15. Fundamental Graph Algorithms

Graphs are absolutely pervasive in computer science and discrete mathematics. They provide an abstract framework for modeling a vast number of theoretical and practical algorithmic problems. We encounter many different types of graphs on a daily basis: every time we walk through a city or travel by car, bus, or airplane we are navigating a graph known as a "transportation network", where nodes indicate specific locations (e.g., intersections, bus stops, or cities) and edges describe the connectivity between these locations. Every time we place a phone call or visit a web page we are utilizing a graph known as a "communication network" where nodes represent switches and routers and edges indicate wires. The set of people with whom we interact, together with their acquaintances (and their acquaintances' acquaintances, etc.) forms a graph called a "social network", where nodes are people and edges correspond to acquaintances. Even as you read this book, you are navigating a directed acyclic graph (DAG) in which nodes correspond to chapters and directed edges indicate chapter dependencies.

Graph algorithms make up the largest section of this book, filling six rather packed chapters. In this chapter we introduce fundamental graph problems, algorithms, and data structures, and in the following five chapters we discuss some of the most prominent topics in the study of graph algorithms:

- **Shortest Paths (Chapter 16).** If edges have associated costs, what are the least expensive paths between certain pairs of nodes in a graph?
- **Minimum Spanning Trees (Chapter 17).** If edges have associated costs, what is the least expensive set of edges that connects together all the nodes of a graph?
- **Network Flows (Chapter 18).** If edges have associated capacities, what is the maximum amount of some commodity one can route between two designated nodes s and t? If edges have capacities and also costs, what is the least expensive way to route a certain prescribed amount of some commodity through a graph?
- **Cuts, Connectivity, and Clustering (Chapter 19).** What is the least expensive set of edges whose removal separates a graph into two or more pieces, or that separates a designated set of nodes from each-other? How

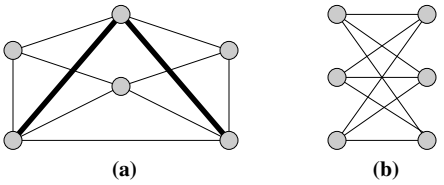

FIGURE 15.1: Graph (a) is planar (just redraw the two bold edges so they curve around "outside" the rest of the graph) but the graph shown in (b) is not: it cannot be drawn in the plane without some pair of edges crossing. As we see in (a), a careful distinction should be made between a "graph" and an "embedding of a graph", since a graph with a particular combinatorial description (i.e., which pairs of nodes are connected by edges) can be embedded in space in many ways.

"well-connected" are certain pairs of nodes in a graph? How can one partition the nodes of a graph into k "well-connected" clusters?

- **Matchings (Chapter 20)**. If the presence of an edge $\{i, j\}$ indicates a valid pairing between i and j, what is the maximum number of nodes that can be paired together (with each node paired with at most one other node)? If the cost of an edge $\{i, j\}$ indicates the cost of pairing up nodes i and j, what is the least expensive way to pair up the nodes of a graph?

The notion of a graph should be reasonably familiar at this point, as we have already studied several simple examples of graph problems and algorithms in preceding chapters. During our study of greedy algorithms we introduced the minimum spanning tree problem and several different variants of matching problems, and the (acyclic) shortest path problem served as our principle example for illustrating the technique of dynamic programming. In the chapters that follow, we will revisit and extend many of these examples, drawing upon the techniques and tools studied in previous chapters, such as optimization techniques (greedy algorithms, dynamic programming, and sometimes even linear programming), as well as simple data structures (priority queues, disjoint sets, search trees, etc.).

It is nearly impossible to study graph algorithms without an occasional detour into the realm of *graph theory*, an active branch of mathematics devoted to studying properties of certain classes of graphs, as well as characterizing the classes of graphs having certain properties. Many types of problems fall under the domain of graph theory. Some particularly notable examples are problems involving *embeddability* of graphs on surfaces. For example, the graph in Figure 15.1(a) is *planar* (i.e., it can be drawn in the plane without any edges crossing), while the graph shown in Figure 15.1(b) is non-planar but can be drawn on the surface of a torus (a donut) without crossing edges. [How?] In Chapter 22, we will study planar graphs and other graphs with nice geometric characteristics and show how many problems can be solved very efficiently on such graphs. Another prominent class of problems investigated by graph theorists is that of *coloring* problems, which ask us to color the nodes of a graph (typically with as few colors as possible) so that no two adjacent nodes receive the same color. We will study these problems in further detail in a few pages in Section 15.5. Since this is an algorithms text rather than

a book on graph theory, we will tend to focus on graph theoretic issues from an algorithmic perspective. Where a graph theorist might be interested in proving that every planar graph can be colored with four colors (a rather famous result in graph theory), we would instead be more interested in knowing that one can compute a 4-coloring of any planar graph in $O(n^2)$ time.

We begin our chapter with a discussion of two absolutely fundamental techniques for searching a graph: breadth-first and depth-first search. Building on these techniques, we discuss efficient algorithmic solutions for a host of simple graph problems: finding connected components, locating bridges and articulation nodes, computing Eulerian paths and cycles, testing for 2-colorability, topological sorting, and many more. We then turn our attention to a host of common packing, covering, and partitioning (coloring) problems in graphs, and finally we discuss special classes of graphs (e.g., trees, intersection graphs, chordal graphs) to which more efficient algorithmic techniques can often be applied.

We assume familiarity with the fundamental terminology of graphs as well as the use of either an adjacency matrix or adjacency lists to represent a graph in memory; these topics are discussed in Section 2.1.4. Unless otherwise stated, we assume all of our graphs are stored using adjacency lists.

15.1 Searching a Graph

Some of the simplest algorithmic questions one can ask of a graph are *connectivity* and *reachability* questions like "is there a path from node i to node j?", or "what is the set of all nodes reachable along any path from node i?". In this section we solve these problems by introducing two fundamental and extremely important algorithms for searching a graph known as *breadth-first search* (BFS) and *depth-first search* (DFS). While simple and fairly intuitive, these algorithms are quite powerful — over the next few pages we shall see how they can help us efficiently solve a wide variety of fundamental graph problems.

Breadth-first and depth-first search apply to both directed and undirected graphs. We will discuss them from the perspective of a directed graph, assuming that in an undirected graph every edge $\{i, j\}$ is represented by a pair of oppositely-oriented directed edges (i, j) and (j, i). Many graph algorithms apply equally well to both undirected and directed graphs, if we translate undirected edges into directed edges in this fashion.

15.1.1 Breadth-First Search

Breadth-first search (BFS) visits the nodes of a graph in a "layer by layer" fashion in order of their distance from a given source node s. Initially the algorithm visits all the neighbors of s, then the neighbors of the neighbors of s, and so on. To accomplish this, it maintains a queue of nodes yet to visit (initially just the source node s). Every iteration of the algorithm entails visiting the next node in this queue, and enqueuing any of its neighbors not yet visited. When it first visits the source, the nodes one hop away from the source are queued up; when these are visited next, all nodes two hops away from the source are being queued up, and so


```
BFS(s):
 1.  For all nodes i:                                  Initialize:
 2.     pred[i] ← null                                    dist[i] = +∞ means
 3.     dist[i] ← +∞                                      i not yet visited
 4.  Q ← {s}                                           Q stored as a queue
 5.  dist[s] ← 0
 6.  While Q is nonempty:                              While nodes left to visit
 7.     i ← next node in Q
 8.     For all nodes j such that (i, j) is an edge:
 9.       If dist[j] = +∞,                            Don't visit nodes twice
10.         pred[j] ← i                                Add (i, j) to BFS tree
11.         dist[j] ← dist[i] + 1                      Compute s ⇝ j distance
12.         Append j to the end of Q
```

FIGURE 15.2: Pseudocode for breadth-first search (BFS). This implementation searches outward from a single source node s, and computes the shortest path distance (in terms of number of edges) from s to every node reachable from s. A suitable value to use for $+\infty$ here is n, since the maximum possible distance any node can lie from the source is $n - 1$.

on. Pseudocode for BFS is shown in Figure 15.2.

Due to the manner in which BFS visits nodes in successive levels of distance from the source, BFS not only finds all nodes reachable from s but also a shortest path from s to each of these nodes, where by "shortest" we mean a path containing the smallest number of edges. These paths are succinctly encoded in the output of the algorithm in terms of a tree directed out of s, known as a *breadth-first search tree*. The tree is specified by the algorithm by setting for each node $j \neq s$ its *predecessor* (or *parent*) in the tree, denoted pred[j] in the pseudocode above. To find the shortest path through the BFS tree from s to any node j, we simply start at j and walk backward along these predecessor links until we reach s — this process traces out the shortest path in reverse, in exactly the same was as we follow "backpointers" to reconstruct the solution of a dynamic programming problem. [Animated example of BFS and a BFS tree].

A breadth-first search requires $O(n + m)$ time to perform, since we first initialize all nodes to "not visited" (by setting their distance labels to $+\infty$), and since we examine every edge in the graph at most once. For graph problems, an $O(n + m)$ algorithm is a called a "linear-time" algorithm, since its running time is linear in the size of the graph.

