classes that will be taken up shortly, *i.e.* pathwidth-k and treewidth-k graphs. Hence, relevant attributes of the latter, whether related to recognizability or to problem-solving, are similarly applicable to the class of



$G_1[\emptyset, \{2,3\}] = 1$
$G_1[\{2\}, \{3\}] = 1$
$G_1[\{3\}, \{2\}] = 1$
$G_1[\{2,3\}, \emptyset] = -\infty$

$G_2[\emptyset, \{3,4\}] = 1$
$G_2[\{3\}, \{4\}] = 1$
$G_2[\{4\}, \{3\}] = 2$
$G_2[\{3,4\}, \emptyset] = -\infty$

$G_3[\emptyset, \{4,5\}] = 1$
$G_3[\{4\}, \{5\}] = 2$
$G_3[\{5\}, \{4\}] = 2$
$G_3[\{4,5\}, \emptyset] = -\infty$

Fig. 47. Maximum cardinality independent set in a bandwidth-2 graph

bandwidth-k structures; indeed, every problem that can be solved on pathwidth-k graphs or treewidth-k graphs can also be solved on bandwidth-k graphs.

8. PATHWIDTH-K GRAPHS

The graph classes in this and the following two sections have their origins in the seminal works of Robertson and Seymour ([Robertson and Seymour 1983],[Robertson and Seymour 1986a]). Most notable among these is the class of so-called treewidth-k graphs which is taken up in Section 9. Here, we begin with a special case known as pathwidth-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 48 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 48 has pathwidth 3.

8.1 Algorithms

Algorithms for problems on pathwidth-k graphs begin by first solving the problem on the subgraph induced by X_1. Next they advance to $X_1 \cup X_2$, then to $X_1 \cup X_2 \cup X_3$, and so on. For the subgraph induced by $X_1 \cup \ldots \cup X_j$, only the vertices in X_j can be adjacent to vertices outside this subgraph, and there are at most $k + 1$ such vertices.

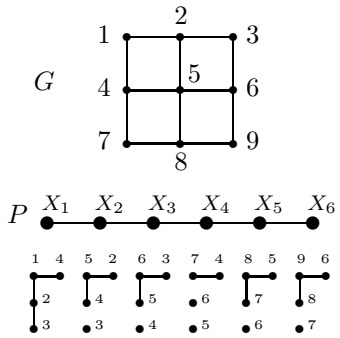


Fig. 48. A path-decomposition

$G_1.\emptyset = 0$	$G_2.\emptyset = 1$	$G_3.\emptyset = 1$	$G_4.\emptyset = 2$	$G_5.\emptyset = 2$	$G_6.\emptyset = 3$
$G_1.\{1\} = 1$	$G_2.\{2\} = 1$	$G_3.\{3\} = 2$	$G_4.\{4\} = 2$	$G_5.\{5\} = 3$	$G_6.\{6\} = 3$
$G_1.\{2\} = 1$	$G_2.\{3\} = 2$	$G_3.\{4\} = 2$	$G_4.\{5\} = 3$	$G_5.\{6\} = 3$	$G_6.\{7\} = 4$
$G_1.\{3\} = 1$	$G_2.\{4\} = 1$	$G_3.\{5\} = 2$	$G_4.\{6\} = 2$	$G_5.\{7\} = 3$	$G_6.\{8\} = 3$
$G_1.\{4\} = 1$	$G_2.\{5\} = 2$	$G_3.\{6\} = 2$	$G_4.\{7\} = 3$	$G_5.\{8\} = 3$	$G_6.\{9\} = 4$
$G_1.\{1,2\} = -\infty$	$G_2.\{2,3\} = -\infty$	$G_3.\{3,4\} = 2$	$G_4.\{4,5\} = -\infty$	$G_5.\{5,6\} = -\infty$	$G_6.\{6,7\} = 3$
$G_1.\{1,3\} = 2$	$G_2.\{2,4\} = 2$	$G_3.\{3,5\} = 3$	$G_4.\{4,6\} = 3$	$G_5.\{5,7\} = 4$	$G_6.\{6,8\} = 4$
$G_1.\{1,4\} = -\infty$	$G_2.\{2,5\} = -\infty$	$G_3.\{3,6\} = -\infty$	$G_4.\{4,7\} = -\infty$	$G_5.\{5,8\} = -\infty$	$G_6.\{6,9\} = -\infty$
$G_1.\{2,3\} = -\infty$	$G_2.\{3,4\} = 2$	$G_3.\{4,5\} = -\infty$	$G_4.\{5,6\} = -\infty$	$G_5.\{6,7\} = 3$	$G_6.\{7,8\} = -\infty$
$G_1.\{2,4\} = 2$	$G_2.\{3,5\} = 3$	$G_3.\{4,6\} = 3$	$G_4.\{5,7\} = 4$	$G_5.\{6,8\} = 4$	$G_6.\{7,9\} = 5$
$G_1.\{3,4\} = 2$	$G_2.\{4,5\} = -\infty$	$G_3.\{5,6\} = -\infty$	$G_4.\{6,7\} = 3$	$G_5.\{7,8\} = -\infty$	$G_6.\{8,9\} = -\infty$

Fig. 49. Independent set in a pathwidth-3 graph

8.1.1 *Example.* Define G_j to be the subgraph of G induced by vertices $X_1 \cup \ldots \cup X_j$. For each subset $Z \subseteq X_j$ let $G_j.Z$ denote the size of the largest independent set of G_j that includes all the vertices of Z but none from $X_j - Z$.

Figure 49 shows the independent set algorithm computations for the pathwidth-3 graph from Figure 48. Observe that all subsets of size 3 or 4 yield the value $-\infty$, so these entries are not shown. The optimum solution is an independent set of size 5, exhibited by $\{1, 3, 5, 7, 9\}$.

8.2 Remarks

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].

As indicated above, Section 9 will describe the important notion of a tree-decomposition, which generalizes the concept of a path-decomposition. Specifically, every pathwidth-k graph is also a treewidth-k graph, and hence every problem that

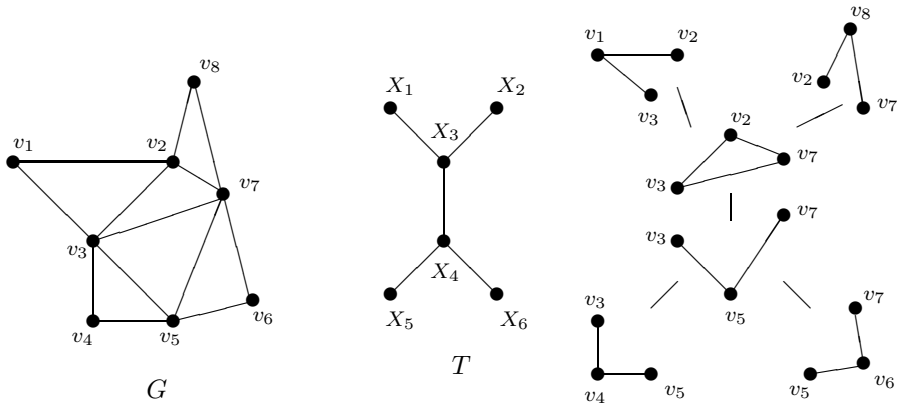


Fig. 50. Example of a tree-decomposition

can be solved on treewidth-k graphs can also be solved on pathwidth-k graphs. Accordingly, the coverage of algorithms on the treewidth-k class is given more extensive coverage as the following section reveals.

