[T]he distributions and partitions of knowledge are not like several lines that meet in one angle, and so touch but in a point, but are like branches of a tree that meet in a stem, which hath a dimension and quantity of entireness and continuance before it come to discontinue and break itself into arms and boughs.

— Francis Bacon, *The Advancement of Learning* (1605)

Thus you see, most noble Sir, how this type of solution bears little relationship to mathematics, and I do not understand why you expect a mathematician to produce it, rather than anyone else.

— Leonhard Euler, describing the Königsburg bridge problem
in a letter to Carl Leonhard Gottlieb Ehler (April 3, 1736)

Well, ya turn left by the fire station in the village and take the old post road by the reservoir and... no, that won't do.

Best to continue straight on by the tar road until you reach the schoolhouse and then turn left on the road to Bennett's Lake until... no, that won't work either.

East Millinocket, ya say? Come to think of it, you can't get there from here.

— Robert Bryan and Marshall Dodge,
Bert and I and Other Stories from Down East (1961)

5

Basic Graph Algorithms

5.1 Introduction and History

A graph is a collection of pairs—pairs of integers, pairs of people, pairs of cities, pairs of stars, pairs of countries, pairs of scientific papers, pairs of web pages, pairs of game positions, pairs of recursive subproblems, even pairs of graphs. Mirroring the most common method for visualizing graphs, the underlying objects being paired are usually called ***vertices*** or ***nodes***, and the pairs themselves are called ***edges*** or ***arcs***, but in fact the objects and pairs can be anything at all.

One of the earliest examples of graphs are *road networks* and maps thereof. Roman engineers constructed a network of more than 400 000 km of public roads across Europe, western and central Asia, and northern Africa during the height of the Roman empire. Travelers on the road network would carry *itineraria*, which were either simple lists or more pictorial representations of the landmarks and distances along various roads. The *Tabula Peutingeriana*, a 13th-century scroll depicting the entire Roman *cursus publicus*, is widely believed to be a medieval copy of a 5th-century revision of a 1st-century *itinerarium pictum*, commissioned during the reign of Augustus Caesar. The Peutinger Table is not a geographically accurate map—historians debate whether it qualifies as a "map" at all!—but an abstract representation of the road network, similar to a modern subway map. Cities along each road are indicated by kinks in the curve representing that road; the names of these cites and the lengths of road segments between them are also indicated on the map. Thus, the map contains enough information to find the shortest route between any two cities in the 5th-century Roman empire. See Figure 5.1.

Figure 5.1. A small excerpt of Konrad Miller's 1872 restoration of the *Tabula Peutingeriana*, showing the Roman road from modern-day Birten (*Veteribus*, top left) through Köln (*Agripina*) and Bonn (*Bonnae*) to Mainz (*Mogontiaco*, top right), with branches to Trier (*Avg Tresvirorvm*, center) and Metz (*Matricorvm*, bottom center). (See Image Credits at the end of the book.)

One of the oldest classical applications of graphs—and specifically trees—is in representing genealogies. Complex family "trees" have been used for centuries to settle legal questions about marriage, inheritance, and royal succession. Civil law in the Roman empire, later adopted as canon law by the early Catholic Church, forbade marriage between first cousins or closer relatives. In the early ninth century, the Church changed both the required distance and the method of computation. Where the Roman *computatio legalis* required the sum of the distances to the nearest common ancestor to be at least four, the newer *computatio canonica* required the maximum of the two distances to be at least seven. In 1215, bowing to practical considerations (and actual practice), the Church relaxed the minimum required distance for marriage to four.[1] The left diagram in Figure 5.2 illustrates a particularly convoluted case: Tirius and Theburga marry and have a son Gaius, after which Tirius dies; Theburga then

[1]During the 11th and 12th centuries, this restriction gradually expanded to include up to four links by affinity, initially through marriage, and later through extra-marital sex, betrothal, and even godparenting. For example, marriage between a man and his sister's husband's sister's husband's sister was formally forbidden, as was a marriage between a widower and his son's wife's widowed mother. These affinity requirements were significantly reduced but not eliminated in 1215; the Church only abandoned the concept of affinity *ex copula illicita* in 1917.

marries Lothar, bears him a son, and dies; finally, Lothar and Bertha marry and have a daughter Gemma. Can Gaius's son legally marry Gemma's daughter?