15.1.2 Depth-First Search

Depth-first search (DFS) is the algorithm we might instinctively use if we were trapped in a maze and looking for an exit. Thinking of the maze as a graph, we start from some source node s and follow an arbitrary path through the maze, carefully marking where we have been so we do not inadvertently revisit the same node twice. Once we reach a dead end (a node from which all outgoing edges lead to nodes we have already visited), we backtrack and attempt to branch off in a different


```
Visit(i):
1.  status[i] ← ``pending''
2.  Increment t, d[i] ← t                          Record "discovery time" of i
3.  For all nodes j such that (i,j) is an edge:
4.     If status[j] = ``unvisited'',               Don't visit nodes twice
5.       pred[j] ← i                               Add (i,j) to DFS tree
6.       Visit(j)
7.  status[i] ← ``finished''
8.  Increment t, f[i] ← t                          Record "finishing time" of i

Full-DFS:
1.  t ← 0                                          t maintains current "time"
2.  For all nodes i:                               Initialize nodes
3.     status[i] ← ``unvisited''
4.     pred[i] ← null
5.  For all nodes i:                               Search entire graph
6.     If status[i] = ``unvisited'', Visit(i)
```

FIGURE 15.3: Pseudocode for a "full" depth-first search (DFS). This version of DFS searches the entire graph by repeatedly locating a yet-unvisited source node and searching from it. It also records the "discovery time", $d[i]$, and "finishing time", $f[i]$, for every node i.

direction. Stated differently, if we are currently at node i, we arbitrarily select an outgoing edges (i, j) and recursively visit j and all nodes reachable from j (except for nodes we've already visited), after which we return to i and continue following the remainder of i's outgoing edges. It should be clear now why this algorithm is named "depth-first" search, since it dives into a graph as deeply as possible before backing up and searching for alternate routes. If we let $R(i)$ denote the set of all nodes reachable from node i, then the DFS process ensures that all of $R(i)$ has been visited by the time we finally backtrack out of i.

A common variant of DFS searches not only from one starting location s, but from a multitude of starting locations until the entire graph has been explored: we first pick an arbitrary starting node s and perform a DFS to explore all nodes in $R(s)$. After that, we continue scanning through the nodes of our graph and if we reach one that has not yet been visited we launch yet another DFS from that node, and so on. For lack of a better name, we call this approach a *full DFS*, and it is illustrated in pseudocode in Figure 15.3. This pseudocode utilizes a recursive *Visit* routine, although there is another popular "iterative" implementation that looks just like BFS except it stores the set of nodes yet to be visited in a stack rather than a queue [Details]. Both DFS from a single starting node and full DFS take $O(m+n)$ time, since only a constant amount of work is spent for each node and each edge. [Animated example of full DFS]

Running Time Subtleties. Both BFS and DFS can run in $\Theta(m+n)$ time in the worst case: $\Theta(m)$ since they might examine every edge exactly once, and $\Theta(n)$ since they initially mark every node as being "unvisited". Often one finds the running time of linear graph algorithms like BFS and DFS written simply as $O(m)$, either out of sloppiness, or due to the rather common assumption that the graph we are

searching is connected, so $m \geq n - 1$ (and hence the $O(m)$ term is dominant). However, if we apply the virtual initialization trick from problem 2 to initialize all nodes to "unvisited" in $O(1)$ time, we can rightly claim a simpler $O(m)$ running time for BFS and DFS (as well as many other similar linear-time graph algorithms), without being sloppy or making unnecessary assumptions. Note that this approach is rarely used in practice, so it mainly serves to give us a theoretical justification for writing $O(m)$ instead of $O(m + n)$. In this book, we henceforth make the assumption when dealing with graph problems that our input graph is connected, thereby simplifying the description of our running times by eliminating this issue from consideration. Without this assumption, we would need to write the running time of many "linear time" graph algorithms as $O(m + n)$, instead of just $O(m)$.

15.1.3 Cycle Detection

Rather than just marking a node as visited or unvisited, the state of a node i during a DFS can be one of three values: *unvisited* (not yet discovered by the algorithm), *pending* (the algorithm has visited i and is currently in the process of exploring $R(i)$), *finished* (DFS has completely finished visiting i and $R(i)$). This extra information can be quite useful. For example, if during a DFS of a directed graph we attempt to follow an edge that leads us to a pending node, this means we have discovered a directed cycle in our graph. Therefore, DFS gives us an $O(m)$ algorithm for detecting whether a directed graph contains a directed cycle [Further details]. In an undirected graph, we can actual detect the presence of any cycles in only $O(n)$ time (a rare running time for a graph algorithm, since it means we don't have time to look at all the edges in a graph). Here, each edge we examine either takes us to an unvisited node (which can happen at most $n-1$ times in total), or to a node we have already visited, in which case we have found a cycle. Another way to view this result is to say that if an undirected graph contains more than $n - 1$ edges, then it *must* contain a cycle (recall that a tree on n nodes contains exactly $n - 1$ edges, so any additional edges must create cycles). For a graph with $m \geq n$, we can therefore restrict our attention to any set of $m' = n$ edges and run DFS to discover such a cycle in $O(m') = O(n)$ time.

15.1.4 Bridges and Articulation Nodes

When performing a full DFS we often find it useful to compute two additional values for every node i: its *discovery time*, $d[i]$, and its *finishing time*, $f[i]$. Discovery time is just a timestamp specifying when the algorithm first visits i, and finishing time is another timestamp specifying when the algorithm finally finishes visiting i (and therefore also all of $R(i)$). These two additional pieces of information can help us solve quite a few simple problems involving graph structure. For example, suppose we wish to determine whether an undirected graph contains a *bridge* (an edge whose deletion would separate the graph into two pieces) or an *articulation node* (a node whose deletion would separate the graph into multiple pieces). We might be interested if certain graphs, for example communication networks, contain such features since this would make them vulnerable to the deletion or malfunction of only a single edge or node. By carefully utilizing discovery and finishing times, we can find bridges and articulation nodes in $O(m)$ time using DFS. [Details]

15.1.5 Topological Sorting Revisited

Recall from Section 3.8 the process of *topologically sorting* a directed acyclic graph (DAG). For any DAG, we can construct a linear ordering of its nodes, known as a *topological ordering*, in which all directed edges point from left to right. In Section 3.8 we investigated a simple $O(m+n)$ topological sorting algorithm. DFS gives us another simple algorithm: perform a full DFS of our graph, and then output the nodes in reverse order of their finishing times. This can be combined with our DFS-based cycle-detection algorithm to give a linear-time algorithm that either produces a topological ordering of the nodes of a directed graph, or concludes the graph is not acyclic by finding a directed cycle. It is relatively easy to prove that we obtain a valid topological ordering of an acyclic graph when we order its nodes in reverse order of finishing time. [Simple proof]

15.2 Eulerian and Hamiltonian Graphs

An *Eulerian* path or cycle in a graph is a path or cycle that visits every edge in the graph exactly once. Eulerian paths and cycles are well-defined in both directed and undirected graphs, and a graph having an Eulerian cycle is called an *Eulerian graph*.

As we remarked in the first chapter of this book, the great mathematician Leonard Euler (after whom these graphs are named) was the first to characterize the conditions under which a graph is Eulerian. An undirected graph is Eulerian if and only if it is connected and all of its nodes have even degree, and a directed graph is Eulerian if and only if it is strongly connected (that is, if every node can reach every other node via a directed path), and each node has the same number of incoming as outgoing edges. A similar characterization holds for Eulerian paths, in which case we require that all but two nodes have even degree and two nodes have odd degree (these two nodes will be the endpoints of the Eulerian path). In a directed graph, an Eulerian path exists from node i to node j if the outdegree of i is one larger than its indegree, the outdegree of j is one less than its indegree, and all other nodes have equal indegree and outdegree. [Short easy proofs of these facts]

Problem 248 (Finding Eulerian Paths and Cycles). Building on depth-first search, give a linear-time algorithm that finds an Eulerian cycle (or path, if conditions are appropriate) in an Eulerian graph. You may want to use ideas from the constructive proof above as a starting point. Make sure your methods apply to both directed and undirected graphs. [Solution]

Problem 249 (DNA Sequencing by Hybridization). The problem of *sequencing* an unknown strand of DNA is one of the most important problems in bioinformatics. Effective solutions to this problem have, for example, enabled researchers to determine the complete DNA sequence for the human genome. In this problem, we show how Eulerian path computation plays an important role in a method for DNA sequence determination known as *sequencing by hybridization*. For our purposes, our unknown strand of DNA is just a string comprised of the characters A, C, G, and T. Using a *DNA microarray*, we can experimentally determine all of the length-k substrings present in our DNA string. For example, if our DNA string is 'ATTGCATA' and $k = 4$, then the microarray experiment will tell us that 'ATTG', 'CATA', 'GCAT', 'TGCA', and 'TTGC' are all present in our

DNA string, but it unfortunately does not tell us their relative ordering. See if you can use an Eulerian path algorithm to determine a feasible ordering of these substrings (where substring y can follow substring x only if the first $k-1$ characters in y are the same as the last $k-1$ characters in x). For simplicity, you can assume that each length-k substring in our input appears only once in the target DNA strand we are trying to sequence[1]. [Solution]

A similar problem to the Eulerian path/cycle problem is that of computing a *Hamiltonian path/cycle*, which visits every *node* in a graph exactly once. A graph or directed graph is said to be *Hamiltonian* if it admits a Hamiltonian cycle. In stark contrast to the Eulerian path/cycle problem, the problems of finding Hamiltonian paths and cycles are both NP-hard. In Chapter 17 we will study simple approximation algorithms for a famous relative of the Hamiltonian cycle problem known as the *traveling salesman problem* in which edges have associated costs and we want to find a minimum-cost Hamiltonian cycle.