9. TREEWIDTH-K GRAPHS

The work on treewidth reported by Robertson and Seymour in ([Robertson and Seymour 1983],[Robertson and Seymour 1986a]) 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 (c.f., [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:

— $\bigcup_{i \in I} X_i = V$;

— 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 $X_i \bigcap X_k \subseteq X_j$.

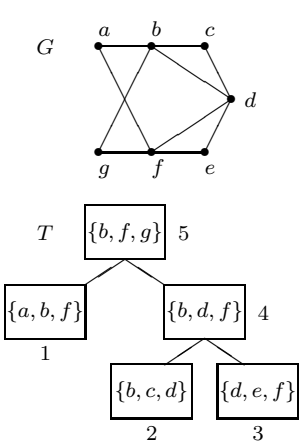
Figure 50 demonstrates with an example.

Trivially, every graph G has a tree-decomposition that is defined by a single vertex, i.e., 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 50 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

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

Fig. 51. Algorithm for maximum cardinality independent set in a treewidth-k graph



Subgraph

$(G_1, X_1 = \{a, b, f\})$
$[\emptyset \to 0, \{a\} \to 1, \{b\} \to 1, \{f\} \to 1, \{b, f\} \to 2]$

$(G_2, X_2 = \{b, c, d\})$
$[\emptyset \to 0, \{b\} \to 1, \{c\} \to 1, \{d\} \to 1]$

$(G_3, X_3 = \{d, e, f\})$
$[\emptyset \to 0, \{d\} \to 1, \{e\} \to 1, \{f\} \to 1]$

$(G_4, X_4 = \{b, d, f\}) = G_2 \otimes G_3$
$[\emptyset \to 2, \{b\} \to 2, \{d\} \to 1, \{f\} \to 2, \{b, f\} \to 2]$

$(G_5, X_5 = \{b, f, g\}) = G_1 \otimes G_4$
$[\emptyset \to 3, \{b\} \to 2, \{f\} \to 2, \{g\} \to 4, \{b, f\} \to 2]$

Fig. 52. Maximum cardinality independent set in a treewidth-2 graph

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 $|X| \leq k + 1$.

9.1 Algorithm for maximum cardinality independent set in a treewidth-k graph

Define:

$G[S]$ = max cardinality independent set that contains $S \subseteq X$ but not $X - S$
$G.max$ = max cardinality independent set

The algorithm is shown in Figure 51. It is demonstrated on the treewidth-2 graph G shown in Figure 52. 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 larger than $-\infty$ within each 8-tuple are shown as the computation progresses. The maximum independent set has size 4, and the explicit solution is $\{a, c, e, g\}$.

9.2 Algorithm for minimum cardinality dominating set in a treewidth-k graph

Define:

if $X = V$ then
    $\forall_{S,T \subseteq X}\ G[S,T] \leftarrow$ if $T = S \cup N_G(S)$ then $|S|$ else ∞
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. 53. Algorithm for minimum cardinality dominating set in a treewidth-k graph

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. 54. Algorithm for chromatic index in a treewidth-k graph

$G[S,T]$ = 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 53. Recall that $N_G(S)$ denotes the set of neighbors of vertices S in graph G.

9.3 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 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 54; it invokes the closure function which is given in Figure 55. Its running time is polynomial because the sum of the entries in each 2^k-tuple is at most $|E|$, and therefore each graph has a polynomial number of valid 2^k-tuples. However, the running time is not linear.

9.4 Monadic second-order logic (MSOL)

Though differing structurally (and in some cases, substantially so), the graph classes described thus far and importantly, the algorithms defined, 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 paper. Moreover, this consistency translates directly to the problem solving realm. Most of the problems solved thus far (e.g., variations of independent set, dominating

```
closure(D′, D″) {
    Z ← D′ × D″ × {(0, ..., 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) ∈ Z;
        choose any S′ ⊆ X1 and S″ ⊆ X2 such that S′ ∩ S″ = ∅ and c′S′ + c″S″ > 0;
        if c′S′ = 0 then S′ ← ∅;
        else if c″S″ = 0 then S″ ← ∅;
        choose any S ⊆ X such that S′ ∩ X = S ∩ X1 and S″ ∩ X = S ∩ X2;
        c′S′ ← max {0, c′S′ − 1};
        c″S″ ← max {0, c″S″ − 1};
        cS ← cS + 1;
        add tuple (C′, C″, C) to Z;
    }
    D ← {C : ((0, ..., 0), (0, ..., 0), C) ∈ Z};
    return D;
}
```

Fig. 55. Closure function for chromatic index algorithm

$$P \rightarrow Q \Leftrightarrow \neg P \vee Q$$
$$P \leftrightarrow Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$$
$$e_i = e_j \Leftrightarrow (\forall v_1)\ (\text{Incident}(v_1, e_i) \leftrightarrow \text{Incident}(v_1, e_j))$$
$$\text{Adjacent}(v_i, v_j) \Leftrightarrow \neg(v_i = v_j) \wedge (\exists e_1)\ (\text{Incident}(v_i, e_1) \wedge \text{Incident}(v_j, e_1))$$

Fig. 56. Some useful MSOL predicates

set, clique, m-vertex coloring (for arbitrary, fixed m), matching, and Hamiltonian cycle/path problems) and many others can be solved by linear-time algorithms on recursive classes. This, 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 a MSOL predicate. Finally, if P is a 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 56. Of particular interest is the creation of legal expressions that formalize the full expression of important graph problems, such as those listed in Figure 57. This is a very short list and interested readers are directed to the literature for a more expansive compilation (c.f., [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.

$$\begin{aligned}
&\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{VertexColorable}_m(V_1, \ldots, V_m) \Leftrightarrow (\forall v_0)\ (v_0 \in V_1 \vee \ldots \vee v_0 \in V_m)\\
&\qquad\qquad \wedge\text{IndependentSet}(V_1) \wedge \ldots \wedge \text{IndependentSet}(V_m)\\
&\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. 57. Graph problems expressed in MSOL

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 virtually all the classes covered in Sections 2 through 10 of this paper, 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.

9.5 Examples

Asking if a given vertex subset V_1 forms an admissible vertex cover in a graph $G = (V, E)$ is easy to state in the predicate calculus. Denoting this by VertexCover(V_1), we can write:

$$\text{VertexCover}(V_1) \Leftrightarrow (\forall e_1)(\exists v_1 \in V_1)(\text{Incident}(v_1, e_1))$$

For a more interesting illustration, let us consider the problem of deciding if a graph $G = (V, E)$ is m-vertex colorable. First, we examine the primitives of $\cup$ and $\cap$ as follows:

$$V_3 = V_1 \cup V_2 \Leftrightarrow (\forall v_1)((v_1 \in V_1 \vee v_1 \in V_2) \leftrightarrow v_1 \in V_3)$$

and

$$V_3 = V_1 \cap V_2 \Leftrightarrow (\forall v_1)((v_1 \in V_1 \wedge v_1 \in V_2) \leftrightarrow v_1 \in V_3).$$

Now, letting $\text{Part}(V, V_1, \ldots, V_m)$ denote an m-partition of the vertex set V, we have

$$\text{Part}(V, V_1, \ldots, V_m) \Leftrightarrow (V_1 \cup \ldots \cup V_m = V) \wedge (V_i \cap V_j = \emptyset)(1 \leq i < j \leq m).$$

Finally,

$$\text{Color}(V_1, \ldots, V_m) \Leftrightarrow \text{Part}(V, V_1, \ldots, V_m) \wedge \text{IndependentSet}(V_i)(1 \leq i \leq m),$$

which establishes that deciding m-colorability on treewidth-k graphs (and the implied, related graph classes) is linear-time solvable.

