1

Graph algorithms

MARTIN CHARLES GOLUMBIC

This chapter provides an introduction to basic graph algorithms and structured graph classes. It presents how they have developed into classical topics of study, necessary for advanced computing applications, and lead to new mathematical research along the way.

1. Introduction

Algorithms lie at the heart of solving graph problems constructively. One of the earliest examples is searching a graph. What we know today as *depth-first search* was introduced to search labyrinths in the early 1880s by Trémaux (see [52]) and Tarry [66], and by Fleury in his 1883 algorithm to produce Eulerian chains [17]. Another example is the minimum spanning tree problem for optimally connecting facilities, whose first algorithm was given by Borůvka in 1926 (see [41] and [61]).

As researchers increasingly modelled real-world problems in terms of graphs, the need for provably correct new algorithms grew, involving a wide range of applications. Optimally colouring the vertices of a 100-vertex graph was a mathematical challenge, limited by pencil, paper and human brainpower. Until the 1950s no one needed to colour a graph with thousands of vertices, and nor could they expect to do so. And then came the computer.

In this chapter, we review some basic graph algorithms that have given rise to new research areas in discrete mathematics and computer science. Often driven by applications of structured graph classes, they have developed into classical topics of study – the theme of this book.

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

What is algorithmic graph theory?

Ask two people and you will get three answers. Here is one on which they will both agree. The main research theme of algorithmic graph theory is the investigation, discovery and exploitation of structural properties of graphs to help in efficiently solving computational problems. Some problems may be intrinsically hard, like graph colouring, while others may be easily tractable, like minimum spanning tree. Whether the problem arises in computer science, optimization, biology, neuroscience, engineering or another application area, or is raised in the abstract imagination of a theoretical researcher, revealing the mathematical properties satisfied *a priori* by its structure often enables finding an algorithm and reducing the time or space complexity required to solve it.

Conversely, the algorithmic approach frequently leads to startling graph-theoretical results, as is evident in every chapter in this book. We might call this 'from algorithms to structure'. For instance, a greedy algorithm gives rise to an underlying matroid associated with the graph, which itself may have special combinatorial properties. Problems from psychology to biology generated new notions in graph theory, like boxicity, phylogenetic trees and indifference graphs (see [62], [63]). Something similar happened when relational database schemes spawned a new hierarchy of acyclic hypergraphs.

This symbiotic relationship was recognized a half-century ago with the establishment of the annual events mentioned earlier in the foreword. The fertile interaction between applications and discrete mathematics kick-started a worldwide throng of dozens of new journals, hundreds of conferences and workshops and thousands of research papers. This is the beauty of algorithmic graph theory and its importance as a discipline.

Efficient algorithms

Complexity analysis deals with the quantitative aspects of problem-solving. It addresses the issue of what can be computed within a practical or reasonable amount of time and space by measuring the resource requirements – exactly, or by obtaining upper and lower bounds for them. Complexity is actually determined at three levels: the problem, the algorithm and the implementation. Naturally, we want the best algorithm that solves our problem, and we want to choose the best implementation of that algorithm.

The computational complexity of the search tree algorithm depends on how the graph is represented and on what data structure is used to maintain the candidates according to their priority. Testing whether a graph is connected can be done in linear-

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

time in the size of the graph – that is, $O(n + m)$, using adjacency lists, where n is the number of vertices and m is the number of edges. This is considered the best that one could expect for any non-trivial graph problem since every vertex and every edge would probably have to be examined at least once. A typical implementation of Prim's algorithm for finding a maximum spanning tree could be $O(m \log n)$ or $O(m + n \log n)$, depending on the data structure (see Section 2).

Computing the chromatic number of a graph is a hard problem – in fact, determining whether a graph is 3-colourable is NP-complete. In contrast to this, although computing the clique number of a graph is also NP-complete, determining whether $\omega(G) \leq k$ for a *fixed* value of k has computational complexity $O(n^k)$. So there is something intrinsically different between these two problems. It places MAXCLIQUE in the complexity class known as $\mathcal{XP}$. A further difference led researchers to develop a new branch of complexity theory, known as *fixed parameter tractability* (FPT), which contains problems solvable in $O(f(k)n^{O(1)})$-time, where f is an arbitrary function depending only on k. VERTEX COVER and many other NP-complete problems have FPT algorithms, whereas others do not, like COLOURING. Recent books in this area are [11] and [18].

Besides time-complexity, we may be interested in space-complexity. A polynomial-time algorithm clearly consumes only polynomial space, but NP-hard problems also consume only polynomial space. More than that, polynomial space problems can be solved in exponential time. A fundamental open question in theoretical computer science is whether each of the following inclusions is strict:

$$\mathcal{P} \subseteq \mathcal{FPT} \subseteq \mathcal{XP} \subseteq \mathcal{NP} \subseteq \mathcal{P}\text{-space} \subseteq \mathcal{EXP}\text{-time}$$

2. Graph search algorithms

Consider the problem of determining whether a graph G is connected. A mathematically elegant solution is the following: G is connected if and only if $\mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \mathbf{M}^3 + \cdots + \mathbf{M}^{n-1}$ has no zero entries, where $\mathbf{M}$ is the adjacency matrix of G, $\mathbf{I}$ is the identity matrix and n is the order of G. However, using this method as an algorithm would require much more work (matrix multiplication and addition) than is actually needed to test connectivity. A better way would be to traverse the edges of the graph.