Yet another related problem is the *Chinese postman problem*, in which we want to find a minimum-edge or minimum-cost tour of a graph visiting every edge *at least once*. In problems 329 and 385 we will see how to solve this problem efficiently using network flow and matching algorithms. If we want to visit every node at least once, rather than every edge, the problem again becomes NP-hard, just like the traveling salesman problem. All of these problems have obvious applications in practice along the lines of planning "delivery routes" that visit either every node or every edge in a transportation network.

Problem 250 (Gray Codes and de Bruijn Sequences). An n-bit *gray code* (named the 20th century physicist Frank Gray) is a list of length 2^n containing all n-bit binary numbers, such that adjacent numbers in the list (including the first and the last) differ in only a single digit. For example, a 3-bit gray code is 000, 001, 011, 010, 110, 111, 101, 100. An n-bit *de Bruijn* sequence (named after the 20th century mathematician Nicolaas Govert de Bruijn) is a 2^n-bit circular binary sequence (a sequence that is taken to "wrap around" from the end back to the beginning) that contains every n-bit binary number as a substring. For example, the circular sequence 00010111 contains every 3-bit binary sequence as a substring. Please give a simple proof of existence for both of these sequences, and also show how to compute them each in $O(2^n)$ time (i.e., linear time in the length of the output). For the gray code, your algorithm should specify each successive number concisely by giving the index of the single changed bit. How are gray codes and de Bruijn sequences related to Hamiltonian and Eulerian cycles, respectively? [Solution]

Problem 251 (Hamiltonian Paths and Cycles in Tournaments). Problems involving finding longest paths and cycles in a (directed) graph tend to be NP-hard, since they could be used to determine whether a graph contains a Hamiltonian path or cycle. Moreover, even if it is known that a graph is Hamiltonian, it still seems very difficult to find long paths and cycles even though these certainly should exist. A notable exception to this statement is special case of *tournaments*. A tournament is a directed graph in which between every pair of nodes i and j there is either a directed edge (i, j) or a directed edge (j, i), but not both. We can think of the n nodes of a tournament as teams in a sporting tournament, in which every team plays against every other team and the direction of the

[1] A drawback of using DNA microarrays is that they will occasionally give false positive indications for substrings not present in our DNA strand, and they also often cannot tell how many occurrences of a re-occurring substring are present. For these reasons and others, we find that in practice other techniques for DNA sequencing are usually preferred over this method of sequencing by hybridization.

edge from i to j indicates the winner when i plays j.

(a) Use induction to prove that every tournament has a Hamiltonian path. [Solution]

(b) Turn your proof from part (a) into an algorithm for finding a Hamiltonian path in a given tournament. While $O(n^2)$ time is acceptable, see if you can achieve a running time of only $O(n \log n)$ using appropriate data structures. Why is it not possible to achieve a better running time than $O(n \log n)$? [Solution]

(c) Try to prove that every strongly-connected tournament contains a Hamiltonian cycle. [Solution]

(d) Turn your proof from part (c) into an $O(n^3)$ algorithm for finding a Hamiltonian cycle in a strongly-connected tournament. For a challenge, see if you can improve the running time to $O(n^2)$. As a hint, first compute a Hamiltonian path and then attempt to transform it into a Hamiltonian cycle. [Solution]

15.3 Connectivity and Connected Components

The *connected components* of an undirected graph are exactly what their name implies they should be: groups of nodes within which all nodes are reachable from each-other. The nodes of any undirected graph can be easily partitioned into connected components in linear time via a straightforward invocation of full DFS. Every time the algorithm searches outward from a source node s it will discover all nodes within s's connected component, and afterward the algorithm moves on to discover any remaining connected components that might exist.

We say a connected graph is *biconnected* (also called *2-connected*) if every pair of nodes is connected by a pair of paths that are node-disjoint except at their endpoints. Similarly, a graph is *triconnected* (also called *3-connected*) if every pair of nodes is connected by three node-disjoint paths, and more generally it is k-connected if it has more than k nodes, and every pair of nodes is connected by k node-disjoint paths. We denote by $\kappa(G)$ the *node connectivity* of a graph G — the largest value of k for which G is k-connected. As we shall see in Chapter 18, *Menger's theorem* tells us that a graph is k-node-connected if and only if at least k nodes must be removed to break the graph into multiple disconnected components. Hence, a graph is biconnected if it has no articulation node, so we can test in linear time using depth-first search if a graph is biconnected.

Connectivity is also commonly defined with respect to edges instead of nodes. A graph is *2-edge-connected* if every pair of nodes is connected by a pair of edge-disjoint paths, and more generally, a graph is k-edge-connected if every pair of nodes is connected by k edge-disjoint paths. Menger's theorem also tells us that a graph is k-edge-connected if we must remove at least k edges to break the graph into multiple disconnected components. Therefore, a graph is 2-edge-connected if it has no bridge, so we can test 2-edge-connectivity in linear time using depth-first search. We denote by $\lambda(G)$ the *edge connectivity* of graph G — the largest value of k for which G is k-edge-connected. It is easy to show that $\lambda(G) \geq \kappa(G)$ for any graph G.

A graph that is not itself k-edge-connected may be partitioned into its k-edge-connected components — maximal sets of nodes that are internally k-edge-connected. In fact, as shown in Figure 15.4(a), the k-edge-connected components of a graph,

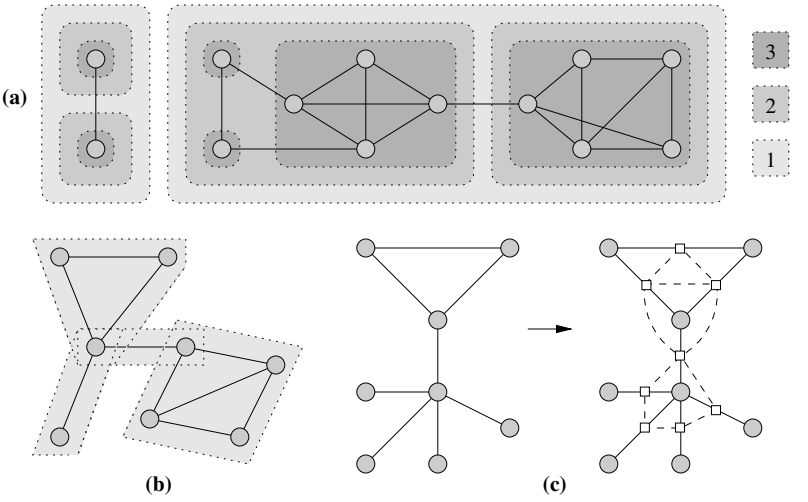

FIGURE 15.4: The nested structure of the k-edge-connected components of a graph (up through $k = 3$) is shown in (a). The numbers corresponding to each region indicate connectivity; for example, 1 indicates a (singly) connected component, 2 a 2-edge-connected component, and so on. In (b), we see the unique decomposition of a graph into its four biconnected components, with each edge belonging to a single component even though articulation nodes might be shared between components (here, we define a single edge as a valid biconnected component, even though 2-connectivity usually only applies to a graph of 3 or more nodes), and (c) shows the reduction from 3-edge-connectivity to 3-connectivity used in problem 253.

for all values of k, give us a rather elegant hierarchical decomposition of the graph: each connected component can be partitioned into 2-edge-connected components, which can each be further partitioned into 3-edge-connected components, and so on. We will develop much more insight into this hierarchical structure when we study Gomory-Hu trees in Chapter 19.

We can also consider, for the case of node connectivity, the decomposition of a graph into k-connected components, within each of which all pairs of nodes are connected by k node-disjoint paths. However, this is slightly more cumbersome, since two k-connected components of a graph may share up to $k - 1$ nodes in common (they cannot share k nodes in common, or we could merge them into a single k-connected component). The special case of biconnected components is fairly easy to handle — one can show that the biconnected components of a graph can be uniquely defined as maximal subsets of nodes within which all pairs of nodes are connected by a pair of node-disjoint paths. Although different biconnected components may share articulation nodes in common at their boundaries, as shown in Figure 15.4(b), one can show that the biconnected components partition the *edges* of our graph, so that every edge belongs to exactly one biconnected component. It is somewhat more complicated (although possible) to define k-connected components for $k \geq 3$ in a way that leads to a *unique* set of k-connected components for any graph. A simple way to arrive at a non-unique configuration of k-connected components is to delete

any *disconnecting set* of $k-1$ or fewer nodes that breaks the graph into multiple components, and to repeat this process recursively on the components that remain until all remaining components are k-connected. We refer the reader to the endnotes for further references on k-connected components for $k \geq 3$.

The k-connected and k-edge-connected components are interesting features of graphs since they measure the extent to which certain regions of a graph are "well-connected". We often want to design graphs, such as communication networks, with a certain amount of built-in "fault tolerance", and a k-(edge)-connected component is resistant to $k-1$ node (edge) failures. In the next two problems, we see how to decompose an arbitrary graph into its biconnected, triconnected, 2-edge-connected, and 3-edge-connected components in linear time. Our discussion of connectivity then continues in Chapter 19, where we focus on higher orders of connectivity.