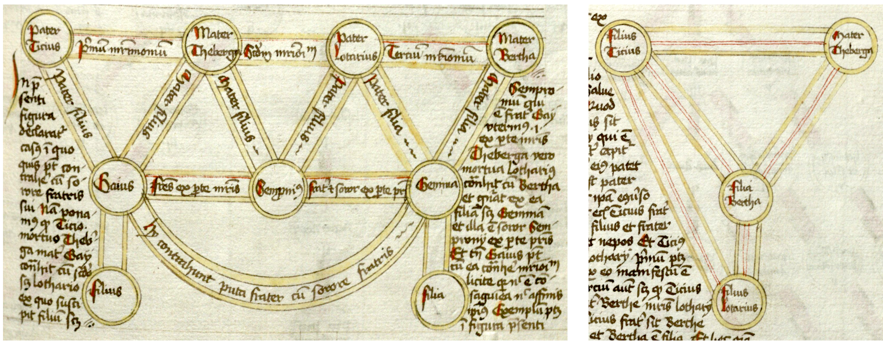


Figure 5.2. Two diagrams describing a complex marriage case, from an anonymous 15th-century treatise on Johannes Andreae's *Super arboribus consanguinitatis et affinitatis*, an early 14th-century treatise on canon law. (See Image Credits at the end of the book.)

In the late 1600s, French mathematician Pierre Varignon developed a graphical method for finding the equilibrium position of a tree-like network of ropes under tension, building on earlier work by Simon Stevin published a century earlier. Varignon observed that when the ropes are at equilibrium, one can draw a graph whose edges are segments parallel to the ropes, with lengths equal to the forces along those ropes, such that the ropes meeting at any point in the network define a closed cycle in the graph. Varignon's method of "graphical statics" was not published in complete detail until 1725, two years after his death. These graphs are now known as *reciprocal force diagrams* or *Maxwell-Cremona diagrams*, after James Clerk Maxwell and Luigi Cremona, who (along with Carl Culmann and others) developed a rich theory of reciprocal diagrams in the late 1800s.

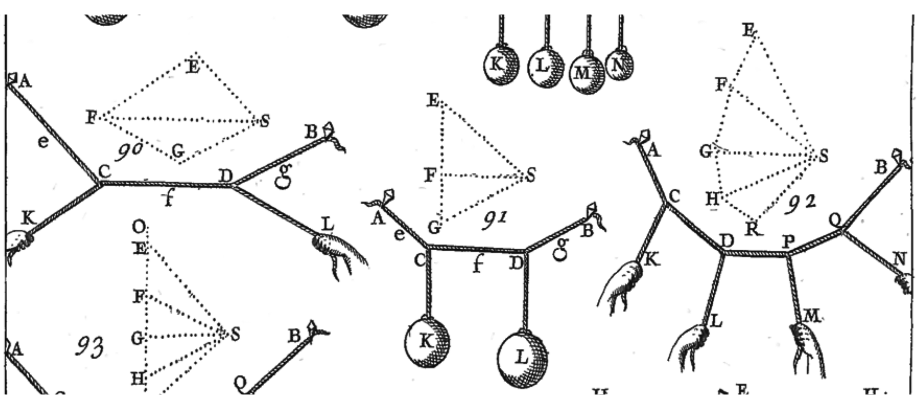


Figure 5.3. Reciprocal force diagrams (dotted), from Varignon's posthumous *Nouvelle mécanique, ou statique, dont le projet fut donné en MDCLXXXVII [New mechanics, or statics, whose project was given in 1687]* (See Image Credits at the end of the book.)