Searching and spanning trees

How to traverse and search a graph is fundamental to many graph algorithms. The following generic search algorithm, SEARCHST, tests connectivity and finds a spanning tree efficiently.

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

Algorithm SEARCHST**: Search Spanning Tree**

STEP I: Start with a tree T consisting of one arbitrary vertex.

STEP II: **if** T contains all the vertices of G, **then** STOP;
comment *T is a spanning tree.*
else do STEP III.

STEP III: Add to T an edge vw which joins a vertex w not yet in T
with a vertex v already in T.
if no such edge exists, **then** STOP;
comment *There is no spanning tree; G is not connected.*
else go-to STEP II.

In STEP III of the SEARCHST algorithm, there may be several edges vw eligible for adding to T. We call such an edge a *candidate edge*. Various priorities can be established to guide the choice of candidates, and each priority yields a slightly different algorithm. For example, if candidates are stored in a queue (first-in, first-out), then SEARCHST is a *breadth-first search* (BFS) of G, and the set T of chosen edges vw is a breadth-first search tree of G. Breadth-first search appears in many shortest path applications.

If candidates are stored in a stack (last-in, first-out), then SEARCHST is a *depth-first search* (DFS), and T is a depth-first search tree of G. Depth-first search is used in many algorithms, including topological sorting and finding strongly connected components.

If the edges have weights associated with them, and if the candidate with minimum weight is always chosen, then SEARCHST produces a minimum spanning tree (MINST). This is the algorithm originally developed in 1930 by V. Jarník and later rediscovered by R. C. Prim 25 years later.

Other specializations of SEARCHST, with suitable priorities for choosing candidates, are found throughout algorithmic graph theory. In Chapters 6 and 7, for example, maximum cardinality search, lexicographic breadth-first search and maximum neighbourhood search are used for solving problems on chordal graphs and strongly chordal graphs. Similarly, shortest path algorithms, critical path algorithms, the heuristic search algorithm A^* in artificial intelligence and others can all be viewed as adaptations of SEARCHST.

Expanding spanning forests

Searching a graph with the SEARCHST algorithm is analogous to a 'boots-on-the-ground' approach, like Prim's army pushing forward the frontier of a minimum spanning tree until the entire graph is captured. In contrast to this, as we will see

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

next, a 'drones-in-the-sky' approach by Kruskal's air force combines smaller trees into larger ones until the entire graph is spanned. The following algorithm, EXPANDST, illustrates the expanding forest method for finding a spanning tree of a graph.

Algorithm EXPANDST: Expanding Spanning Tree

STEP I: Start with a forest T consisting of the n vertices and no edges.
STEP II: **if** T contains $n - 1$ edges of G, **then** STOP;
comment *T is a spanning tree.*
else do STEP III.
STEP III: Add to T an edge vw which joins a vertex w in one subtree of T with a vertex v in a different subtree of T.
if no such edge exists, **then** STOP;
comment *There is no spanning tree; G is not connected.*
else go-to STEP II.

In STEP III of the EXPANDST algorithm, as in the SEARCHST algorithm, there may be several candidate edges vw eligible for adding to T, according to the priority established for that instance of the algorithm.

If the edges have weights and the candidate with minimum weight is always chosen, then EXPANDST is precisely *Kruskal's minimum spanning tree algorithm*. It has complexity $O(m \log m)$, since the edges must be sorted by their weight. This may be greater than Prim's algorithm, depending on the graphs and the data structures used. Faster algorithms have been designed, but the best possible complexity of MINST is not known.

Algorithms, like EXPANDST, that join smaller trees into larger trees occur in other contexts as well – for example, in merging heaps in heapsort, balancing search trees and in a variety of clustering algorithms.

Shortest path problems

The starting vertex in STEP I of algorithm SEARCHST is the *root* of the spanning tree T. For unweighted graphs, breadth-first search provides a shortest path from the root to each of the other vertices in the graph. For graphs with edge weights, where the length of a path is the sum of the weights of its edges, a more sophisticated algorithm must be employed. One of these is *Dijkstra's algorithm* SPT, shown below. It can be viewed as a variation of SEARCHST, using a type of best-first search as new vertices join the spanning tree.

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

Algorithm SPT**: Shortest Path Tree**

STEP I: Let Q be a data structure containing $V(G)$;
for each vertex v **do**
parent(v) := UNDEFINED; [*its 'tentative' parent in the rooted tree* T]
dist(v) := ∞; [*its 'tentative' shortest distance from the root*]
end-for

STEP II: Start with a tree T consisting of one arbitrary vertex r [*the root*];
dist(r) := 0; remove r from Q;
for each neighbour v of r **do**
parent(v) := r;
dist(v) := weight(v, r);
end-for

STEP III: **while** Q is not empty **do**
w := the vertex in Q with minimum dist(w);
if dist(w) := ∞ **then** STOP;
comment *There is no spanning tree; G is not connected.*
else remove w from Q and add the edge $\{w, parent(w)\}$ to T;
for each neighbour v of w that is still in Q **do**
if change := weight(v, w) + dist(w) $<$ dist(v) **then do**
parent(v) := w;
dist(v) := change;
comment *Changing the parent shortens the 'tentative' path from v to r.*
end-if
end-for
end-while

Depending on the data structure maintaining the priority queue Q, the complexity of implementing Dijkstra's algorithm can be as low as $O(m + n \log n)$.