Problem 252 (Decomposing a Graph into Biconnected Components). It is relatively trivial to partition a graph into its 2-edge-connected components, since we can do this by removing every bridge (and we can find all bridges in linear time using depth-first search). Please try to devise a linear-time algorithm for partitioning a graph into biconnected components that is similarly based on depth-first search. [Solution]

Problem 253 (Reducing k-Edge-Connectivity to k-Connectivity). Hopcroft and Tarjan have shown how to test whether a graph is triconnected (and more generally, to determine its triconnected components) in linear time using a highly non-trivial approach based on depth-first search. Due to its complexity, we do not discuss it here; see the endnotes for additional references. In this problem, we show how to leverage this result to test 3-edge-connectivity using a simple reduction to the problem of testing 3-connectivity. As a result, this will give us a linear-time algorithm for testing 3-edge-connectivity of a graph (which can also be used to partition a graph into its 3-edge-connected components in linear time, but we don't focus on that here, since that would require a more in-depth discussion of 3-connected components that we wish to undertake at this time). We note that 3-connectivity is a particularly important property when dealing with planar graphs, which we will discuss later in this Chapter, so the Hopcroft-Tarjan algorithm is especially useful in that domain of study.

(a) As shown in Figure 15.4(c), suppose we subdivide each edge e our graph by adding a node $x(e)$ to the middle of e, and that we connect the nodes $x(e_1) \ldots x(e_k)$ in a cycle for all edges $e_1 \ldots e_k$ incident to an original node in our graph. Please show that the resulting graph is 3-connected if and only if our original graph is 3-edge-connected. [Solution]

(b) Can you generalize this transformation so that it reduces the problem of testing whether a graph is k-edge-connected to the problem of testing whether a graph with $O(kn+m)$ nodes and $O(km)$ edges is k-connected? [Solution]

15.3.1 Strongly-Connected Components in a Directed Graph

In a directed graph, we generalize the notion of a connected component to what is called a *strongly-connected component*, within which there is directed connectivity among all nodes. That is, for every pair of nodes (i, j) in a strongly-connected component, there is a directed path from i to j, as well as a directed path from j to i. Every directed graph can be partitioned into strongly-connected components, but it is perhaps not so obvious how to do this efficiently. Remarkably, we can still

use DFS to perform this task in linear time, but it takes two iterations of full DFS rather than one. We first run full DFS on our graph, then we reverse the direction of the edges in the graph and run full DFS a second time, processing source nodes in decreasing order of their finishing times from the first DFS. The sets of nodes $R(s_1), R(s_2), \ldots, R(s_k)$, reachable from the k source nodes $s_1 \ldots s_k$ used by our second invocation of full DFS give us the strongly-connected components in our graph. This is clearly a linear-time algorithm, but it seems truly mysterious why it correctly performs its desired task! It is easy to argue correctness of this algorithm, however, by establishing a link with topological sorting. [Simple argument]

Problem 254 (The One-Way Street Problem). Suppose we have an undirected graph and we would like to assign directions to its edges such that the entire graph becomes strongly connected (i.e., every node is reachable from every other node via a directed path). This is known as the *one-way street problem* since we can think of taking the streets in a city map and trying to orient them as one-way streets so that it is still possible to travel between any two locations in the city. Prove that the one-way street problem has a solution if and only if our original graph is connected and has no bridge (an edge whose removal disconnects the graph). Can you devise an efficient algorithm that takes a bridgeless graph and computes an appropriate orientation of its edges? [Solution]

Problem 255 (Universally Reachable Nodes). Let us call a node i in a directed graph *universally reachable* if there exists a directed path from every node in the graph to i. Give a linear-time algorithm that identifies all universally reachable nodes in a given directed graph. [Solution]

Problem 256 (Augmenting a Directed Graph to Achieve Strong Connectivity). Given a directed graph, suppose we want to augment it with the minimum possible number of directed edges so that it becomes strongly connected. Please describe a linear-time algorithm for computing the number of edges in an optimal solution, and an $O(mn)$ algorithm for computing the actual set of edges in an optimal solution. [Solution]

Problem 257 (Ear Decompositions). A (directed) graph has an *ear decomposition* if it can be built from a single node to which "ears" are successively added. Adding an ear to a graph involves choosing two nodes x and y in the graph and adding a new (directed) path of one or more edges (including their intermediate nodes) between them.

(a) Show that a graph is 2-connected if and only if it has an ear decomposition. [Solution]

(b) Show that a directed graph is strongly-connected if and only if it has an ear decomposition. Does this (plus the solution of the last part) imply a simple solution for the theorem related to problem 254 stating that a 2-connected graph can always have its edges oriented so that it becomes strongly-connected? [Solution]

(c) Show how to compute ear decompositions for parts (a) and (b) in linear time. As a hint, argue that if a graph is minimally 2-connected or strongly-connected (i.e., removing any edge would break this property), then it must have a node of degree two. [Solution]

Problem 258 (Breaking up Strong Components). If we compress each strongly-connected component in a graph to a single "supernode", it is easy to see that we obtain a DAG. This viewpoint – regarding an arbitrary directed graph as a DAG of strong components – can sometimes be a useful means of effectively treating any directed graph as a DAG. It also leads to an useful hierarchical decomposition of any directed graph, since we can take each of its strong components and further decompose it by "temporarily" pretending one of its edges (i, j) is absent, thereby breaking the component into a sub-DAG of even smaller strongly-connected components, which we can continue to break apart in

a recursive fashion. Sub-DAGs obtained in this fashion have the interesting property that they contain exactly one "source" (a component containing only node j) and exactly one "sink" (a component containing only node i). Moreover, if our original graph was minimally strongly connected (removal of any edge would break strong connectivity), then we can choose an appropriate edge (i, j) to remove such that the resulting sub-DAG is nothing more than a path of strong components connected by directed edges[2]. To practice using this hierarchical decomposition, please give an algorithm that takes as input an arbitrary strongly-connected graph and removes a subset of its edges so that there are no more directed cycles, but for any edge (i, j) removed, there is still a directed path from j to i. [Solution]

15.3.2 Dynamic Connectivity

By computing the connected components of an undirected graph in linear time using DFS and labeling each node with a number identifying the component to which it belongs, we can easily answer subsequent queries of the form "do nodes i and j belong to the same connected component?" in $O(1)$ time by comparing the labels of i and j. However, what if we now insert a new edge to the graph or delete an existing edge? This leads us to a discussion of data structures for the *dynamic connectivity* problem, where we must process edge insertions and deletions as well as connectivity queries.

The incremental case of the dynamic graph connectivity problem is by far the easiest. In this case, we only need to accommodate edge insertions and connectivity queries, and we have already studied an ideal data structure for this task! Using a disjoint set data structure (Section 4.6), we simply maintain each connected component in our graph as a set in our collection of disjoint sets. Initially, each node starts out as its own singleton set. A connectivity query on nodes i and j asks us whether elements i and j belong to the same set (the *find-set* operation), and the insertion of edge $\{i, j\}$ requires that we merge the set containing i with the set containing j (the *union* operation), unless i and j were already in the same set, in which case we do nothing. Using the highly-efficient tree-based data structure for disjoint sets from Section 4.6, all of these operations can be easily implemented in $O(\alpha(n))$ amortized time on an n-node graph. When we later reach Chapter 17, we will see that this approach to incremental dynamic connectivity plays a vital role in Kruskal's famous minimum spanning tree algorithm.

As with most dynamic algorithms, the fully-dynamic case (with both edge insertions and deletions) tends to be significantly harder than the incremental and decremental cases by themselves (in fact, even the decremental case is quite challenging on a general graph). We first note that in a forest, fully-dynamic connectivity is easy to obtain with $O(\log n)$ time per operation by using dynamic tree data structures that support linking and cutting, such as Euler tour trees (Section 8.5.2). Applying this idea in a somewhat sophisticated hierarchical fashion, one can obtain a data structure for fully-dynamic connectivity in a general graph that supports connectivity queries in $O(\log n)$ time and edge insertions and deletions in $O(\log^2 n)$ amortized

[2] In this case, adding the edge (i, j) back in yields a cycle of strong components. By recursively continuing this construction to break down each of these strong components into another cycle of strong components, and so on, we obtain what Knuth calls the "wheels within wheels" decomposition of a directed graph. It is closely related to an ear decomposition (the preceding problem), and you may want to use ear decompositions to help you solve this problem.

time. Although this data structure is rather complicated, the advanced reader is encouraged to listen to its [details].

15.4 Packing and Covering Problems

We have already studied quite a few variants of packing and covering problems in this book, most notably the general set packing and covering problems (Section 10.5.2), which include as special cases most other packing and covering problems we tend to encounter. Many of the most prominent graph problems we find in practice are packing or covering problems:

- **Node Packings (Independent Sets).** A *node packing* (also called an *independent set* or a *stable set* is a subset of the nodes of a graph, no two of which are adjacent. If we construct a graph in which nodes correspond to tasks, and edges join together any pair of tasks that both depend on the same resource (so both cannot be performed at the same time), then a maximum-cardinality node packing tells us the largest possible set of tasks we can work on in parallel.

- **Edge Packings (Matchings).** An *edge packing* (more commonly called a *matching*) is a subset of edges, no two of which share an endpoint. Matchings have numerous applications, and we devote an entire chapter to their study later in the book.

- **k-Packings.** A subset S of nodes is a *k-packing* if the distance between every two nodes in S is larger than k. Hence, a 1-packing is an independent set, a 2-packing is a set of nodes whose neighborhoods are disjoint, and so on.

- **Node Covers.** A *node cover* (also commonly called a *vertex cover*) is a subset of nodes that collectively covers every edge of a graph; that is, every edge must have at least one endpoint in the cover.

- **Edge Covers.** An *edge cover* is a subset of edges that covers every node in a graph. That is, for every node, at least one of its incident edges must belong to the cover.