Of course, there are many other familiar examples of graphs, like board games (dating to antiquity); vertices and edges of convex polyhedra (formally studied by ancient Greek philosophers, but much older); visualizations of star patterns (already developed in East Asia by the 7th century CE); knight's tours (described by al-Adli, Rudraṭa, al-Suli, and others in the 9th and 10th centuries), mazes (introduced in their modern form by Giovanni Fontana circa 1420); geodetic triangulations (introduced by Gemma Frisius in 1533, and used to calculate the circumference of the earth by Willebrod Snell in 1615 and to define the meter in 1799), Leonhard Euler's well-known partial[2] solution to the Bridges of Königsburg puzzle (1735); telegraph and other communication networks (first proposed in 1753, developed by Ronalds, Schilling, Gauss, Weber, and others in the early 1800s, and deployed worldwide by the late 1800s); electrical circuits (formalized in the early 1800s by Ohm, Maxwell, Kirchhoff, and others); molecular structural formulas (introduced independently by August Kekulé in 1857 and Archibald Couper in 1858); social networks (first studied in the mid-1930s by sociologist Jacob Moreno); digital electronic circuits (proposed by Charles Sanders Peirce in 1886, and cast into their modern form by Claude Shannon in 1937); and yeah, okay, if you insist, the modern internet.

The word "graph" for the abstract mathematical was coined by James Sylvester in 1878, who adapted Kekulé's "chemicographs" to describe certain algebraic invariants, at the suggestion of his colleague William Clifford. The word "tree" was first used for connected acyclic graphs by Arthur Cayley in 1857, although the abstract concept of trees had already been used by Gustav Kirchhoff and Karl von Staudt ten years earlier. The zeroth book on graph theory was published by André Sainte-Laguë in 1926; Dénes Kőnig published the first graph theory book ten years later.

5.2 Basic Definitions

Formally, a (simple) ***graph*** is a pair of sets (V, E), where V is an arbitrary non-empty finite set, whose elements are called ***vertices***[3] or ***nodes***, and E is a set of pairs of elements of V, which we call ***edges***. In an ***undirected*** graph, the edges are unordered pairs, or just sets of size two; I usually write $\boldsymbol{uv}$ instead of $\{u, v\}$ to denote the undirected edge between u and v. In a ***directed*** graph, the edges are ordered pairs of vertices; I usually write $\boldsymbol{u \to v}$ instead of (u, v) to denote the directed edge from u to v.

[2] Euler dismissed the final step of his argument—actually finding an Euler tour of a graph when every vertex has even degree—as obvious. Euler also failed to notice that a graph with an Euler tour must be connected. The first complete proof that a graph has an Euler tour if and only if it is connected and every vertex has even degree was published by Carl Hierholzer in 1873.

[3] The singular of the English word "vertices" is ***vertex***. Similarly, the singular of "matrices" is ***matrix***, and the singular of "indices" is ***index***. Unless you're speaking Italian, there is no such

Following standard (but admittedly confusing) practice, I will also use V to denote the *number* of vertices in a graph, and E to denote the *number* of edges. Thus, in any undirected graph we have $0 \le E \le \binom{V}{2}$, and in any directed graph we have $0 \le E \le V(V-1)$.

The ***endpoints*** of an edge uv or $u\to v$ are its vertices u and v. We distinguish the endpoints of a directed edge $u\to v$ by calling u the ***tail*** and v the ***head***.

The definition of a graph as a pair of *sets* forbids multiple undirected edges with the same endpoints, or multiple directed edges with the same head and the same tail. (The same directed graph can contain both a directed edge $u\to v$ and its reversal $v\to u$.) Similarly, the definition of an undirected edge as a *set* of vertices forbids an undirected edge from a vertex to itself. Graphs *without* loops and parallel edges are often called ***simple*** graphs; non-simple graphs are sometimes called ***multigraphs***. Despite the formal definitional gap, most algorithms for simple graphs extend to multigraphs with little or no modification, and for that reason, I see no need for a formal definition here.