Another algorithm that computes shortest paths from a single source vertex to all of the other vertices is the *Bellman–Ford algorithm*. It is slower than Dijkstra's algorithm for the same problem, but it is capable of handling graphs in which some of the edge weights are negative numbers and can detect negative cycles in the graph. Faster shortest paths in dense distance graphs can be found in [59].

All pairs shortest path and the diameter of a graph

To calculate the shortest distance between *all pairs of vertices*, one could run Dijkstra's algorithm or the Bellman–Ford algorithm n times – by making each vertex the root. However, an algorithm due to Floyd and Warshall with $O(n^3)$-complexity would generally be better for non-sparse graphs. It can be found in numerous books on algorithms. No truly sub-cubic algorithm is known for the all pairs shortest distance problem – that is, in $O(n^{3-\varepsilon})$-time, for any fixed $\varepsilon > 0$.

The Floyd–Warshall algorithm is often used to compute the transitive closure and in various matrix applications. A novel use in temporal reasoning is found in the

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

survey [23]. The *diameter* of a graph (directed or weighted) is the largest distance between a pair of vertices. For general graphs, the current fastest way to compute the diameter of a graph is by computing all pairs of shortest paths between its vertices and taking the largest.

For planar graphs, in a breakthrough result, Cabello [7] presented the first subquadratic algorithm for computing the diameter of a directed planar graph. It was a randomized algorithm running in $\tilde{O}(n^{11/6})$-time. The *soft-O* notation $\tilde{O}(f(n))$ means $O(f(n)\log^k n)$, for some k, a convenient shorthand that ignores logarithmic factors just as big-O ignores constant factors. Shortly after, in [21], the algorithm was made deterministic and the running time was improved to $\tilde{O}(n^{5/3})$. The improvement was by designing an efficient construction of Voronoi diagrams. Computing the diameter is a fundamental problem in graph algorithms, with numerous applications and research papers studying its complexity.

Further reading on shortest path and other connectivity problems, such as biconnectivity, 2-edge connectivity, strong connectivity of digraphs and others, can be found in the many standard books on algorithms – for example, [10], [14], [15], [24] and [64].

3. Greedy graph colouring

Graph colouring is a computationally hard problem, yet thousands of applications rely on solving such problems, from scheduling and resource allocation, to circuit and software design. We have two ways of coping with this potential intractability:

(1) exploiting *a priori* knowledge about the expected structure of the graphs we must colour, in order to design efficient special purpose algorithms, and
(2) empirically testing a variety of heuristic algorithms to see what works best most of the time.

Heuristics offer no guarantees, but sometimes structure and heuristics can work together.

Greedy colouring is the simplest and most common type of algorithm, where the vertices are coloured successively, according to some priority ordering, with each being assigned a colour different from any colour already assigned to one of its neighbours. It is called a *first-fit greedy colouring* if the colours are numbered and the colour to be assigned is the smallest among those unused for its neighbours.

We often distinguish between two cases. *Static greedy colouring* chooses a fixed ordering for colouring the vertices in advance. *Dynamic greedy colouring* maintains a data structure for the uncoloured vertices, and uses a priority or heuristic to choose the next vertex to be coloured.

The problem of colouring interval graphs, the intersection graphs of intervals on a line, provides an example where applying static first-fit greedy colouring is optimal, by colouring them according to the order of the left endpoints of the intervals. The same is true for many other classes of perfect graphs. A chordal graph can be optimally coloured by applying greedy colouring to the vertices in the reverse of

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

a perfect elimination ordering, and a comparability graph by a topological sorting of a transitive orientation (see [24]). Similar specialized methods apply to tolerance graphs [39] and others.

For unstructured graphs, a popular greedy colouring heuristic is *postponing the vertex of smallest degree*. Delete a vertex of smallest degree, thus reducing the degree of each neighbour by 1. Continue in this fashion, until all vertices are deleted. Then greedily colour the vertices in the reverse order of their deletion. This method was demonstrated to be effective and efficient for register allocation in compilers (see Chapter 6, Section 7).

A better heuristic for general graphs is the well-known DSATUR algorithm by Daniel Brélaz [4]. It successively chooses the vertex adjacent to the largest number of different colours, its *saturation degree*.

Algorithm DSATUR: Degree of Saturation

STEP I: Colour a vertex of maximal degree with colour 1.
STEP II: Choose a vertex with a maximal saturation degree,
breaking ties by maximal degree in the uncoloured subgraph.
STEP III: Colour the chosen vertex using first-fit;
repeat STEP II until all vertices are coloured.