9.6 Remarks

Once a problem is expressed in MSOL, a linear-time dynamic programming algorithm can be created mechanically [Borie et al. 1992]. Though interesting, this is an existential outcome and the algorithms generated accordingly do not produce practical procedures directly. That is, 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.

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. Somewhat surprisingly, this result is obtained in part due to the existence of an MSOL expression for stating that a graph has treewidth at most k [Bodlaender 1996].

For some problems, an MSOL expression cannot be written and a linear-time algorithm cannot be found. In these cases it may still be possible to develop a linear-time algorithm via an extension to MSOL [Borie et al. 1992], or 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 [Borie 1995]. This is relevant in the case of chromatic index where Vizing's theorem [Vizing 1964] holds that the chromatic index of any graph is either Δ or Δ+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 54 is bounded as $0 \leq c_S \leq |V|$ for all $S \subseteq X$, and the size of each set Z in the closure function of Figure 55 is $O(V^j)$ where $j = 3{\times}2^k$. Therefore the chromatic index algorithm runs in polynomial time with appropriate data structures.

10. BRANCHWIDTH-K GRAPHS

The concept of branchwidth was introduced in [Robertson and Seymour 1991], [Robertson and Seymour 1995]. 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 non-leaf 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. That is, 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 58 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.

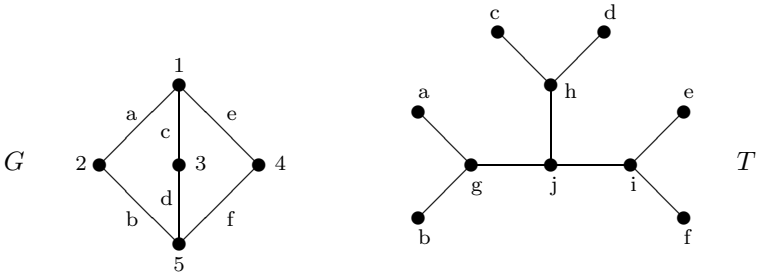


Fig. 58. A branchwidth-2 graph and its branch-decomposition

2-terminal operation	Nodes of subtree	Edges of subgraph	Terminals	4-tuple of values
base graph	$\{a\}$	$\{a\}$	$t_1 = 1, t_2 = 2$	$(0, 1, 1, \infty)$
base graph	$\{b\}$	$\{b\}$	$t_1 = 2, t_2 = 5$	$(0, 1, 1, \infty)$
base graph	$\{c\}$	$\{c\}$	$t_1 = 1, t_2 = 3$	$(0, 1, 1, \infty)$
base graph	$\{d\}$	$\{d\}$	$t_1 = 3, t_2 = 5$	$(0, 1, 1, \infty)$
base graph	$\{e\}$	$\{e\}$	$t_1 = 1, t_2 = 4$	$(0, 1, 1, \infty)$
base graph	$\{f\}$	$\{f\}$	$t_1 = 4, t_2 = 5$	$(0, 1, 1, \infty)$
add node g	$\{a, b, g\}$	$\{a, b\}$	$t_1 = 1, t_2 = 5$	$(1, 1, 1, 2)$
add node h	$\{c, d, h\}$	$\{c, d\}$	$t_1 = 1, t_2 = 5$	$(1, 1, 1, 2)$
add node i	$\{e, f, i\}$	$\{e, f\}$	$t_1 = 1, t_2 = 5$	$(1, 1, 1, 2)$
add node j	$\{a, b, c, d, e, f, g, h, i, j\}$	$\{a, b, c, d, e, f\}$	$t_1 = 1, t_2 = 5$	$(3, 1, 1, 2)$

Fig. 59. Independent set in a branchwidth-2 graph

10.1 Algorithms

Every branchwidth-k graph can be regarded as a k-terminal graph. Specifically, to build a branchwidth-k graph G using k-terminal operations, start from the base graphs which are the individual (2-terminal) edges, and which correspond to the isolated leaves of the tree T. As each new interior node is added to the tree T, along with its (at most 3) incident edges, the corresponding (at most 3) k-terminal graphs are joined using a k-terminal operation. Once all nodes and edges are added to T, the entire graph G is constructed.

10.1.1 *Example.* Figure 59 shows a 2-terminal construction of the branchwidth-2 graph from Figure 58, and traces the maximum cardinality independent set algorithm on this graph. This algorithm is the usual 2-terminal algorithm for maximum independent set. It maintains a 4-tuple of values $(G[\emptyset], G[\{t_1\}], G[\{t_2\}], G[\{t_1, t_2\}])$ for each 2-terminal subgraph that is built. The 4 entries in each tuple correspond to each possible subset of terminals that can belong to the independent set. The maximum independent set of G is $\{2, 3, 4\}$ with size 3.

10.2 Remarks

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-

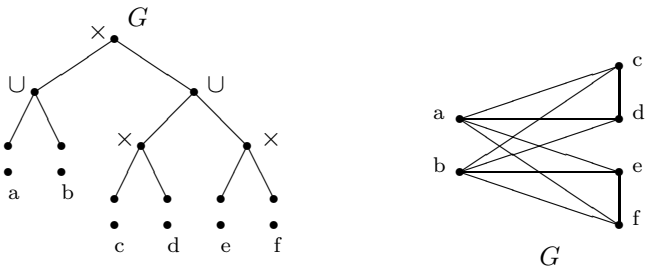


Fig. 60. Cograph construction

```
if |V| = 1 then
    G.i ← 1
else if G = G1 ∪ G2 then
    G.i ← G1.i + G2.i
else if G = G1 × G2 then
    G.i ← max {G1.i, G2.i}
```

Fig. 61. Algorithm for maximum cardinality independent set in a cograph

$(\lfloor \frac{3k}{2} \rfloor - 1)$ graph.

11. COGRAPHS

Properties of *cographs*, also known as *complement reducible graphs*, were largely developed in [Corneil et al. 1981], [Corneil et al. 1984], and [Corneil et al. 1985]. 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 60 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.

11.1 Algorithm for maximum cardinality independent set in a cograph

The algorithm is shown in Figure 61.

11.2 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 62.