- **Dominating Sets.** A *dominating set* is a subset S of nodes such that every node not in S is at least adjacent to some node in S. That is, each node in S is capable of covering (i.e., dominating) the nodes in its neighborhood, then we want the set S to cover (dominate) the entire graph. We sometimes also seek a *distance−k* dominating set, where each node is capable of covering all the nodes up to distance k away (so a distance-1 dominating set is a standard dominating set). Node/edge cover and dominating set problems have many applications, the prototypical one being the allocation of a resources (e.g., facilities) throughout a network so that all nodes or edges are served by a nearby facility.

In terms of notation, graph theorists denote by $\alpha_0(G)$ and $\alpha_1(G)$ the minimum cardinalities of node and edge covers in a graph G, by $\beta_0(G)$ and $\beta_1(G)$ the cardinalities

of maximum node and edge packings, by $P_k(G)$ the cardinality of a maximum k-packing, and by $\gamma(G)$ the cardinality of a minimum dominating set. Note that we can define weighted variants of all of these problems; for example, we can associate values with the nodes in a graph and ask for a maximum-value node packing (independent set), or we might assign a cost to every edge and ask for a minimum-cost edge cover.

In terms of complexity, the good news is that edge packing (matching) and edge covering problems can be solved in polynomial time; we will study algorithms for these problems in great detail in Chapter 20 (also recall that we introduced fast approximation algorithms for matching problems back in Chapter 10). The bad news is that all of the other packing and covering problems above are unfortunately NP-hard. We therefore focus on approximation results. Recall that we have already discussed 2-approximation algorithms for the minimum node cover problem (Sections 10.4.2 and 12.5.1). The question of whether one can improve on the approximation guarantee of 2 for this problem has long been one of the most pressing questions in the domain of approximation algorithms. One can show that the k-packing problem for $k \geq 2$ and the distance-k dominating set problem for $k \geq 1$ are more or less equivalent to the general set packing and set cover problems, so the best approximation guarantees we can hope to obtain for them are $\Theta(1/\sqrt{n})$ and $\Theta(\log n)$ respectively, and we can obtain these guarantees using natural greedy algorithms (see Section 10.5.2). [Further details on this equivalence]

The maximum node packing (independent set) problem is truly disheartening in terms of complexity: assuming that $NP \neq ZPP$ (a complexity-theoretic conjecture widely believed to be true), then we cannot even approximate the optimal solution of this problem to within a factor of $n^{1-\varepsilon}$ in polynomial time, for any constant $\varepsilon > 0$. Therefore, the ridiculously simplistic $1/n$-approximation algorithm that chooses a single arbitrary node is in principal the best we can hope to do. Another common problem on graphs is the *maximum clique problem*, where we seek to find a maximum subset of nodes that is a *clique* (a set of nodes in which all pairs are connected by edges). This problem is equivalent to the maximum independent set problem if we take the *complement* of our graph — replacing edges with non-edges and non-edges with edges, so the maximum clique problem also suffers from the same dismal inapproximability results as the maximum independent set problem.

The packing and covering problems above are related in many nice ways. As you might expect, since packing and covering are natural "duals" of each-other, we can obtain several min-max inequalities using duality; for example, in any graph G, linear programming duality immediately tells us that $\beta_0(G) \leq \alpha_1(G)$, $\beta_1(G) \leq \alpha_0(G)$, and $P_2(G) \leq \gamma(G)$. [Why?] These relationships are even satisfied by equality for certain special classes of graphs — the first two for bipartite graphs (this is known as *König's theorem*, discussed in problem 380), and the last for trees (Section 12.5.1).

Problem 259 (The Gallai Identities). The Gallai identities give us another nice set of relationships among several graph properties. The most common such identities state that (i) in any graph G, $\alpha_0(G) + \beta_0(G) = n$, and (ii) in any graph G with no isolated nodes, $\alpha_1(G) + \beta_1(G) = n$. In prose, (i) says that the cardinality of a minimum node cover plus that of a maximum node packing (independent set) equals the total number of nodes in a graph, and (ii) says that the cardinality of a minimum edge cover plus that of a maximum edge packing (matching) equals the total number of nodes in a graph, as long

as there are no isolated nodes. For a challenge, please try to prove these properties. As a hint for the second identity (which is slightly trickier to prove than the first), a minimum edge cover in a graph will always be a collection of "stars" (a star being a tree of depth 1), since if it contained a path of length 3 or longer, then one of the edges on such a path could be safely discarded. [Solution]

15.5 Partitioning/Coloring Problems

A substantial branch of graph theory studies *coloring* problems on graphs. A coloring of an undirected graph is an assignment of colors to nodes (we use integers $1, 2, \ldots$, to denote these colors) such that no two adjacent nodes share the same color. The set of nodes of a particular color is known as a *color class*, and since this set contains no two adjacent nodes it is a node packing (independent set). A coloring is therefore a partition of the nodes of a graph into independent sets.

The *chromatic number* of a graph G, typically denoted $\chi(G)$, is the minimum number of colors required to color the nodes of G. It is in general an NP-hard problem to determine the chromatic number of a graph, although we shall see in a moment that we can easily check if the chromatic number of a graph is 2. Using a variant of depth-first search, we can easily color a graph of maximum degree Δ with at most $\Delta + 1$ colors in linear time [Details]. It seems much harder to compute a coloring with substantially fewer colors, even if it is known that our graph has a low chromatic number!

Coloring problems arise in practice when we wish to schedule a set of activities that all contend for a set of limited resources. For example, let the n nodes of a graph denote n unit-duration activities, and let an edge between two nodes indicate the fact that activities corresponding to the nodes cannot be performed simultaneously due to a resource constraint (e.g., we cannot bake both bread and cake since both of these activities make use of an oven, and we have but one oven). A coloring of this graph partitions the activities into phases (corresponding to color classes) each of whose activities can be performed in parallel. By minimizing the number of colors, we minimize the number of phases required to perform all the activities. We will study further applications of coloring in the context of planar graphs in Chapter 22, when we discuss the famous result that any planar graph (e.g. a map) can be colored with no more than 4 colors.

15.5.1 2-Colorable (Bipartite) Graphs

A 2-colorable graph is known as a *bipartite* graph, since its nodes can be divided into two sets such that all edges connect a node in one set to a node in the other set. We typically draw bipartite graphs as in Figure 15.1(b), with one set of nodes on the left and one on the right. Bipartite graphs arise in many applications, particularly in matching and assignment problems, where we want to pair up elements from one set with elements from another set (e.g., jobs with machines, students with dormitories, etc.). It is not too difficult to see that a graph is bipartite if and only if it contains no odd-length cycles. [Simple justification]

Problem 260 (Testing Bipartiteness). Based on the characterization of bipartite graphs above, design a simple linear-time algorithm based on BFS or DFS that tests whether or not a graph is bipartite. If the graph is bipartite, your algorithm should output a partition of the nodes into two sets; otherwise, your algorithm should demonstrate an odd cycle. [Solution]

Problem 261 (Finding an Even Cycle). In the preceding problem, we learned how to test whether a graph contains an odd cycle in $O(m)$ time. Here, we consider the slightly more complicated problem of testing for the presence of an even cycle. We can do this in linear time as well, for undirected graphs (for directed graphs, it is surprisingly still an open problem if this problem can be solved in polynomial time!). The first goal of this problem is to show that any graph with $m \geq 3n/2$ must contain an even cycle, so if we run our algorithm on an arbitrary subgraph of size $3n/2$ edges, we can obtain a running time of only $O(n)$! Can you describe how to do this? To get started, first compute a BFS tree, and then analyze carefully the edges not belonging to this tree. [Solution]

Problem 262 (Coloring a 3-Colorable Graph). It is NP-hard to determine the exact chromatic number of a graph, if it is 3 or higher (i.e. non-bipartite). Unfortunately, it also seems very difficult to approximate the chromatic number of a graph (see the endnotes for full details) or to color a non-bipartite graph with a reasonably small number of colors even if we know its chromatic number. To give some idea of how difficult this seems, consider the following problem, which is one of the strongest known results along these lines: given a graph known to have chromatic number 3, give an algorithm to color it with only $O(\sqrt{n})$ colors. As a hint, consider separately nodes of high degree and nodes of low degree. [Solution]

15.5.2 Edge Coloring

Although it is more common to encounter problems involving coloring of nodes, one can also consider problems in which we must color the edges of a graph so that no two edges incident to a common node share the same color. Equivalently, an edge coloring partitions the set of edges in a graph into edge packings (matchings), since every node can be adjacent to at most one other node using the edges of a specific color class.

Edge coloring problems also have applications in resource allocation. For example, take a bipartite graph where nodes on the left represent jobs and nodes on the right represent machines, in which an edge from job i to machine j indicates that j must at some point perform one unit of work on i. Of course, no job can be simultaneously processed by multiple machines, nor may any machine simultaneously process two or more jobs. An edge coloring of this bipartite graph partitions the edges into groups which can each be processed in parallel in a single time step with no conflicts. In order to minimize the total number of time steps required, we therefore must compute an edge coloring using the smallest possible number of colors.