For any edge uv in an undirected graph, we call u a ***neighbor*** of v and vice versa, and we say that u and v are ***adjacent***. The ***degree*** of a node is its number of neighbors. In directed graphs, we distinguish two kinds of neighbors. For any directed edge $u\to v$, we call u a ***predecessor*** of v, and we call v a ***successor*** of u. The ***in-degree*** of a vertex is its number of predecessors; the ***out-degree*** is its number of successors.

A graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A ***proper subgraph*** of G is any subgraph other than G itself.

A ***walk*** in an undirected graph G is a sequence of vertices, where each adjacent pair of vertices are adjacent in G; informally, we can also think of a walk as a sequence of edges. A walk is called a ***path*** if it visits each vertex at most once. For any two vertices u and v in a graph G, we say that v is ***reachable*** from u if G contains a walk (and therefore a path) between u and v. An undirected graph is ***connected*** if every vertex is reachable from every other vertex. Every undirected graph consists of one or more ***components***, which are its maximal connected subgraphs; two vertices are in the same component if and only if there is a path between them.[4]

A walk is ***closed*** if it starts and ends at the same vertex; a ***cycle*** is a closed walk that enters and leaves each vertex at most once. An undirected graph is ***acyclic*** if no subgraph is a cycle; acyclic graphs are also called ***forests***. A ***tree*** is a connected acyclic graph, or equivalently, one component of a forest. A ***spanning***

thing as a vertice, matrice, indice, appendice, helice, apice, vortice, radice, simplice, codice, directrice, dominatrice, Unice, Kleenice, Asterice, Obelice, Dogmatice, Getafice, Cacofonice, Vitalstatistice, Geriatrice, or Jimi Hendrice! If you have trouble remembering this rule, stick to "node".

[4]Components are often called "connected components", but this usage is redundant; components are connected by definition.

tree of an undirected graph G is a subgraph that is a tree and contains every vertex of G. A graph has a spanning tree if and only if it is connected. A ***spanning forest*** of G is a collection of spanning trees, one for each component of G.

Directed graphs require slightly different definitions. A ***directed walk*** is a sequence of vertices $v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ such that $v_{i-1} \to v_i$ is a directed edge for every index i; directed paths and directed cycles are defined similarly. Vertex v is ***reachable*** from vertex u in a directed graph G if and only if G contains a directed walk (and therefore a directed path) from u to v. A directed graph is ***strongly connected*** if every vertex is reachable from every other vertex. A directed graph is ***acyclic*** if it does not contain a directed cycle; directed acyclic graphs are often called ***dags***.

5.3 Representations and Examples

The most common way to visually represent graphs is by ***drawing*** them. A drawing of a graph maps each vertex to a point in the plane (typically drawn as a small circle or some other shape) and each edge to a curve or straight line segment between the two vertices. A graph is ***planar*** if it has a drawing where no two edges cross; such a drawing is also called an ***embedding***.[5] The same graph can have many different drawings, so it is important not to confuse a particular drawing with the graph itself. In particular, planar graphs can have non-planar drawings!

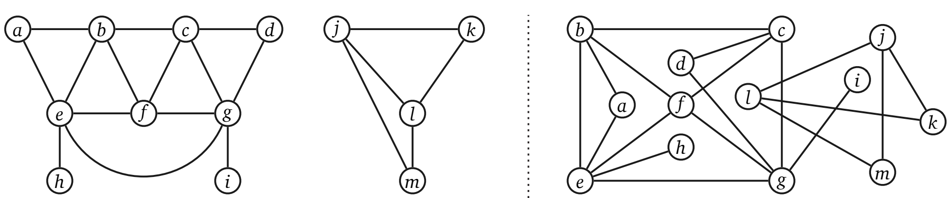


Figure 5.4. Two drawings of the same disconnected planar graph with 13 vertices, 19 edges, and two components. Only the drawing on the left is an embedding.