11.3 Algorithm for minimum cardinality dominating set in a cograph

The algorithm is shown in Figure 63.

```
if |V| = 1 then
      G.c ← 1
else if G = G1 ∪ G2 then
      G.c ← max {G1.c, G2.c}
else if G = G1 × G2 then
      G.c ← G1.c + G2.c
```

Fig. 62. Algorithm for maximum cardinality clique and chromatic number in a cograph

```
if |V| = 1 then
      G.d ← 1
else if G = G1 ∪ G2 then
      G.d ← G1.d + G2.d
else if G = G1 × G2 then
      G.d ← min {G1.d, G2.d, 2}
```

Fig. 63. Algorithm for minimum cardinality dominating set in a cograph

```
if |V| = 1 then
      G.m ← 0
else if G = G1 ∪ G2 then
      G.m ← G1.m + G2.m
else if G = G1 × G2 then
      G.m ← min {G1.m + |V2|, G2.m + |V1|, ⌊(|V1| + |V2|)/2⌋}
```

Fig. 64. Algorithm for maximum cardinality matching in a cograph

11.4 Algorithm for maximum cardinality matching in a cograph

The algorithm is shown in Figure 64. The last formula in this algorithm is obtained by simplifying the following more straightforward but less efficient expression:

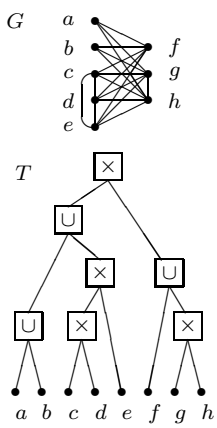
$$\max\{k + \min\{G_1.m, \lfloor\frac{|V_1| - k}{2}\rfloor\} + \min\{G_2.m, \lfloor\frac{|V_2| - k}{2}\rfloor\} : 0 \leq k \leq \min\{|V_1|, |V_2|\}\},$$

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

11.4.1 *Example.* The algorithms of Figures 61 through 64 are demonstrated on the cograph G shown in Figure 65. T denotes the tree decomposition of G, and each 4-tuple exhibits values for $G.i, G.c, G.d$, and $G.m$. The maximum independent set has size 3, for example $\{a, b, c\}$. The maximum clique has size 5, given by $\{c, d, e, g, h\}$. The minimum dominating set has size 2, for example $\{a, f\}$. Lastly, the maximum matching has size 4, for example $\{(a, f), (b, g), (c, h), (d, e)\}$.

11.5 Algorithm for Hamiltonian path and Hamiltonian cycle in a cograph

Define:



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

Fig. 65. Maximum cardinality independent set, clique, dominating set, and matching in a cograph

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. 66. Algorithm for Hamiltonian path and Hamiltonian cycle in a cograph

$G.dp$ = 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 66.

11.6 Remarks

From the graph-theoretic perspective, there are numerous interesting properties of cographs. For example, the complement of any cograph is also a cograph, and all cographs are perfect. Algorithmically, weighted versions of the independent set, clique, and dominating set problems are solvable in linear time on cographs by extending the preceding algorithms in fairly natural ways. However, weighted versions of problems like matching and Hamiltonian path/cycle do not appear to be solvable in linear time on cographs (although they are solvable in polynomial time) because, intuitively, the cross product operation adds too many edges, where each edge potentially has a different weight.

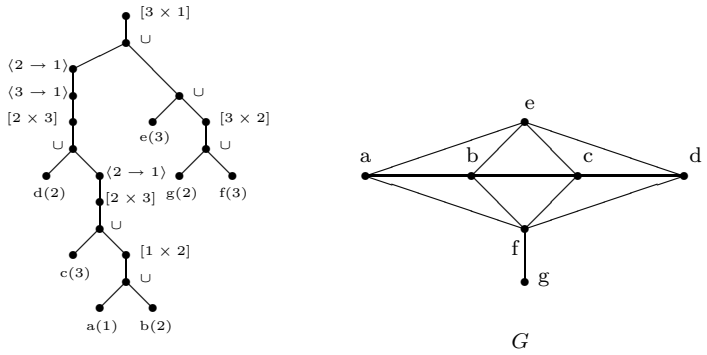


Fig. 67. A cliquewidth-3 graph construction

12. 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 $\{1, 2, \ldots, k\}$. 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 $G_1 \cup G_2$ is a cliquewidth-k graph. Also, the graph $(G_1)_{i \times j}$ 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 $(G_1)_{i \to j}$ 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] $= \{1, \ldots, k\}$ using only the operations $G = G_1 \cup G_2, G = (G_1)_{i \times j}$, and $G = (G_1)_{i \to j}$. The *cliquewidth* of a graph G is the smallest value of k such that G is 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 labelled, single-vertex graph, and each non-leaf node of the tree is obtained by applying one of the operations $\cup$, $i \times j$, or $i \to j$ to its child or children. A cliquewidth-3 decomposition is demonstrated in Figure 67.

12.1 Algorithm for maximum cardinality independent set in a cliquewidth-k graph

Define:

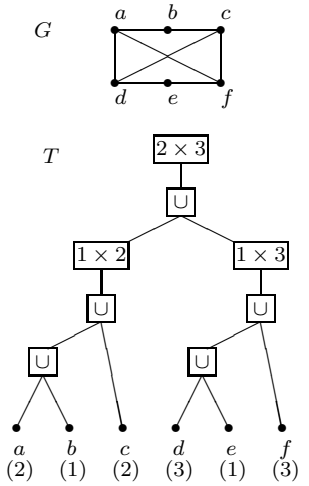
$G[S]$ = max cardinality independent set that contains only labels from $S \subseteq [k]$
$G.max$ = max cardinality independent set

The algorithm is shown in Figure 68.

12.1.1 *Example.* We demonstrate the algorithm of Figure 68 on the cliquewidth-3 graph G shown in Figure 69. 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 maximum independent set has size 3, given by either $\{a, c, e\}$ or $\{b, d, f\}$.

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 $G = G_1 \cup G_2$ then
  $\forall_{S \subseteq [k]}\ G[S] \leftarrow G_1[S] + G_2[S]$
else if $G = (G_1)_{i \times j}$ then
  $\forall_{S \subseteq [k]}\ G[S] \leftarrow \max\ \{G_1[S - \{i\}], G_1[S - \{j\}]\}$
else if $G = (G_1)_{i \rightarrow j}$ then
  $\forall_{S \subseteq [k]}\ G[S] \leftarrow$ if $j \in S$ then $G_1[S \cup \{i\}]$ else $G_1[S - \{i\}]$
$G.max \leftarrow \max\ \{G[S] : S \subseteq [k]\}$

Fig. 68. Algorithm for maximum cardinality independent set in a cliquewidth-k graph



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

Fig. 69. Maximum cardinality independent set in a cliquewidth-3 graph

12.2 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 70.

12.3 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*

if $|V| = 1$ then
  let $v \in V$
  $\forall_{S,T \subseteq [k]}\ G[S,T] \leftarrow$ if S=$\{\text{label}(v)\}$ and $T = [k]$ then 1
    else if S=$\emptyset$ and T=$[k]$−$\{\text{label}(v)\}$ then 0
    else ∞
else if $G = G_1 \cup G_2$ 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 $G = (G_1)_{i \times j}$ 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 $G = (G_1)_{i \rightarrow j}$ 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. 70. Algorithm for minimum cardinality dominating set in a cliquewidth-k graph

(c.f., [Courcelle and Olariu 2000]).