The literature overflows with other dynamic colouring algorithms, some greedy and others employing various levels of backtracking strategies to obtain better solutions, at the cost of higher complexity. For further reading, see Dechter [13] on constraint-based heuristics, Fomin and Kratsch [19] on exact exponential algorithms, Husfeldt [46] on graph colouring algorithms and Lewis [51] on practical applications.

In Chapter 2 of this book, Alain Hertz and Bernard Ries discuss three graph colouring variations – selective colouring, online colouring and mixed graph colouring – each motivated by applications. Chapter 3, by Celina de Figueiredo, is a survey of total colouring – assigning a colour to each vertex and edge of a graph, so that there are no incidence colour conflicts. Both theoretical and algorithmic results are considered for this alternative colouring problem.

4. The structured graph approach

Exploiting graph structure is one of the fundamental approaches to designing efficient algorithms. To solve important practical problems, algorithmic graph theory strives to understand and use the underlying properties and form of the specific graphs expected as input. This is especially true for special classes that naturally arise in applications. In this section, we discuss several of these classes.

Planar graphs

Planar graphs illustrate well the idea of exploiting structure to help solve problems efficiently. We saw an example of this for computing the diameter of a planar graph

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

in sub-quadratic time at the end of Section 2. A further result is an approximation algorithm for the diameter of planar graphs in near-linear time [68].

In recent years the frontier of research on algorithms for optimization problems in planar graphs has been pushed forward (see [50]). An example of this is an $O(n \log\log n)$-time algorithm for computing the minimum cut (or equivalently, the shortest cycle) of a weighted directed planar graph [58]. Another result on minimum cut, for general graphs, is an $O(m \log^2 n)$-time algorithm in [22]. Improving algorithms on planar graphs automatically leads to progress on applications to basic problems in computer vision, navigation on road networks and circuit design.

A *distance oracle* is a data structure maintaining a preprocessed compact representation of a graph from which the distance or a shortest path between any pair of vertices can be retrieved efficiently. Distance oracles are useful in applications ranging from geographic information systems, databases, packet routing and logistics, to computer games, web search, computational biology and social networks. There are natural trade-offs between space and query-time for exact distance oracles which are studied for directed weighted planar graphs by Charalampopoulos *et al.* [8]. Their results are almost optimal in the sense that they are within polylogarithmic, subpolynomial or arbitrarily small polynomial factors from the naïve linear-space constant query-time lower bound. These trade-offs include an oracle with space $O(n^{1+\varepsilon})$ and query-time $\tilde{O}(1)$ for any constant $\varepsilon > 0$, an oracle with space $\tilde{O}(n)$ and query-time $O(n^{\varepsilon})$ for any constant $\varepsilon > 0$ and an oracle with space $n^{1+o(1)}$ and query-time $n^{o(1)}$. These bounds were achieved by designing an elegant and efficient point location data structure for Voronoi diagrams on planar graphs.

Intersection graphs

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be a collection of subsets of a set S. Recall that the *intersection graph* of $\mathcal{S}$ is the graph G obtained by assigning a distinct vertex v_i of G for each set S_i in $\mathcal{S}$ and joining two vertices by an edge precisely when their corresponding sets have a non-empty intersection – that is, $v_i v_j \in E(G)$ if and only if $i \neq j$ and $S_i \cap S_j \neq \varnothing$. The collection $\mathcal{S}$ is called a *representation* of G on the *host* S.

Interval graphs are the oldest and best-known family of intersection graphs. They are the intersection graphs of intervals on a line, and arise in scheduling problems, bioinformatics, temporal reasoning and many other applications. Interval graphs have been characterized mathematically and algorithmically in many settings, and generalized to larger families such as tolerance graphs and trapezoid graphs, all of which have efficient polynomial-time algorithms for recognition, colouring, maximum clique and many other optimization problems (see [12], [16] and [39]). So much has been written about intersection graphs in other books (for example, [3], [24], [53], [54] and [65]) that we refrain from repeating it here. Rather, we cover just a few recent results.

5. Specialized classes of intersection graphs

There are two important aspects to consider when defining a class of intersection graphs. The first aspect is the *type of subsets* and their host. For example, *proper*

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

interval graphs restrict the collection $\mathcal{S}$ to be intervals that are pairwise incomparable – that is, no interval properly contains another. Similarly, *unit disc graphs* are the intersection graphs of solid circles of radius 1 in the plane. The recognition, 3-colouring, Hamiltonian and maximum independent set problems have linear-time solutions on proper interval graphs, but they are all NP-hard for unit disc graphs. Only the maximum clique problem is tractable for unit disc graphs given a unit disc representation for the graph.

The second aspect is the *type of intersection* which may vary from the usual set intersection or some other form of 'interference', such as measured intersection. For example, for fixed $k \geq 1$, *k-edge-intersection graphs of paths in a tree* (k-EPT graphs) have a representation as paths S_i in a tree T, where two vertices v_i and v_j are adjacent if S_i and S_j share at least k edges in T (see [30]). Recognition and 3-colouring of k-EPT graphs are NP-complete for any fixed $k \geq 1$, although maximum clique is polynomial. A further generalization is the $\langle h, s, t \rangle$ graphs which are discussed below. Edge-intersection graphs are used in network applications, such as scheduling calls in a tree network or assigning wavelengths to virtual connections in an optical network, problems that are equivalent to colouring an EPT graph.