However, drawings are far from the only useful representation of graphs. For example, the ***intersection graph*** of a collection of geometric objects has a node for every object and an edge for every intersecting pair of objects. Whether a particular graph can be represented as an intersection graph depends on what kind of object you want to use for the vertices. Different types of objects—line segments, rectangles, circles, etc.—define different classes of graphs. One particularly useful type of intersection graph is an *interval graph*, whose vertices

[5]Confusingly, the word "embedding" is often used as a synonym for "drawing", even when the edges intersect. Please don't do that.

are intervals on the real line, with an edge between any two intervals that overlap.

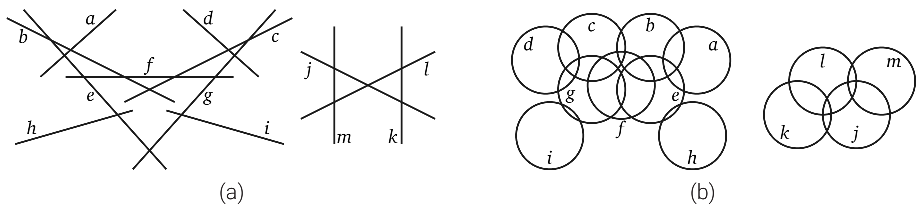


Figure 5.5. The graph in Figure 5.4 is also the intersection graph of (a) a set of line segments and (b) a set of circles.

Another good example is the ***dependency graph*** of a recursive algorithm. Dependency graphs are directed acyclic graphs. The vertices are all the distinct recursive subproblems that arise when executing the algorithm on a particular input. There is an edge from one subproblem to another if evaluating the second subproblem requires a recursive evaluation of the first. For example, for the Fibonacci recurrence

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{otherwise,} \end{cases}$$

the vertices of the dependency graph are the integers $0, 1, 2, \dots, n$, and the edges are the pairs $(i-1)\rightarrow i$ and $(i-2)\rightarrow i$ for every integer i between 2 and n.

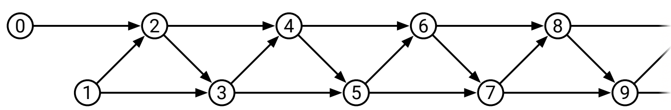


Figure 5.6. The dependency graph of the Piṅgala-Fibonacci recurrence.

As a more complex example, recall the recurrence for the *edit distance* problem from Chapter 3:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min\left\{ \begin{array}{c} Edit(i-1,j)+1 \\ Edit(i,j-1)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

The dependency graph of this recurrence is an $m \times n$ grid of vertices (i, j) connected by vertical edges $(i-1, j)\rightarrow(i, j)$, horizontal edges $(i, j-1)\rightarrow(i, j)$, and diagonal edges $(i-1, j-1)\rightarrow(i, j)$. Dynamic programming works efficiently for any recurrence that has a reasonably small dependency graph; a proper evaluation order ensures that each subproblem is visited *after* its predecessors.

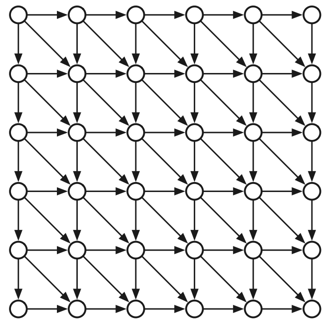


Figure 5.7. The dependency graph of the edit distance recurrence.

Another interesting example is the ***configuration graph*** of a game, puzzle, or mechanism like tic-tac-toe, checkers, the Rubik's Cube, the Tower of Hanoi, or a Turing machine. The vertices of the configuration graph are all the valid configurations of the puzzle; there is an edge from one configuration to another if it is possible to transform one configuration into the other with a single simple "move". (Obviously, the precise definition depends on what moves are allowed.) Even for reasonably simple mechanisms, the configuration graph can be extremely complex, and we typically only have access to local information about the configuration graph.