Many problems including variations of independent set, dominating set, clique, and m-vertex colorability (for fixed m) can 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, e_i with domain E, and V_i with domain 2^V. The language contains primitive predicates such as $\mathrm{v}_i = \mathrm{v}_j$, Incident($\mathrm{v}_i$, e_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 the same as MSOL without edge set variables E_i and primitive predicates such as $\mathrm{e}_i \in \mathrm{E}_j$ that refer to edge set variables.

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 or 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 such problems as IndependentSet(V_1), Clique(V_1), DominatingSet(V_1), and VertexColorable$_m$(V_1, ..., V_m) given previously in Figure 57 are also MSOL′ expressions. On the other hand, our MSOL expressions for Matching(E_1), Connected(E_1), HamCycle(E_1), and HamPath(E_1) in Figure 57 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 in the following algorithm.

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 $G = G_1 \cup G_2$ 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 $G = (G_1)_{i \times j}$ then
  $G.Q \leftarrow$ closure$(G_1.Q, i, j)$
else if $G = (G_1)_{i \to j}$ 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 $\{h, l\} \cap \{i, j\} = \emptyset\}$
$G.hp \leftarrow \bigvee \{1 = \Sigma_{1 \le i \le j \le k}\ p_{ij} : P \in G.Q\}$

Fig. 71. Algorithm for Hamiltonian path in a cliquewidth-k graph

```
closure(Q', i, j) {
    Q ← Q';
    while (more tuples can be added to Q) do {
        choose any tuple P' ∈ Q and labels h, l such that p'_hi > 0 and p'_jl > 0;
        P ← P';
        p_hi ← p_hi − 1;
        p_jl ← p_jl − 1;
        p_hl ← p_hl + 1;
        add tuple P to Q;
    }
    return Q;
}
```

Fig. 72. Closure function for Hamiltonian path algorithm

12.4 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 71; it invokes the closure function which is given in Figure 72. 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 appropriate data structures.

12.5 Remarks

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,

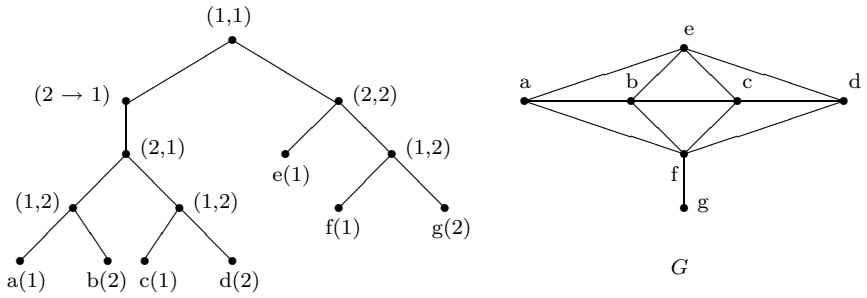


Fig. 73. A 2-NLC graph construction

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].

13. 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 relabelling operations. Again, let [k] denote the set of integers $\{1, 2, \ldots, k\}$. 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 $G_1 \cup G_2$ 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 $(G_1)_{i \to j}$ 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] $= \{1, \ldots, k\}$ using only the operations $G = G_1 \times_B G_2$ and $G = (G_1)_{i \to j}$. A 2-NLC construction is demonstrated in Figure 73.

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

13.1 Algorithm for maximum cardinality independent set in a k-NLC graph

Define:

$G[S]$ = max cardinality independent set that contains only labels from $S \subseteq [k]$
$G.max$ = max cardinality independent set

The algorithm is shown in Figure 74. It is demonstrated on the 2-NLC graph G shown in Figure 75. 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.

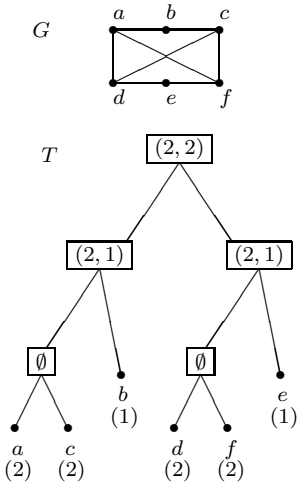
13.2 Algorithm for minimum cardinality dominating set in a k-NLC graph

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

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
    $\forall_{S \subseteq [k]}\ G[S] \leftarrow \max\{G_1[S_1] + G_2[S_2] : S = S_1 \cup S_2, (S_1 \times S_2) \cap E_B = \emptyset\}$
else if $G = (G_1)_{i \to j}$ then
    $\forall_{S \subseteq [k]}\ G[S] \leftarrow$ if $j \in S$ then $G_1[S \cup \{i\}]$ else $G_1[S - \{i\}]$
$G.max \leftarrow \max\{G[S] : S \subseteq [k]\}$

Fig. 74. Algorithm for maximum cardinality independent set in a k-NLC graph



Subgraph

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

Fig. 75. Maximum cardinality independent set in a 2-NLC graph

labels in $T \subseteq [k]$
$G.min =$ min cardinality dominating set

The algorithm is shown in Figure 76.

13.3 Remarks

The class of 1-NLC graphs is identical to the class of cographs, and every treewidth-k graph is a $(2^{k+1} - 1)$-NLC graph. In addition, every cliquewidth-k graph is a k-NLC graph. Finally, every k-NLC graph is a cliquewidth-$2k$ graph.

14. K-HB GRAPHS

k-HB (homogeneous balanced) graphs are graphs such that a particular $O(n^{k+2})$-time top-down decomposition algorithm constructs a pseudo-cliquewidth-$(k + 2^k)$ balanced 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: $V \to V_B$ 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 $y \in V_2$.

if $|V| = 1$ then
  let $v \in V$
  $\forall_{S,T \subseteq [k]}\ G[S,T] \leftarrow$ if S=$\{\text{label}(v)\}$ and $T = [k]$ then 1
    else if S=$\emptyset$ and T=$[k]$−$\{\text{label}(v)\}$ then 0
    else ∞
else if $G = G_1 \times_B G_2$ then
  $\forall_{S,T \subseteq [k]}\ G[S,T] \leftarrow \min\ \{G_1[S_1,T_1] + G_2[S_2,T_2] : S = S_1 \cup S_2,$
      $T = (T_1 \cup \{i : (i,j) \in E_B,\ j \in S_2\}) \cap (T_2 \cup \{j : (i,j) \in E_B,\ i \in S_1\})\ \}$
else if $G = (G_1)_{i \to j}$ 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. 76. Algorithm for minimum cardinality dominating set in a k-NLC graphe

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

Fig. 77. Algorithm for maximum cardinality independent set in a k-HB graph

This k-HB decomposition leads to polynomial-time algorithms for many problems on k-HB graphs, using recursion (top-down) rather than dynamic programming (bottom-up). 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].