These two aspects, type of subsets and type of intersection, may be combined as well, for example, in the class of *unit neighbourhood subtree tolerance graphs* (unit NeST graphs; see [2] and [42]).

Tolerance graphs

Tolerance graphs were introduced in 1982 as a natural extension of interval graphs (see [35] and [36]). Each vertex is associated with an interval on the real line and a positive number called its *tolerance*. A tolerance is considered *unbounded* if it exceeds the length of the interval. Two vertices are adjacent if and only if the length of the intersection of their associated intervals is not less than the tolerance of one of them. We can think of two meetings that are set to overlap in time, yet are assigned to the same meeting room. In the interval graph model they conflict; in the tolerance model, if both are sufficiently tolerant, they do not.

This tolerance–conflict model set the stage for decades of further research on multiple themes – special families of tolerance graphs and their properties, directed graph versions, generalizations beyond intervals and restricted models. All of these involve some notion of measured intersection, known as *tolerance*. We have bounded, proper and unit tolerance graphs, several types of tree tolerance graphs, rank tolerance graphs [28], Archimedean ϕ-tolerance graphs [29] and others (see [25] and [39]).

The computational complexity of recognizing tolerance graphs and bounded tolerance graphs had remained open for 28 years. Hayward and Shamir [43] showed that the problem is in NP, and Mertzios, Sau and Zaks [56], [57] proved that it is NP-hard. Thus we have the following result.

Theorem 5.1 *Recognizing tolerance graphs and bounded tolerance graphs are NP-complete problems.*

The following result answers the complexity question for bipartite graphs [6].

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

Theorem 5.2 *Recognizing bipartite tolerance graphs has linear-time complexity.*

Narasimhan and Manber [60] presented a polynomial-time algorithm to find a maximum weighted stable set of a tolerance graph, given a tolerance representation for the graph.

Theorem 5.3 *A maximum weighted stable set of a tolerance representation can be found in time* $O(n^2 \log n)$.

Colouring bounded tolerance graphs in polynomial time is an immediate consequence of their being cocomparability graphs. Golumbic and Siani [38] gave an algorithm to find a colouring of a tolerance graph, whose complexity depends on the number of unbounded tolerances in a given tolerance representation for it.

Theorem 5.4 *A minimum colouring of a tolerance representation with at most* q *intervals having unbounded tolerance can be found in* $O(qn + n \log n)$*-time.*

Tolerance graphs on trees

Let T be a tree and let $\{T_i\}$ be a collection of subtrees (connected subgraphs) of T. We may think of the host tree T as either

(1) a continuous model of a tree embedded in the plane, thus generalizing the real line from the 1-dimensional case, or
(2) a finite discrete model of a tree, a connected graph of vertices and edges having no cycles, thus generalizing the graph P_k from the 1-dimensional case.

The distinction between these two models becomes important when measuring the size of the intersection of two subtrees. For example, in the continuous model (1), we might take the size of the intersection to be the length of a longest common path of the two subtrees measured along the host tree (see [2]). In the discrete model (2), we might count the number of common vertices or common edges (see [26], [27], [48] and [49]). Typically, one uses the expressions 'non-empty intersection' or 'vertex-intersection' to mean sharing a vertex of T (or a point, in the continuous model), and 'non-trivial intersection' or 'edge-intersection' to mean sharing an edge or otherwise measurable segment of T. In this way, edge-intersection is more tolerant than vertex-intersection. Using this terminology, we have the following classical result of Buneman [5], Gavril [20] and Walter [67].

Theorem 5.5 *A graph is the vertex-intersection graph of a set of subtrees of a tree if and only if it is a chordal graph.*

McMorris and Shier [55] gave an analogous version for split graphs.

Theorem 5.6 *A graph* G *is the vertex-intersection graph of distinct induced subtrees of a star* $K_{1,n}$ *if and only if* G *is a split graph.*

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

In contrast to these results, it was observed in [26] that the family of edge-intersection graphs of subtrees of a tree yield all possible graphs. In fact, the following variation on Marczewski's theorem holds.

Theorem 5.7 *Every graph can be represented as the edge-intersection graph of substars of a star.*

Two different classes of intersection graphs also arise when considering simple paths (instead of subtrees) of an arbitrary host tree T. The class of *path graphs*, which are the vertex-intersection graphs of paths on a tree (also known as *VPT graphs*), are a subfamily of chordal graphs and inherit all their nice algorithmic properties. However, the graphs obtained as the edge-intersection graphs of paths in a tree (called *EPT graphs*) are not necessarily chordal. EPT graphs are not perfect graphs, and the recognition problem for them is NP-complete, whereas the VPT graphs are perfect and can be recognized efficiently. The same dichotomy between EPT and VPT holds for colouring. Thus, EPT graphs are a more tolerant model than VPT graphs, but they have a high algorithmic cost.