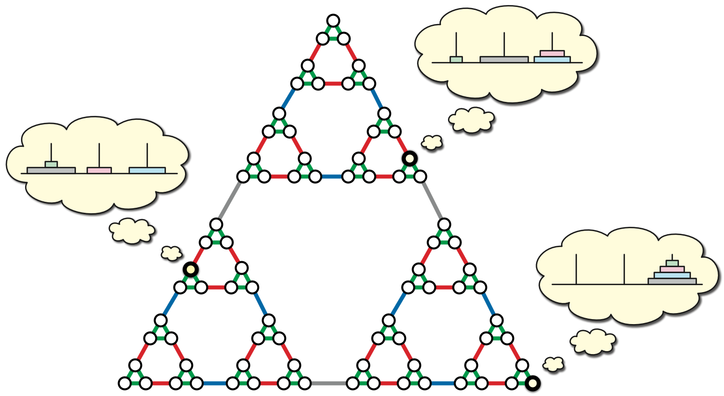


Figure 5.8. The configuration graph of the 4-disk Tower of Hanoi.

Configuration graphs are close relatives of the *game trees* we considered in Chapter 2, but with one crucial difference. Each state of a game appears *exactly* once in its configuration graph, but can appear many times in its game tree. In short, configuration graphs are *memoized* game trees!

Finite-state automata used in formal language theory can be modeled as labeled directed graphs. Recall that a deterministic finite-state automaton is formally defined as a 5-tuple $M = (\Sigma, Q, s, A, \delta)$, where Σ is a finite set called the *alphabet*, Q is a finite set of *states*, $s \in Q$ is the *start state*, $A \subseteq Q$ is the set of *accepting states*, and $\delta : Q \times \Sigma \to Q$ is a *transition function*. But it is often more useful to think of M as a directed graph G_M whose vertices are the states Q, and whose edges have the form $q{\to}\delta(q,a)$ for every state $q \in Q$ and symbol $a \in \Sigma$. Basic questions about the language $L(M)$ accepted by M can then be phrased as questions about the graph G_M. For example, $L(M) = \varnothing$ if and only if no accepting state/vertex is reachable from the start state/vertex s.

Finally, sometimes one graph can be used to implicitly represent other larger graphs. A good example of this implicit representation is the *subset construction*, which is normally used to convert NFAs into DFAs, but can be applied to *arbitrary* directed graphs as follows. Given *any* directed graph $G = (V, E)$, we can define a new directed graph $G' = (2^V, E')$ whose vertices are all *subsets* of vertices in V, and whose edges E' are defined as follows:

$$E' := \left\{A{\to}B \;\middle|\; u{\to}v \in E \text{ for some } u \in A \text{ and } v \in B\right\}$$

We can mechanically translate this definition into an algorithm to construct G' from G, but strictly speaking, this construction is unnecessary, because G is *already* an *implicit* representation of G'.

It's important not to confuse any of these examples/representations with the actual formal *definition*: A graph is a pair of sets (V, E), where V is an *arbitrary* non-empty finite set, and E is a set of pairs (either ordered or unordered) of elements of V. In short: A graph is a set of pairs of things.

5.4 Data Structures

In practice, graphs are usually represented by one of two standard data structures: *adjacency lists* and *adjacency matrices*. At a high level, both data structures are arrays indexed by vertices; this requires that each vertex has a unique integer identifier between 1 and V. In a formal sense, these integers *are* the vertices.

Adjacency Lists

By far the most common data structure for storing graphs is the ***adjacency list***. An adjacency list is an array of lists, each containing the neighbors of one of the vertices (or the out-neighbors if the graph is directed).[6] For undirected graphs, each edge uv is stored twice, once in u's neighbor list and once in v's neighbor list; for directed graphs, each edge $u{\to}v$ is stored only once, in the neighbor list of the tail u. For both types of graphs, the overall space required for an adjacency list is $O(V + E)$.