14.1 Algorithm for maximum cardinality independent set in a k-HB graph

Define:

$G[S]$ = max cardinality independent set that contains only vertices in $S \subseteq V$
$G.indep$ = max cardinality independent set

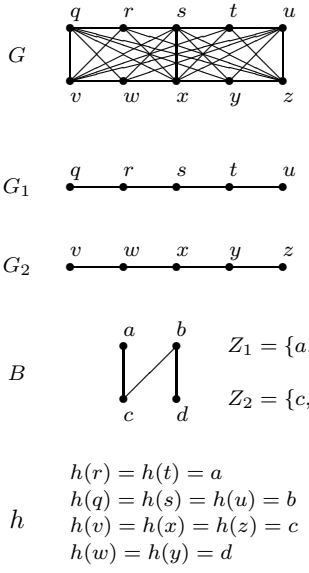
The algorithm is shown in Figure 77. We demonstrate this algorithm on the 2-HB graph G as shown in Figure 78. Note that $G = G_1 \times_{B,h} G_2$ where G_1, G_2, B, and h are as shown. The top level computations are summarized on the right. The maximum independent set has size 4, and the explicit solution is $\{r, t, w, y\}$.

14.2 Algorithm for maximum cardinality clique in a k-HB graph

Define:

$G[S]$ = max cardinality clique that contains only vertices in $S \subseteq V$
$G.clique$ = max cardinality clique

The algorithm is shown in Figure 79.



Recursive calls

$X = \emptyset \Rightarrow T = \emptyset \Rightarrow G_1[T] = 0$
$X = \{a\} \Rightarrow T = \{r, t\} \Rightarrow G_1[T] = 2$
$X = \{b\} \Rightarrow T = \{q, s, u\} \Rightarrow G_1[T] = 3$
$X = \{a, b\} \Rightarrow T = \{q, r, s, t, u\} \Rightarrow G_1[T] = 3$

$Y = \emptyset \Rightarrow U = \emptyset \Rightarrow G_2[U] = 0$
$Y = \{c\} \Rightarrow U = \{v, x, z\} \Rightarrow G_2[U] = 3$
$Y = \{d\} \Rightarrow U = \{w, y\} \Rightarrow G_2[U] = 2$
$Y = \{c, d\} \Rightarrow U = \{v, w, x, y, z\} \Rightarrow G_2[U] = 3$

Compatible subsets

$X = \emptyset, Y = \emptyset \Rightarrow G_1[T] + G_2[U] = 0 + 0 = 0$
$X = \emptyset, Y = \{c\} \Rightarrow G_1[T] + G_2[U] = 0 + 3 = 3$
$X = \emptyset, Y = \{d\} \Rightarrow G_1[T] + G_2[U] = 0 + 2 = 2$
$X = \emptyset, Y = \{c, d\} \Rightarrow G_1[T] + G_2[U] = 0 + 3 = 3$
$X = \{a\}, Y = \emptyset \Rightarrow G_1[T] + G_2[U] = 2 + 0 = 2$
$X = \{a\}, Y = \{d\} \Rightarrow G_1[T] + G_2[U] = 2 + 2 = 4$
$X = \{b\}, Y = \emptyset \Rightarrow G_1[T] + G_2[U] = 3 + 0 = 3$
$X = \{a, b\}, Y = \emptyset \Rightarrow G_1[T] + G_2[U] = 3 + 0 = 3$

Fig. 78. Maximum cardinality independent set in a 2-HB graph

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

Fig. 79. Algorithm for maximum cardinality clique in a k-HB graph

14.3 Algorithm for m-vertex colorability in a k-HB graph

Define:

$G[S_1, \ldots, S_m]$ = an m-coloring exists where only vertices in S_j can use color j
$G.colorable$ = an m-coloring exists

The algorithm is shown in Figure 80.

14.4 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 pseudo-cliquewidth decomposition is similar to a k-NLC decomposition, except that the vertex labels used at one node in the tree are not enforced at other nodes. A balanced decomposition of an n-vertex graph is a decomposition tree that has height $O(\lg\ n)$. The requirement that the decomposition must be balanced is more restrictive, while simultaneously the pseudo-cliquewidth condition is less

if $|V| = 1$ then
  $G[S_1, \ldots, S_m] \leftarrow (S_1 \cup \ldots \cup S_m = V)$
else if $G = G_1 \times_{B,h} G_2$ then
  $G[S_1, \ldots, S_m] \leftarrow \bigvee \{G_1[T_1, \ldots, T_m] \wedge G_2[U_1, \ldots, U_m] :$
    $X_j \subseteq h(V_1)$, $Y_j \subseteq h(V_2)$, $(X_j \times Y_j) \cap E_B = \emptyset$,
    $T_j = S_j \cap h^{-1}(X_j)$, $U_j = S_j \cap h^{-1}(Y_j)$, for $1 \leq j \leq m\}$
$G.colorable = G[V, \ldots, V]$

Fig. 80. Algorithm for m-vertex colorability in a k-HB graph

$(\exists \mathrm{V}_1) \ldots (\exists \mathrm{V}_m)$ $((\forall \mathrm{v}_1)$ $\mathrm{F}_0(\mathrm{v}_1 \in \mathrm{V}_1, \ldots, \mathrm{v}_1 \in \mathrm{V}_m)$
  $\wedge (\forall \mathrm{v}_2)$ $(\forall \mathrm{v}_3)$ $(\mathrm{Adjacent}(\mathrm{v}_2, \mathrm{v}_3) \rightarrow \wedge_{1 \leq i \leq j \leq m}$ $\mathrm{F}_{ij}(\mathrm{v}_2 \in \mathrm{V}_i, \mathrm{v}_3 \in \mathrm{V}_j))$
  $\wedge (\forall \mathrm{v}_4)$ $(\forall \mathrm{v}_5)$ $(\neg \mathrm{Adjacent}(\mathrm{v}_4, \mathrm{v}_5) \rightarrow \wedge_{1 \leq i \leq j \leq m}$ $\mathrm{F'}_{ij}(\mathrm{v}_4 \in \mathrm{V}_i, \mathrm{v}_5 \in \mathrm{V}_j)))$

Fig. 81. Format of MSOL″ predicates

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 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 to be 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 $\mathrm{Adjacent}(\mathrm{v}_i, \mathrm{v}_j)$ 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 any arbitrary way; rather every MSOL″ expression must possess the format shown in Figure 81.

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

As an example, the previous MSOL expressions for IndependentSet, Clique, and $\mathrm{VertexColorable}_m$ given in Figure 57 can be rewritten as equivalent MSOL″ expressions as shown in Figure 82. However, other MSOL expressions such as DominatingSet from Figure 57 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.