The minimum number of colors required to color the edges of a graph is known as the *edge chromatic number* of a graph, and it is clear that at this quantity must be at least Δ, the maximum degree of our graph. Interestingly, as opposed to the case of node coloring, where it is very difficult to approximate the chromatic number of a graph, for the case of edge coloring there is an efficient algorithm that computes a $(\Delta + 1)$-coloring of the edges of any graph [Algorithm details]. In general it is NP-hard to determine whether the edge chromatic number of a graph is Δ or $\Delta + 1$. However, for the particular case of bipartite graphs we again have simpler results:

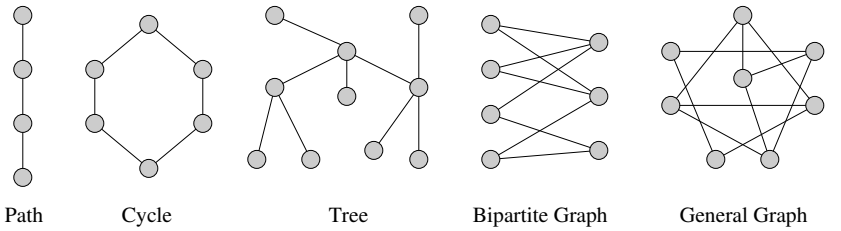

FIGURE 15.5: A progression (from left to right) of successively more complicated classes of graphs. With the exception of the cycle, each class is a special case of the next graph in the sequence.

as we shall see in problem 412 when we study matchings, one can always color the edges of a bipartite graph with Δ colors in nearly linear time.

15.6 Special Classes of Graphs

Graph theorists have studied so many special classes of graphs that one could fill several volumes with a description of all their properties. Many of these special classes of graphs are interesting from an algorithmic perspective as well, since they have structural properties that give us additional leverage for constructing efficient solutions to prominent graph problems. In this section, we briefly recount some of the most commonly-studied graph classes of algorithmic importance. If the reader at any time finds himself or herself suffering from a lack of graph problems, it is easy to construct new problems by taking an existing problem and asking "can I solve this more efficiently on a path? a cycle? a tree? a bipartite graph?", and so on, for as many different graph classes as you desire.

15.6.1 Bipartite Graphs

We have already studied a great deal about bipartite (2-colorable) graphs. According to problem 260, we can recognize a bipartite graph in linear time. Many algorithmic problems are easier in bipartite graphs than in general graphs. For example, when we study matching problems in Chapter 20, we will see that bipartite matching problems are quite a bit simpler to solve than their non-bipartite relatives. More importantly, however, many problems that are NP-hard on general graphs are solvable in polynomial time on bipartite graphs. The most prominent examples here are probably the minimum-cardinality node cover problem and the maximum-cardinality independent set problem, both of which can be solved in polynomial time on a bipartite graph by using a combination of the Gallai identities (problem 259) and König's theorem (problem 380).

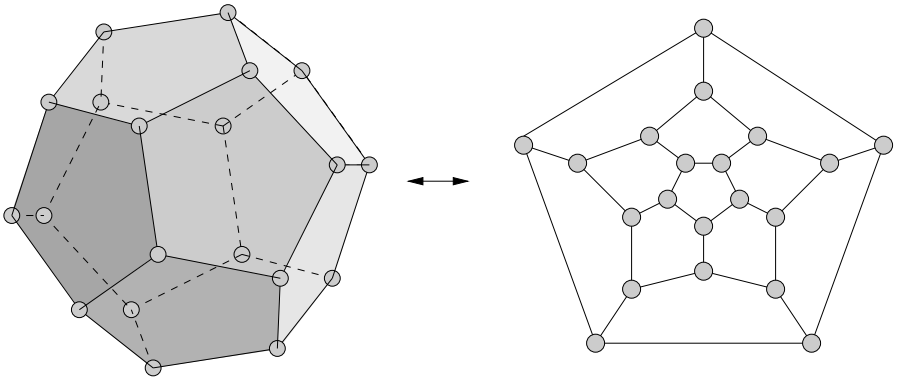

FIGURE 15.6: The dodecahedron as a polyhedron and an equivalent planar graph.

15.6.2 Directed Acyclic Graphs

Many directed graph problems can be solved more efficiently on a directed acyclic graph (DAG), usually by topologically sorting the DAG as a preprocessing step and applying a procedure based on dynamic programming. For example, in Chapter 11 we showed how to solve the single-source shortest path and longest path problems on a DAG in only linear time, whereas in general directed graphs no linear-time algorithm is known for shortest paths, and the longest path problem is actually NP-hard, as we shall see in the next chapter.

15.6.3 Planar Graphs

In Chapter 22, we will undertake a detailed study of graphs that have various geometric properties, the most famous being planar graphs, which can be drawn in the two-dimensional plane with no edges crossing. Many problems can be solved more efficiently in planar graphs, and although many NP-hard problems remain NP-hard on planar graphs, they can often be approximated better; for example, all of the NP-hard graph packing and covering problems we introduced in Section 15.4 have a PTAS on planar graphs. We will discuss these results and other algorithmic tricks for planar graphs once we reach Chapter 22. At this point, we wish to introduce only one important fact about planar graphs that we need to use prior to Chapter 22: every connected planar graph satisfies *Euler's formula*,

$$f + n - m = 2,$$

where f is the number of closed regions (known as *faces*) in a planar embedding of the graph, including the "outer face" surrounding the graph an extending out to infinity. [A simple proof by induction of Euler's formula]

Euler's formula applies to convex polyhedra as well as planar graphs. In fact, from a combinatorial perspective, the "skeleton" of a convex polyhedron is equivalent to a *polyhedral* planar graph — a planar graph that is 3-node-connected (meaning you

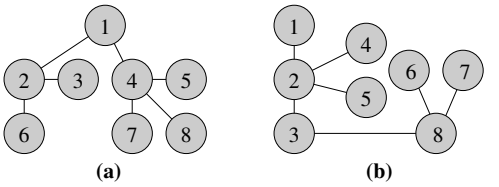

FIGURE 15.7: Two isomorphic trees.

must remove at least 3 nodes in order to break the graph into two separate components). If you take any 3D convex polyhedron and "unfold" it after puncturing one of its faces, *Balinski's theorem* tells us that we get a polyhedral planar graph (where the punctured face becomes the outer face of the graph). More generally, *Steinitz' Theorem* says that any polyhedral planar graph can be folded into a convex polyhedron, so there is a one-to-one correspondence between the two classes of objects. Figure 15.6 shows the dodecahedron, with 20 nodes, 30 edges, and 12 faces, as a polyhedron and as an equivalent planar graph.

Since each face in a planar graph is bordered by at least 3 edges, and since each edge lies at the border of exactly 2 faces, we can write $2m \geq 3f$. By plugging this into Euler's formula and applying a small amount of algebra, we see that $m \leq 3n - 6 = O(n)$. Every planar graph therefore has only $O(n)$ edges, so we can automatically improve the running time of any algorithm applied to a planar graph by replacing m with $O(n)$.

15.6.4 Trees

Of all special graph classes, trees are particularly interesting from an algorithmic perspective. Their unique acyclic structure allows us to apply a large number of powerful algorithmic techniques, such as divide and conquer (via separator decompositions, as shown in Section 8.3.4), primal-dual methods (Section 12.5.1), and dynamic programming (problem 229). Of all these approaches, dynamic programming is perhaps the most powerful, and it gives us linear-time solutions to a wide range of problems that are NP-hard on general graphs, including all of the packing and covering problems listed in Section 15.4. Examples of some of these algorithms appear in problem 229. Note that trees are bipartite and planar, and since they satisfy $m = n - 1$, we can always write $O(n)$ in place of m for the running time of a tree algorithm, just as with planar graphs.

Problem 263 (Tree Isomorphism Testing). Two graphs are said to be *isomorphic* if one can relabel the nodes of one graph to obtain the other graph. For example, Figure 15.7 depicts two isomorphic trees. In a general graph, the problem of determining whether two graphs are isomorphic is neither known to be NP-hard nor known to be solvable in polynomial time (this is a significant open problem in modern algorithmic graph theory!). For the special case of a tree, however, it is not too difficult to perform isomorphism testing efficiently in polynomial time.

(a) Suppose we have two *rooted* n-node trees. Unlike some of the rooted trees (in partic-

ular, search trees) we have seen thus far in this book, there is no particular ordering imposed on the children of any node. Two rooted trees are said to be isomorphic if we can order of the children for all nodes in both trees and obtain two trees with the same "shape" (we can define this more formally by saying the roots must have the same number of children, and corresponding children of the roots of the two trees must recursively have the same "shape"). Give an $O(n)$ algorithm for isomorphism testing in this setting. [Solution]

(b) The algorithm above can be used as a building block to obtain an $O(n^3)$ algorithm for isomorphism testing in non-rooted trees (i.e., *free trees*): we simply apply it for all $O(n^2)$ potential pairs of root nodes (one from the first tree and the other from the second tree), and for each pair we root the two trees and test for isomorphism between the two rooted trees. Can improve the overall running time of this algorithm to only $O(n^2)$ time? What about $O(n)$ time? [Solution]

15.6.5 Graphs of Bounded Treewidth

Many graphs are not trees, but they are sufficiently "tree-like" that we can apply to them many of the same algorithmic methods we have for trees. We do this by computing a *tree decomposition* of our graph — an $O(n)$-node tree whose nodes each correspond to a subset of the nodes in our original graph. The size of the largest such subset (minus one) is called the *width* of our tree decomposition, and the smallest width we can obtain over any tree decomposition of a graph is called the *tree-width* of the graph. Trees have tree-width 1, cycles have tree-width 2, planar graphs have tree-width $O(\sqrt{n})$, and an n-node clique (the least tree-like of any graph) has tree-width $n-1$. In general, the more a graph behaves like a tree, the smaller its tree-width. [Detailed animated description of a tree decomposition]

Unfortunately, it is an NP-hard problem to compute the exact tree-width of a general graph. However, one can compute a tree decomposition of at most twice the tree-width of a graph in polynomial time [Details]. This is typically sufficient for algorithmic purposes, since most problems that can be solved in linear time on trees by dynamic programming (e.g., the packing and covering problems from Section 15.4) can be solved in time linear in n but exponential in k on an n-node graph of tree-width k by dynamic programming. As long as we have constant tree-width, we therefore still obtain a linear total running time; see the endnotes for further remarks on the types of problems that can be solved in linear time in this manner. [Example: computing independent sets]

15.6.6 Intersection Graphs

Intersection graphs come in many shapes and sizes. They are often implicitly described in terms of intersections among combinatorial or geometric objects, as shown in Figure 15.8. This can cause certain algorithmic difficulties; for example, the m edges of a unit disk graph (Figure 15.8(c)) are implicitly defined by specifying the locations of n unit circles in the plane; hence even if we apply an $O(m)$ algorithm to such a graph, we can no longer rightly claim that it runs in "linear" time in the size of our input, which is only $O(n)$ (so it is conceivable that one may be able to solve a problem on such a graph in less than $O(m)$ time, which is typically not possible for explicitly-represented graphs).