In 1985 Golumbic and Jamison [27] showed that, in the special case where the host tree T has maximum vertex-degree 3 (binary trees), the VPT and EPT classes are the same. This led to a broader study of degree-constrained subtree representations, which we now describe.

Jamison and Mulder [48], [49] introduced a constant tolerance model for subtrees of a tree where degree restrictions are placed on the trees, further generalizing VPT and EPT graphs. An $\langle h, s, t\rangle$-representation of a graph G consists of a collection of subtrees $\{S_v : v \in V(G)\}$ of a tree T, such that

(i) the maximum degree of T is at most h,
(ii) every subtree has maximum degree at most s,
(iii) there is an edge between two vertices in G if and only if the corresponding subtrees in T have at least t vertices in common.

Using this notation, where ∞ denotes that no restriction is imposed, we immediately see the equivalence of many familiar graph classes within this model:

- $\langle 2, 2, 1\rangle \equiv$ interval graphs;
- $\langle \infty, 2, 1\rangle \equiv$ VPT graphs or path graphs;
- $\langle \infty, 2, 2\rangle \equiv$ EPT graphs;
- $\langle \infty, 2, k-1\rangle \equiv k$-EPT graphs;
- $\langle \infty, \infty, 1\rangle \equiv \langle 3, 3, 1\rangle \equiv \langle 3, 3, 2\rangle \equiv$ chordal graphs;
- $\langle 3, 2, 1\rangle \equiv \langle 3, 2, 2\rangle \equiv$ VPT $\cap$ chordal $\equiv$ EPT $\cap$ chordal;
- $\langle 4, 2, 2\rangle \equiv$ EPT $\cap$ weakly chordal.

In a series of papers, Cohen, Golumbic, Lipshteyn and Stern also characterized the classes $\langle 4, 4, 2\rangle$ and $\langle 4, 3, 2\rangle$ and gave polynomial-time recognition algorithms for them (see [9], [30], [31], [32] and [33]). The class $\langle 3, 3, 3\rangle$ is studied in [48]. For further results in this area, see [25], [49] and [39].

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

Intersection graphs of paths on a grid

We conclude this section with another pair of graph classes that contrast the difference between vertex-intersection and edge-intersection, this time with paths on a grid. They were motivated by applications in circuit layout and chip manufacturing, but could be equally applied to traffic routing, scheduling and other natural problems.

A *vertex-intersection graph of paths on a grid* (or *VPG graph*) is a graph for which there exists a family of paths on a grid in one-one correspondence with its vertex-set for which two vertices are adjacent if and only if the corresponding paths share at least one grid-point. An *edge-intersection graph of paths on a grid* (or *EPG graph*), is defined similarly, with the exception that two vertices are adjacent if and only if the corresponding paths share at least one grid-edge. A recent survey on EPG graphs appears in [37].

It was shown in [1] that VPG graphs are equivalent to the class of string graphs. This class is NP-complete to recognize and NP-hard to colour. On the other hand, for edge-intersection, it was shown in [34] that every graph is an EPG graph. In both cases, to make these models relevant to real-world applications, it is natural to restrict each path to a limited number of bends – that is, 90-degree turns at a grid-point. A representation is called B_k if each path has at most k bends. We consider the classes B_k-VPG and B_k-EPG graphs, for various values of $k \geq 0$ – that is, those which admit a B_k representation in the VPG model and the EPG model, respectively. They are very different classes. The B_0-EPG graphs are equivalent to interval graphs, which have efficient algorithms for most computational problems. The B_0-VPG graphs are equivalent to the intersection graphs of horizontal and vertical segments in the plane, which are NP-complete to recognize and NP-hard to colour.

Another difference is in determining the minimum number of bends required to represent all graphs in some given class. For example, every planar graph has a B_1-VPG representation and there are planar graphs that require at least one bend [40]. However, for EPG representations, every planar graph has a B_4-EPG representation, and there are planar graphs that are not B_2-EPG (see [44] and [45]). So whether the EPG bend-number of a planar graph is 3 or 4 remains an open question.

This has been a taste of some of the many computational results and challenges involving graph parameters and special graph classes. A database that is worth consulting for further references is *ISGCI: Information system on graph classes and their inclusions* [47].

References

1. A. Asinowski, E. Cohen, M. C. Golumbic, V. Limouzy, M. Lipshteyn and M. Stern, Vertex intersection graphs of paths on a grid, *J. Graph Algorithms Appl.* **16** (2012), 129–150.
2. E. Bibelnieks and P. M. Dearing, Neighborhood subtree tolerance graphs, *Discrete Appl. Math.* **43** (1993), 13–26.
3. A. Brandstädt, V. B. Le and J. P. Spinrad, *Graph Classes: A Survey,* SIAM Monographs on Discrete Math. Appl. **3**, 1999.