$\text{IndependentSet} \Leftrightarrow (\exists \text{V}_1)\ (\forall \text{v}_2)\ (\forall \text{v}_3)\ (\text{Adjacent}(\text{v}_2,\ \text{v}_3) \rightarrow \neg\ (\text{v}_2 \in \text{V}_1 \wedge \text{v}_3 \in \text{V}_1))$
$\text{Clique} \Leftrightarrow (\exists \text{V}_1)\ (\forall \text{v}_4)\ (\forall \text{v}_5)\ (\neg\ \text{Adjacent}(\text{v}_4,\ \text{v}_5) \rightarrow \neg\ (\text{v}_4 \in \text{V}_1 \wedge \text{v}_5 \in \text{V}_1))$
$\text{VertexColorable}_m \Leftrightarrow (\exists \text{V}_1) \ldots (\exists \text{V}_m)\ ((\forall \text{v}_1)\ (\text{v}_1 \in \text{V}_1 \vee \ldots \vee \text{v}_1 \in \text{V}_m)$
$\wedge (\forall \text{v}_2)\ (\forall \text{v}_3)\ (\text{Adjacent}(\text{v}_2,\ \text{v}_3) \rightarrow \wedge_{1 \le i \le m} \neg\ (\text{v}_2 \in \text{V}_i \wedge \text{v}_3 \in \text{V}_i)))$

Fig. 82. Some MSOL″ predicates

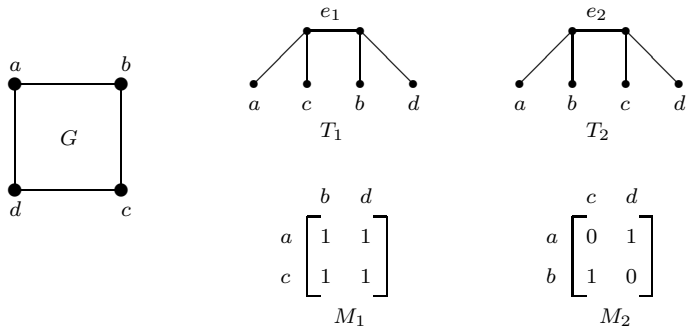


Fig. 83. A rankwidth-1 graph

15. RANKWIDTH-K GRAPHS

Rankwidth is a relatively new graph parameter that is defined and studied in [Oum and Seymour 2006], [Oum 2005a], and [Oum 2005b]. It is similar to the concept of branchwidth that was introduced in Section 10, and it also closely relates to the concept of cliquewidth covered in Section 12.

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 $\text{cutrank}_G(X, Y)$ 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 $\text{cutrank}_G(X) = \text{cutrank}_G(X, V - X)$.

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 $\text{cutrank}_G(f(X))$, 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 83. 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.

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 $O(V^9 lgV)$ 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 $O(V^9 lgV)$ to $O(V^4)$. Also, faster $O(V^3)$ algorithms are presented, but there is a tradeoff: the rankwidth is increased to $24k$ and the cliquewidth is increased 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. That is, 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.

Finally note that we do not take space to contrive an example algorithm on a rankwidth-k graph. Rather, one could proceed by exploiting the especially close relationship the structures have with cliquewidth-k graphs, i.e., a cliquewidth-k' algorithm could be applied to a rankwidth-k instance where k' depends on k.

16. SUMMARY AND CONCLUDING REMARKS

In Figure 84, 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 one 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', e.g., every treewidth-k graph is a branchwidth-$(k + 1)$ graph; co-graphs are cliquewidth-2 graphs, etc. A single, double-headed arrow indicates an equivalence between particular classes (e.g., partial k-trees and treewidth-k graphs) and a label of ∞ on an arc from class x to class y indicates that for a member in x, there exists a member in y with arbitrarily large parameter (e.g., for any k, there exist cliquewidth-k graphs having arbitrarily large treewidth.)

It should be evident that the schematic in Figure 84 is not, strictly speaking, complete in that a number of relationships are not explicitly exhibited; most of these are obvious however. For example, a directed arc, labeled $k = 1$, from trees to k-trees or one from series-parallel graphs to partial k-trees with label $k = 2$ is not shown; though valid, these are trivial or at least well-known relationships. Similarly, we could easily have directed an arc from trees (and various other classes) to bandwidth-k graphs carrying label ∞. Finally, though it should be clear, it may be instructive to remark why there are no arcs directed *from* the node identified with k-terminal graphs. As indicated in Section 1, 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.

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

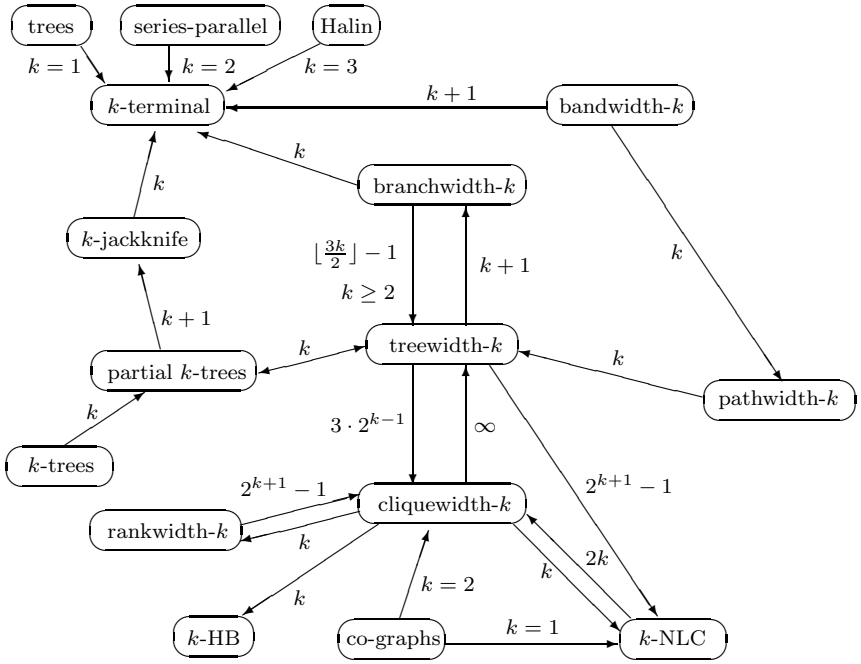


Fig. 84. Relationships between recursive graph classes

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 here in this regard (c.f., Sections 9.4-9.5) 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.

REFERENCES

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. Algebraic and 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 Applied Mathematics 54,* 2–3. Guest editors of special issue.

Arnborg, S., Lagergren, J., and Seese, D. 1991. Easy problems for tree-decomposable graphs. *Journal of Algorithms 12*, 308–340.

Arnborg, S. and Proskurowski, A. 1985. Characterization and recognition of partial k-trees. *Congressus Numerantium 47*, 69–75.

Arnborg, S. and Proskurowski, A. 1986. Characterization and recognition of partial 3-trees. *SIAM J. Algebraic and Discrete Methods 7*, 305–314.

Arnborg, S. and Proskurowski, A. 1989. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics 23*, 11–24.

Arnborg, S., Proskurowski, A., and Corneil, D. 1990. Forbidden minors characterization of partial 3-trees. *Discrete Mathematics 80*, 1–19.

Babel, L. and Olariu, S. 1998. On the structure of graphs with few p_4s. *Discrete Applied Mathematics 84*, 1–13.

Beineke, L. and Pippert, R. 1971. Properties and characterizations of k-trees. *Mathematik 18*, 141–151.

Bern, M., Lawler, E., and Wong, A. 1987. Linear time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms 8*, 216–235.

Bienstock, D., Robertson, N., Seymour, P., and Thomas, R. 1991. Quickly excluding a forest. *Journal of Combinatorial Theory Series B 52*, 274–283.

Bodlaender, H. 1987. Dynamic programming on graphs with bounded tree-width. Ph.D. thesis, Massachusetts Institute of Technology.

Bodlaender, H. 1990. Classes of graphs with bounded treewidth. Tech. Rep. RUU-CS-86-22, Utrecht University.

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. Computing 25*, 1305–1317.