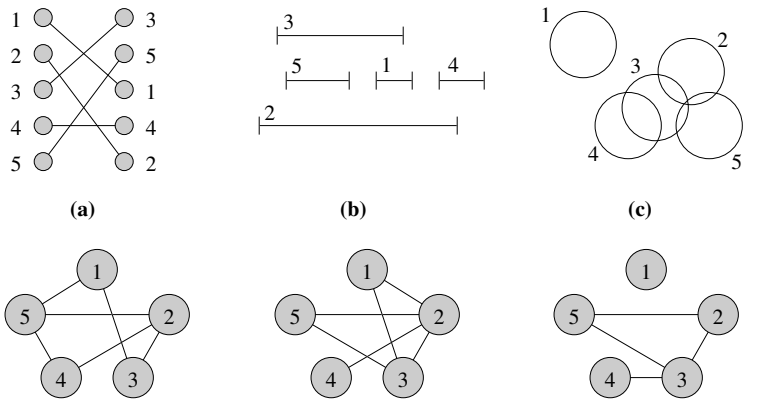

FIGURE 15.8: Examples of intersection graphs: (a) depicts a permutation graph, with an edge corresponding to each inversion in the permutation $\pi = 3, 5, 1, 4, 2$ (each of which in turn corresponds to an intersection between two edges when we depict our permutation as a bipartite graph), (b) gives an example of an interval graph, where two nodes are connected by an edge if their corresponding intervals overlap, and (c) illustrates a unit disk graph, whose edges are implicitly defined by intersections among a set of n unit circles in the plane.

One of the most difficult challenges we face with intersection graphs is recognizing them to begin with. That is, given a graph, can we efficiently test whether or not it is a permutation graph, an interval graph, a unit disk graph, or some other type of intersection graph? Once we recognize a particular type of intersection graph, however, we can often solve many problems more efficiently than we could on general graphs, and many NP-hard problems now admit polynomial-time solutions. Examples are given in the following exercises.

Problem 264 (Permutation Graphs). Suppose we wish to compute a permutation of the integers $1 \ldots n$ that is described only in terms of its inversions (recall that an inversion is a pair (i, j) such that $i < j$ and i appears after j in the permutation). For example, we might describe the permutation $\pi =< 3, 5, 1, 4, 2 >$ by saying that $n = 5$ and that pairs $(1, 3)$, $(1, 5)$, $(2, 3)$, $(2, 4)$, $(2, 5)$, and $(4, 5)$ constitute inversions in π. From this information, can we reconstruct π? Or more generally, given a set of inversions, does there exist a corresponding permutation and if so, what is it? This question is equivalent to the question of recognizing what are known as *permutation graphs*. As shown in Figure 15.8(a), a permutation graph is defined by a permutation by including an edge for every inversion in the permutation. In other words, permutation graphs are intersection graphs described by edge crossings in the bipartite representation of a permutation, shown in Figure 15.8(a).

(a) Let us work towards an efficient algorithm for recognizing a permutation graph, and constructing its associated permutation. First, please try to prove that if G is a permutation graph, then both G and its complement (the graph you get when you replace edges with non-edges, and non-edges with edges) must be comparability graphs (see problem 270). [Solution]

(b) Prove that the condition from part (a) is actually sufficient to characterize a permutation graph. In other words, suppose that G and its complement are both comparability

graphs. In this case, we can compute a transitive orientation of both of these graphs, the union of which will be a tournament (recall problem 251). Tournaments in general are not necessarily acyclic, but this particular tournament must be. Please try to prove this fact, and argue that a topological ordering of the resulting acyclic tournament gives us the permutation we seek. Hint: If our tournament has a long directed cycle, argue that it must also have a directed cycle of length 3, and that this leads to a contradiction. [Solution]

(c) Many problems can be solved more efficiently in permutation graphs than on general graphs. For example, the NP-hard maximum-cardinality independent set problem can be solved in $O(n \log \log n)$ expected time on a permutation graph, once we know its underlying permutation; this is equivalent to problem 210(e). Please show how to use dynamic programming to solve the weighted version of this problem, the *maximum-value* independent set problem, in $O(n^2)$ time, assuming we are given the underlying permutation that defines our graph. Next, show how your algorithm can be used to compute a maximum-value clique in $O(n^2)$ time; as a hint, what can you say about the complement of a permutation graph? [Solution]

(d) Please show how to compute the chromatic number of a permutation graph (along with a corresponding coloring) in polynomial time, assuming we are given its underlying permutation. [Solution]

Problem 265 (Unit Disk Graphs). A *unit disk graph* is the intersection graph of a collection of n unit circles in the plane (we treat tangent circles as intersecting), as shown in Figure 15.8(c). It is unfortunately an NP-hard problem to recognize whether a graph is a unit disk graph. However, if we know that a graph is a unit disk graph, we can still achieve improved algorithmic performance on certain problems, even without knowing the underlying geometric configuration of disks that defines the graph. Please give a simple greedy algorithm that computes a 5-approximation to the maximum-cardinality independent set problem in a unit disk graph, *without* knowing its underlying geometric representation. To get started, note that within the immediate neighborhood of any node, at most 5 nodes can belong to an independent set. If you would like a challenge, see if you can improve the approximation guarantee to 3 (hint: show that the same reasoning above allows you to replace 5 with 3 for a node on the boundary of the collection). [Solution]

15.6.7 Chordal Graphs

Chordal graphs are perhaps slightly less prominent than most of the other classes above, but we feel they deserve a brief mention due to their algorithmic properties. A *chord* of a cycle C in a graph is an edge that connects nodes that are not neighbors along C, and a chordal graph is a graph in which every cycle of length at least 4 has a chord. To give a more intuitive feeling for this property, note that chordal graphs are sometimes called "triangulated" graphs: every cycle C of length at least 4 can be decomposed into two smaller cycles by adding a chord to C, and we can repeatedly apply this process to eventually decompose C into a collection of triangles[3]. As a consequence of the chordal property, every chordal graph must also have a node that is *simplicial*, whose set of neighbors is a clique. Chordal graphs can also be characterized as intersection graphs — one can show that they are intersection graphs of subtrees in a tree. In fact, it is fairly easy to transform a chordal graph into a tree on at most n nodes, called a *clique tree*, in which we can

[3]In Chapter 22, we will see another type of "triangulated" graph in the form of planar graphs in which all faces are triangles; although the word "triangulated" is used in both cases, the two types of graphs are not related.

identify a collection of subtrees (one for each node in the original chordal graph) for which intersection between subtrees corresponds to the existence of an edge in the original chordal graph. [Further details]

From an algorithmic perspective, chordal graphs are interesting because their nodes can always be arranged along the number line in a *perfect elimination ordering*, where the neighbors of each node i preceding i in this ordering form a clique. The final node is therefore simplicial, and if we remove it from the graph, the next-to-last node becomes simplicial, and so on. In fact, one way to obtain a perfect elimination ordering is to repeatedly select an arbitrary simplicial node from our graph and remove it. Perfect elimination orderings play an algorithmic role much like topological orderings for DAGs. By computing a perfect elimination ordering of a chordal graph as a preprocessing step, we can often then solve certain graph problems more efficiently than on general graphs; for example, the maximum-value clique, maximum-value independent set, and chromatic number problems are all solvable in linear time on a chordal graph once we are given a perfect elimination ordering. [Details]

Several relatively simple algorithms are known for computing perfect elimination orderings (and more generally, for recognizing chordal graphs) in linear time. One popular approach is a variant of BFS called *lexicographic BFS* [Further details], and another is the *maximum adjacency (MA) ordering* algorithm, which we discuss in greater detail later in Section 19.3.1. [Further details].

Problem 266 (Interval Graphs). *Interval graphs* are a well-studied subcategory of intersection graphs, defined by the intersections within a collection of intervals on the number line, as shown in Figure 15.8(b). One can actually recognize interval graphs in linear time (and also construct a set of compatible intervals on the number line), but we omit discussion of algorithms for doing so, since they are slightly complicated. Note that many graph problems can be solved efficiently on an interval graph once we know its underlying interval representation; for example, the greedy algorithm we discussed in Section 10.5 for the maximum interval packing problem can be used to solve the maximum independent set problem in an interval graph in linear time. Here, we focus on giving a simple mathematical characterization for such graphs that can at least be verified in polynomial time. Interval graphs have several applications in practice; for example, in computational biology, if we imagine intervals to be small pieces of DNA, then interval graph recognition helps us determine how configurations of these pieces might be arranged when they adhere together in a biological system.