[6]Attentive students might notice that despite is name, an adjacency list is not a list. This nomenclature is an example of the Red Herring Principle: In computer science, as in mathematics, a red herring is neither necessarily red nor necessarily a fish.

There are several different ways to represent these neighbor lists, but the standard implementation uses a simple singly-linked list. The resulting data structure allows us to list the (out-)neighbors of a node v in $O(1+\deg(v))$ time; just scan v's neighbor list. Similarly, we can determine whether $u \rightarrow v$ is an edge in $O(1+\deg(u))$ time by scanning the neighbor list of u. For undirected graphs, we can improve the time to $O(1+\min\{\deg(u), \deg(v)\})$ by simultaneously scanning the neighbor lists of both u and v, stopping either when we locate the edge or when we fall of the end of a list.

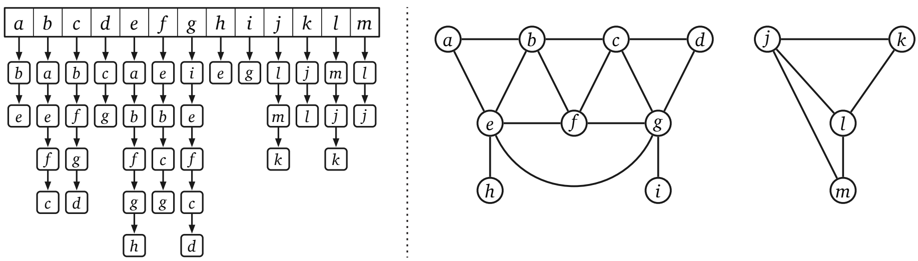


Figure 5.9. An adjacency list for our example graph.

Of course, linked lists are not the only data structure we could use; any other structure that supports searching, listing, insertion, and deletion will do. For example, we can reduce the time to determine whether uv is an edge to $O(1+\log(\deg(u)))$ by using a balanced binary search tree to store the neighbors of u, or even to $O(1)$ time by using an appropriately constructed hash table.[7]

One common implementation of adjacency lists is the ***adjacency array***, which uses a single array to store all edge records, with the records of edges incident to each vertex in a contiguous interval, and with a separate array storing the index of the first edge incident to each vertex. Moreover, it is useful to keep the intervals for each vertex in sorted order, as shown in Figure 5.10, so that we can check in $O(\log \deg(u))$ time whether two vertices u and v are adjacent.

Adjacency Matrices

The other standard data structure for graphs is the ***adjacency matrix***,[8] first proposed by Georges Brunel in 1894. The adjacency matrix of a graph G is a $V \times V$ matrix of 0s and 1s, normally represented by a two-dimensional array $A[1..V, 1..V]$, where each entry indicates whether a particular edge is present in G. Specifically, for all vertices u and v:

- if the graph is undirected, then $A[u, v] := 1$ if and only if $uv \in E$, and

[7]This is a *lot* more subtle than it sounds. Most popular hashing techniques do *not* guarantee fast query times, and even most *good* hashing methods can guarantee only $O(1)$ *expected* time. See http://algorithms.wtf for a more thorough discussion of hashing.

[8]See footnote 3.

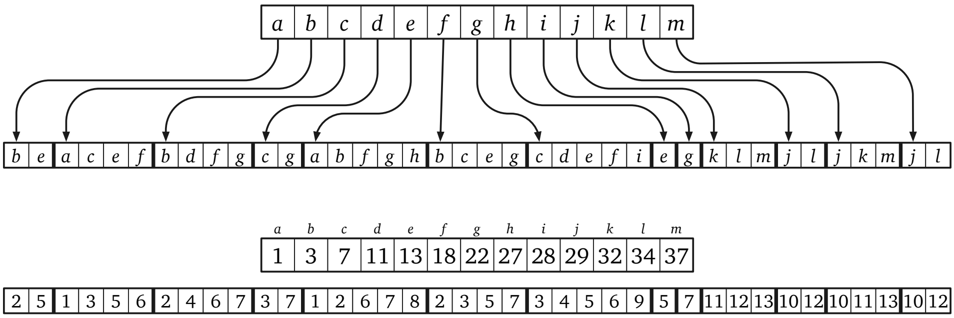


Figure 5.10. An abstract adjacency array for our example graph, and its actual implementation as a pair of integer arrays.