Bodlaender, H., Gilbert, J., Hafsteinsson, H., and Kloks, T. 1995. Approximating treewidth, pathwidth, and minimum elimination tree height. *J. Algorithms 18*, 238–255.

Bodlaender, H. and Kloks, T. 1996. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithm 21*, 358–402.

Bodlaender, H. and Mohring, R. 1993. The pathwidth and treewidth of cographs. *SIAM J. Discrete Mathematic 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 of Operations Research 33*, 127–149.

Borie, R., Parker, R., and Tovey, C. 1991b. Deterministic decomposition of recursive graph classes. *SIAM J. Discrete Mathematics 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.

Brandstadt, A., Le, V., and Spinrad, J. 1999. *Graph Classes: A Survey*. SIAM Monographs on Discrete Mathematics and Applications, Philadelphia, PA.

Corneil, D., Habib, M., Lanlignel, J., Reed, B., and Rotics, U. 2000. Polynomial-time recognition of clique-width $\leq$ 3 graphs. In *4th Latin American Symposium on Theoretical Informatics*. Lecture Notes in Computer Science, vol. 1776. Springer, Punta del Este, Uruguay, 126–134.

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

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

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

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

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

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

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

Courcelle, B. 1992. The monadic second-order logic of graphs iii: Tree-decompositions, minors, and complexity issues. *Inform. Theorique Appl. 26*, 257–286.

Courcelle, B. 1995. The monadic second-order logic of graphs viii: Orientations. *Annals of Pure and Applied Logi 72*, 103–143.

Courcelle, B. 1996. The monadic second-order logic of graphs x: Linear orderings. *Theoretical Computer Scienc 160*, 87–143.

Courcelle, B., Engelfriet, J., and Rozenberg, G. 1993. Handle-rewriting hypergraph grammars. *Journal of Computer and Systems Sciences 46*, 218–270.

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

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

Courcelle, B. and Olariu, S. 2000. Upper bounds to the clique-width of graphs. *Discrete Applied Mathematics 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 graphs. *J. Math. Anal. Appl. 10*, 303–318.

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

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

Egervary, E. 1931. On combinatorial properties of matrices. *Mat. Lapok 38*, 16–28.

El-Mallah, E. and Colbourn, C. 1988. Partial k-tree algorithms. *Congressus Numerantium 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 *27th International Workshop on Graph Theory*. Lecture Notes in Computer Science, vol. 2204. Springer, Boltenhagen, Germany, 117–128.

Garey, M., Graham, R., Johnson, D., and Knuth, D. 1978. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics 34*, 477–495.

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

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

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

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

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

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

Jamison, B. and Olariu, S. 1995. Linear time optimization algorithms for p4-sparse graphs. *Discrete Applied Mathematics 61*, 155–175.

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

Johansson, O. 2000. Nlc 2-decomposition in polynomial time. *International J. Foundations of Computer Science 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 ISCAS*. IEEE, Kyoto, Japan, 1179–1182.

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

Kassios, I. 2001. Translating borie-parker-tovey calculus into mutumorphisms. Manuscript.

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

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

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

Oum, S. 2005a. Approximating rank-width and clique-width quickly. In *31st International Workshop on Graph Theory*. Lecture Notes in Computer Science, vol. 3787. Springer, Metz, France, 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. Combin. Theory Ser. 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, Victoria, BC, Canada, 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, England, 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. *Journal of Combinatorial Theory Series 35*, 39–61.

Robertson, N. and Seymour, P. 1984. Graph minors. iii. planar treewidth. *Journal of Combinatorial Theory Series B 36*, 49–64.

Robertson, N. and Seymour, P. 1986a. Graph minors. ii. algorithmic aspects of treewidth. *Journal of Algorithms 7*, 309–322.

Robertson, N. and Seymour, P. 1986b. Graph minors. v. excluding a planar graph. *Journal of Combinatorial Theory Series B 41*, 92–114.

Robertson, N. and Seymour, P. 1986c. Graph minors. vi. disjoint paths across a disc. *Journal of Combinatorial Theory Series B 41*, 115–138.

Robertson, N. and Seymour, P. 1988. Graph minors. vii. disjoint paths on a surface. *Journal of Combinatorial Theory Series B 45*, 212–254.

Robertson, N. and Seymour, P. 1990a. Graph minors. iv. treewidth and well-quasi-ordering. *Journal of Combinatorial Theory Series B 48*, 227–254.

Robertson, N. and Seymour, P. 1990b. Graph minors. ix. disjoint crossed paths. *Journal of Combinatorial Theory Series B 49*, 40–77.

Robertson, N. and Seymour, P. 1990c. Graph minors. viii. a kuratowski theorem for general surfaces. *Journal of Combinatorial Theory Series B 48*, 255–288.

Robertson, N. and Seymour, P. 1991. Graph minors. x. obstructions to tree-decompositions. *Journal of 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. *Journal of Combinatorial Theory Series B 60*, 72–106.

Robertson, N. and Seymour, P. 1995. Graph minors. xiii. the disjoint paths problem. *Journal of Combinatorial Theory Series B 63*, 65–110.

Robertson, N. and Seymour, P. 2003. Graph minors. xvi. excluding a non-planar graph. *Journal of Combinatorial Theory Series B 89*, 43–76.

Robertson, N. and Seymour, P. 2004. Graph minors. xx. wagner's conjecture. *Journal of Combinatorial Theory Series B 92*, 325–357.

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

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

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

Satyanarayana, A. and Tung, L. 1990. A characterization of partial 3-trees. *Networks 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 *Proceedings of the Fischland Colloquium on Discrete Mathematics and Applications*. Wustrow, Rostock, Germany.

Scheffler, P. 1989. The treewidth of graphs as a measure for the complexity of algorithmic problems. Ph.D. thesis, Akademie der Wissenschaften der DDR.

Scheffler, P. and Seese, D. 1986. Graphs of bounded tree-width and linear-time algorithms for np-complete problems. In *Proceedings of the Bilateral Seminar*. Samarkand, Uzbek, USSR.

Scheffler, P. and Seese, D. 1988. A combinatorial and logical approach to linear-time computability. In *European Conference on Computer Algebra*. Lecture Notes in Computer Science, vol. 378. Springer, Leipzig, Germany, 379–380.

Seymour, P. and Thomas, R. 1993. Graph searching and a min-max theorem for treewidth. *Journal of 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 9th Workshop on Graph-Theoretic Concepts in Computer Science*. Trauner, Haus Ohrbeck, Germany, 342–353.

Takamizawa, K., Nishizeki, T., and Saito, N. 1982. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the ACM 29*, 623–641.

Valdes, J., Tarjan, R., and Lawler, E. 1982. The recognition of series parallel digraphs. *SIAM Journal on Computing 11*, 298–313.

Vizing, V. 1964. On an estimate of the chromatic class of a p-graph. *Diskret. Analiz. 3*, 25–30.

Wanke, E. 1994. k-nlc graphs and polynomial algorithms. *Discrete Applied Mathematics 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 Numerantium 63*, 161–176.

Wimer, T., Hedetniemi, S., and Laskar, R. 1985. A methodology for constructing linear graph algorithms. *Congressus Numerantium 50*, 43–60.