(a) Argue that every interval graph is chordal. Hence, problems that can be solved efficiently on chordal graphs can also be solved efficiently on interval graphs. Also, argue that the complement of an interval graph is a comparability graph (see problem 270). [Solution]

(b) Please try to argue that the conditions above are actually sufficient, so a graph is an interval graph if and only if it is chordal and its complement is a comparability graph. As a hint, try to show that the comparability condition forces the clique tree of our graph to be a path, from which we can construct an interval representation. [Solution]

15.7 Additional Problems

Problem 267 (Solving the 2SAT Problem). In the first chapter of this book we

discussed the NP-complete satisfiability (SAT) problem, which asks us to decide whether a Boolean expression can be satisfied by some assignment of true and false values to its variables. Every Boolean expression can be rewritten in *conjunctive normal form* (CNF) as a set of "OR" *clauses* that are all "AND"ed together; for example, take the following expression:

$$\underbrace{(x_2 \vee \overline{x}_4 \vee \overline{x}_3)}_{\text{clause 1}} \wedge \underbrace{x_1}_{\text{clause 2}} \wedge \underbrace{(\overline{x}_2 \vee x_4)}_{\text{clause 3}} \wedge \underbrace{(x_2 \vee \overline{x}_1 \vee x_3)}_{\text{clause 4}}.$$

This expression is satisfiable if we set $x_1 \ldots x_4$ all to true. Not only can every Boolean expression be written in CNF form, it can also be written in what is called 3CNF form, where each clause contains at most 3 variables (can you see how to do this?). As a result, the 3SAT problem (determining satisfiability of a 3CNF formula) is NP-complete[4], since the SAT problem is NP-complete. However, some Boolean formulas can be written in 2CNF form, with only 2 variables per clause. Show how to solve the 2SAT problem in linear time using graph search techniques. [Solution]

Problem 268 (Static Cuckoo Hash Table Construction). Suppose we are given a set of n integers and a pair of hash functions from which we would like to build a cuckoo hash table (see Section 7.3.2). For both the single-array and dual-array Cuckoo hash table variants, please use simple graph algorithms to compute a valid arrangement of the n elements within a Cuckoo hash table of a given size in $O(n)$ time. Your algorithm should also be able to determine if no such valid arrangement exists. [Solution]

Problem 269 (Orienting Edges to Achieve Even In-Degree). Let G be a connected graph with an even number of edges. Please give an efficient algorithm for orienting the edges of G (i.e., each edge $\{i, j\}$ must be oriented as a directed edge (i, j) or (j, i), but not both) so that every node has even in-degree. [Solution]

Problem 270 (Transitive Orientation). An undirected graph is said to be a *comparability graph* if its edges can be oriented (i.e., each edge $\{i, j\}$ must be oriented as a directed edge (i, j) or (j, i), but not both) so that the resulting graph is transitive: if it contains directed edges (i, j) and (j, k), then it must also contain the directed edge (i, k). Argue briefly that the transitive orientation of a graph is a DAG. Linear time algorithms have been discovered for recognizing comparability graphs and computing their transitive orientations, but these algorithms are somewhat complicated — see the endnotes for further details. Show how to compute a transitive orientation of a comparability graph in $O(mn)$ time by a simple reduction to the 2SAT problem above. [Solution]

Problem 271 (Disjoint Edge Covers). Consider the problem of partitioning the edges of a graph into two disjoint sets, each of which is an edge cover. We can think of this as "coloring" the edges of a graph with two colors so that each node has at least one incident edge of each color (note, however, that this differs quite a bit from the usual notion of coloring, where each node can have at most one incident edge of each color). Please give a simple linear-time algorithm for solving this problem, which is also capable of recognizing whether no feasible solution exists. [Solution]

Problem 272 (Series-Parallel Graphs). *Series-parallel graphs* are a special type of graph with which you may be familiar if you have studied analog circuits, since these graphs correspond to resistive networks that can be simplified down to a single resistor by applying "series" and "parallel" reductions. We can define series-parallel graphs more formally in a recursive fashion, according to Figure 15.9. We say a graph is s-t series-parallel if it is series-parallel with respect to two designated nodes s and t as follows: a single edge (s, t) is defined to be s-t series-parallel as a base case. We can then construct

[4] Interestingly, although we can determine if a 2SAT formula is completely satisfiable in polynomial time, the related problem MAX-2SAT asking us to determine the maximum number of satisfiable clauses in an unsatisfiable 2SAT formula is NP-hard!

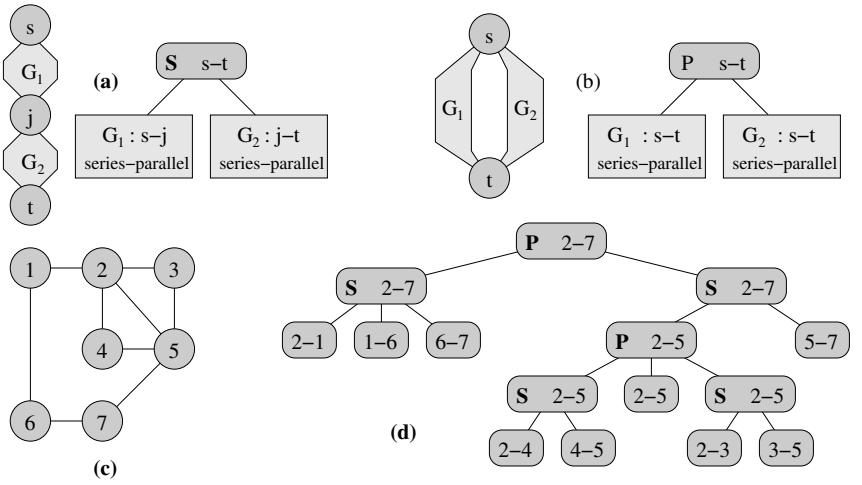

FIGURE 15.9: Example of (a) an s-t series-parallel graph where $s = 2$ and $t = 7$, and (b) a series-parallel tree representing this graph. The leaves of the tree correspond to edges in the original graph, and the internal nodes are of two types: serial nodes (whose children are ordered), and parallel nodes (whose children are unordered).

larger graphs by placing two or more series-parallel graphs in series, as shown in Figure 15.9(a) (i.e., we place an s-j series-parallel graph in series with a j-t series-parallel graph by identifying nodes j in each of the graphs), or in parallel, as shown in Figure 15.9(b) (i.e., we take two s-t series-parallel graphs and identify nodes s and t). The recursive structure of a large series-parallel graph (Figure 15.9(c)) can be depicted in a convenient form as a *series-parallel tree* (or *s-p tree*) shown in Figure 15.9(d). You may note that these trees are somewhat similar to the tree decompositions we studied in Section 15.6.5 (in fact, series-parallel graphs are known to have tree-width at most 2).

(a) Suppose we wish to test whether a given graph is s-t series-parallel for some pair of nodes s and t, and if so, construct its s-p tree. We can accomplish this task in much the same fashion as we might simplify an electrical network of resistors. Starting with every individual edge as its own series-parallel graph, we repeatedly located pairs of these graphs that are in series or in parallel with each-other and merge these pairs into larger aggregate series-parallel graphs, constructing the s-p tree in the process. Argue that our original graph is series-parallel for some pair of nodes s and t if and only if this process terminates with a single graph. Can you implement this approach so it runs in only linear time? [Solution]

(b) Many problems that are difficult to solve in general graphs are much easier to solve in series-parallel graphs. Informally, you may want to think of series-parallel graphs as just slightly more complicated than trees. For example, let us consider the *minimum-cost node cover problem*, which takes as input an undirected graph with costs associated with nodes and asks for a minimum-cost subset of nodes that "covers" all the edges of the graph (i.e., for every edge, at least one of its endpoints must belong to the cover). Although this problem is NP-hard in general graphs, when we studied dynamic programming we discovered a linear-time algorithm for the special case of trees (problem 229). See if you can generalize this algorithm to obtain a linear-time algorithm for series-parallel graphs. You can assume the series-parallel graph is given

to you in the form of an s-p tree, if you wish. [Solution]

Problem 273 (Recognizing Valid Degree Sequences). Within a group of five individuals whom we call A, B, C, D, and E, you are told that A has 4 friends, B has 3 friends, and C, D, and E all have one friend each. Assuming friendship is an undirected (i.e., symmetric) relationship, is this possible? In terms of a graph, we are asking whether there exists some 5-node graph in which node degrees are 4, 3, 1, 1, and 1. It is not too difficult to see no such graph exists. In this problem, we develop an efficient algorithm for recognizing "degree sequences" that correspond to valid graphs.

(a) Suppose you are given a sequence $d_1 \ldots d_n$, and you would like to know if an n-node graph exists whose node degrees match this sequence. Show that if such graphs do exist, then in at least one of them we will find that a node (say, i) of maximum degree (say, d_{max}) is connected to the d_{max} highest-degree nodes excluding i. As a hint, start with any graph satisfying the degree sequence, and perform appropriate "exchanges" to bring it into the desired form while maintaining degrees of nodes. [Solution]

(b) Given the insight from part (a) and using appropriate data structures, design an $O(n \log n)$ algorithm for recognizing whether a degree sequence $d_1 \ldots d_n$ corresponds to some valid graph. The output of your algorithm should be 'yes' or 'no'. [Solution]