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

4. D. Brélaz, New methods to color the vertices of a graph, *Comm. Assoc. Comput. Mach.* **22** (1979), 251–256.
5. P. Buneman, A characterisation of rigid circuit graphs, *Discrete Math.* **9** (1974), 205–212.
6. A. H. Busch and G. Isaak, Recognizing bipartite tolerance graphs in linear time, *Proc. 33rd Conf. on Graph-Theoretic Concepts in Computer Science (WG'07)*, Lecture Notes in Computer Science **4769**, Springer (2007), 12–20.
7. S. Cabello, Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs, *ACM Trans. Algorithms* **15** (2018), 21:1–38.
8. P. Charalampopoulos, P. Gawrychowski, S. Mozes and O. Weimann, Almost optimal distance oracles for planar graphs, *Proc. 51st ACM Symp. Theory of Computing* (2019), 138–151.
9. E. Cohen, M. C. Golumbic, M. Lipshteyn and M. Stern, Tolerance intersection graphs of degree bounded subtrees of a tree with constant tolerance 2, *Discrete Math.* **340** (2017), 209–222.
10. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms* (3rd edn.), MIT Press, 2009.
11. M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk and S. Saurabh, *Parameterized Algorithms*, Springer, 2015.
12. I. Dagan, M. C. Golumbic and R.Y. Pinter, Trapezoid graphs and their coloring, *Discrete Applied Math.* **21** (1988), 35–46.
13. R. Dechter, *Constraint Processing*, Morgan Kaufmann, 2003.
14. J. Erickson, *Algorithms*, 2019. *http://jeffe.cs.illinois.edu/teaching/algorithms/*.
15. S. Even, *Graph Algorithms* (2nd edn.) (ed. G. Even), Cambridge Univ. Press, 2011.
16. S. Felsner, R. Müller and L. Wernisch, Trapezoid graphs and generalizations, geometry and algorithms, *Discrete Applied Math.* **74** (1997), 13–32.
17. M. Fleury, Deux problèmes de géométrie de situation, *J. Mathématiques Élémentaires*, 2nd ser., **2** (1883), 257–261.
18. F. V. Fomin, D. Lokshtanov, S. Saurabh and M. Zehavi, *Kernelization: Theory of Parameterized Preprocessing*, Cambridge Univ. Press, 2019.
19. F. V. Fomin and D. Kratsch, *Exact Exponential Algorithms*, Springer, 2010.
20. F. Gavril, The intersection graphs of subtrees in trees are exactly the chordal graphs, *J. Combin. Theory* (*B*) **16** (1974), 47–56.
21. P. Gawrychowski, H. Kaplan, S. Mozes, M. Sharir and O. Weimann, Voronoi diagrams on planar graphs, and computing the diameter in deterministic $\tilde{O}(n^{5/3})$ time, *Proc. 29th ACM–SIAM Symp. on Discrete Algorithms* (2020), 495–514.
22. P. Gawrychowski, S. Mozes and O. Weimann, Minimum cut in $O(m \log^2 n)$ time, *Proc. 47th Internat. Colloq. on Automata, Languages and Programming (ICALP 2020)*, to appear.
23. M. C. Golumbic, Reasoning about time, *Mathematical Aspects of Artificial Intelligence* (ed. F. Hoffman), Proc. Symp. Applied Math. **55** (1998), 19–53.
24. M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs* (2nd edn.), Elsevier, 2004.
25. M. C. Golumbic, Tolerance graphs, *Handbook of Graph Theory* (2nd edn.) (eds. J. L. Gross, J. Yellen and P. Zhang), CRC Press (2013), 1105–1120.
26. M. C. Golumbic and R. E. Jamison, The edge intersection graphs of paths in a tree, *J. Combin. Theory* (*B*) **38** (1985), 8–22.
27. M. C. Golumbic and R. E. Jamison, Edge and vertex intersection of paths in a tree, *Discrete Math.* **55** (1985), 151–159.
28. M. C. Golumbic and R. E. Jamison, Rank-tolerance graph classes, *J. Graph Theory* **52** (2006), 317–340.

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

29. M. C. Golumbic, R. E. Jamison and A. N. Trenk, Archimedean ϕ-tolerance graphs, *J. Graph Theory* **41** (2002), 179–194.
30. M. C. Golumbic, M. Lipshteyn and M. Stern, The k-edge intersection graphs of paths in a tree, *Discrete Appl. Math.* **156** (2008), 451–461.
31. M. C. Golumbic, M. Lipshteyn and M. Stern, Representing edge intersection graphs of paths on degree 4 trees, *Discrete Math.* **308** (2008), 1381–1387.
32. M. C. Golumbic, M. Lipshteyn and M. Stern, Equivalences and the complete hierarchy of intersection graphs of paths in a tree, *Discrete Appl. Math.* **156** (2008), 3203–3215.
33. M. C. Golumbic, M. Lipshteyn and M. Stern, Intersection models of weakly chordal graphs, *Discrete Appl. Math.* **157** (2009), 2031–2047.
34. M. C. Golumbic, M. Lipshteyn and M. Stern, Edge intersection graphs of single bend paths on a grid, *Networks* **54** (2009), 130–138.
35. M. C. Golumbic and C. L. Monma, A generalization of interval graphs with tolerances, *Congressus Numer.* **35** (1982), 321–331.
36. M. C. Golumbic, C. L. Monma and W. T. Trotter, Tolerance graphs, *Discrete Appl. Math.* **9** (1984), 157–170.
37. M. C. Golumbic and G. Morgenstern, Edge intersection graphs of paths in a grid, *50 Years of Combinatorics, Graph Theory, and Computing*, CRC Press (2019), 193–209.
38. M. C. Golumbic and A. Siani, Coloring algorithms for tolerance graphs: reasoning and scheduling with interval constraints, *Lecture Notes in Computer Science* **2385**, Springer (2002), 196–207.
39. M. C. Golumbic and A. N. Trenk, *Tolerance Graphs*, Cambridge Univ. Press, 2004.
40. D. Gonçalves, L. Isenmann and C. Pennarun, Planar graphs as L-intersection or L-contact graphs, *Proc. 29th ACM-SIAM Symp. Discrete Algorithms* (2018), 172–184.
41. R. L. Graham and P. Hell, On the history of the minimum spanning tree problem, *Ann. Hist. Comput.* **7** (1985), 43–57.
42. R. B. Hayward, P. E. Kearney and A. Malton, NeST graphs, *Discrete Appl. Math.* **121** (2002), 139–153.
43. R. B. Hayward and R. Shamir, A note on tolerance graph recognition, *Discrete Appl. Math.* **143** (2004), 307–311.
44. D. Heldt, K. Knauer and T. Ueckerdt, Edge-intersection graphs of grid paths: the bend-number, *Discrete Appl. Math.* **167** (2014), 144–162.
45. D. Heldt, K. Knauer and T. Ueckerdt, On the bend-number of planar and outerplanar graphs, *Discrete Appl. Math.* **179** (2014), 109–119.
46. T. Husfeldt, Graph colouring algorithms, *Topics in Chromatic Graph Theory* (eds. L. W. Beineke and R. J. Wilson), Cambridge Univ. Press (2015), 277–303.
47. ISGCI: Information system on graph classes and their inclusions, *www.graphclasses.org/*.
48. R. E. Jamison and H. M. Mulder, Tolerance intersection graphs on binary trees with constant tolerance 3, *Discrete Math.* **215** (2000), 115–131.
49. R. E. Jamison and H. M. Mulder, Constant tolerance intersection graphs of subtrees of a tree, *Discrete Math.* **290** (2005), 27–46.
50. P. Klein and S. Mozes, Optimization algorithms for planar graphs, to appear, *www.planarity.org/*.
51. R. M. R. Lewis *A Guide to Graph Colouring: Algorithms and Applications*, Springer, 2015.
52. E. Lucas, Récréations Mathématiques, I, II, III, IV, Paris, 1882–1894.
53. N. V. R. Mahedev and U. N. Peled, *Threshold Graphs and Related Topics*, North-Holland, 1995.
54. T. A. McKee and F. R. McMorris, *Topics in Intersection Graph Theory*, SIAM Monographs on Discrete Mathematics and Applications, 1999.

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press

55. F. R. McMorris and D. R. Shier, Representing chordal graphs on $K_{1,n}$, *Comment. Math. Univ. Carolin.* **24** (1983), 489–494.
56. G. B. Mertzios, I. Sau and S. Zaks, A new intersection model and improved algorithms for tolerance graphs, *SIAM J. Discrete Math.* **23** (2009), 1800–1813.
57. G. B. Mertzios, I. Sau and S. Zaks, The recognition of tolerance and bounded tolerance graphs, *SIAM J. Comput.* **40** (2010), 1234–1257.
58. S. Mozes, C. Nikolaev, Y. Nussbaum and O. Weimann, Minimum cut of directed planar graphs in $O(n \log\log n)$ time, *Proc. 29th ACM-SIAM Symp. Discrete Algorithms* (2018), 477–494.
59. S. Mozes, Y. Nussbaum and O. Weimann, Faster shortest paths in dense distance graphs, with applications, *Theor. Comput. Sci.* **711** (2018), 11–35.
60. G. Narasimhan and R. Manber, Stability number and chromatic number of tolerance graphs, *Discrete Appl. Math.* **36** (1981) 47–56.
61. J. Nešetřil, E. Milková and H. Nešetřilová, Otakar Borůvka on minimum spanning tree problem: translation of both the 1926 papers, comments, history, *Discrete Math.* **233** (2001), 3–36.
62. F. S. Roberts, *Discrete Mathematical Models, with Applications to Social, Biological and Environmental Problems,* Prentice-Hall, 1976.
63. F. S. Roberts, *Graph Theory and its Applications to Problems of Society*, Society for Industrial and Applied Mathematics, 1978.
64. R. Sedgewick and K. Wayne, *Algorithms* (4th edn.), Addison-Wesley, 2011.
65. J. P. Spinrad, *Efficient Graph Representations*, Fields Institute Monographs, Amer. Math. Soc., 2003.
66. G. Tarry, Le problème des labyrinthes, *Nouvelles Annales de Math.* **14** (1895), 187.
67. J. R. Walter, Representations of chordal graphs as subtrees of a tree, *J. Graph Theory* **2** (1978), 265–267.
68. O. Weimann and R. Yuster, Approximating the diameter of planar graphs in near linear time, *ACM Trans. Algorithms* **12** (2016), 12:1–12:13.

https://doi.org/10.1017/9781108592376.004 Published online by Cambridge University Press