- if the graph is directed, then $A[u, v] := 1$ if and only if $u \to v \in E$.

For undirected graphs, the adjacency matrix is always *symmetric*, meaning $A[u, v] = A[v, u]$ for all vertices u and v, because uv and vu are just different names for the same edge, and the diagonal entries $A[u, u]$ are all zeros. For directed graphs, the adjacency matrix may or may not be symmetric, and the diagonal entries may or may not be zero.

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0

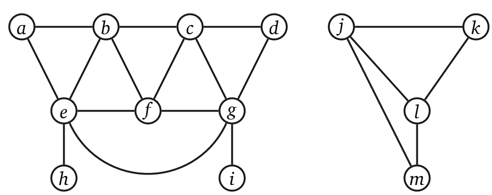


Figure 5.11. An adjacency matrix for our example graph.

Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge just by looking in the appropriate slot in the matrix. We can also list all the neighbors of a vertex in $\Theta(V)$ time by scanning the corresponding row (or column). This running time is optimal in the worst case, but even if a vertex has few neighbors, we still have to scan the entire row to find them all. Similarly, adjacency matrices require $\Theta(V^2)$ space, regardless of how many edges the graph actually has, so they are only space-efficient for very *dense* graphs.

Comparison

Table 5.1 summarizes the performance of the various standard graph data structures. Stars* indicate expected amortized time bounds for maintaining dynamic hash tables.[9]

	Standard adjacency list (linked lists)	Fast adjacency list (hash tables)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$	$O(1)$
Test if $u{\to}v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)^*$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$	$O(1)$

Table 5.1. Times for basic operations on standard graph data structures.

In light of this comparison, one might reasonably wonder why anyone would *ever* use an adjacency matrix; after all, adjacency lists with hash tables support the same operations in the same time, using less space. The main reason is that for sufficiently dense graphs, adjacency matrices are simpler and more efficient in practice, because they avoid the overhead of chasing pointers and computing hash functions; they're just contiguous blocks of memory.

Similarly, why would anyone use *linked lists* in an adjacency list structure to store neighbors, instead of balanced binary search trees or hash tables? Although the primary reason in practice is almost surely *tradition*—If they were good enough for Donald Knuth's code, they should be good enough for yours!—there are more principled arguments. One is that standard adjacency lists are in fact *good enough* for most applications. Most standard graph algorithms never (or rarely) actually ask whether an arbitrary edge is present or absent, or attempt to insert or delete edges, and so optimizing the data structures to support those operations is unnecessary.

But in my opinion, the most compelling reason for *both* standard data structures is that many graphs are *implicitly* represented by adjacency matrices and standard adjacency lists. For example:

- Intersection graphs are usually represented as a list of the underlying geometric objects. As long as we can test whether two objects intersect in constant time, we can apply any graph algorithm to an intersection graph by *pretending* that the input graph is stored explicitly as an adjacency matrix.

- Any data structure composed from records with pointers between them can be seen as a directed graph. Graph algorithms can be applied to these data structures by *pretending* that the graph is stored in a standard adjacency list.

[9]Don't worry if you don't understand the phrase "expected amortized".

- Similarly, we can apply any graph algorithm to a configuration graph *as though* it were represented as a standard adjacency list, provided we can enumerate all possible moves from a given configuration in constant time each.

For the last two examples, we can enumerate the edges leaving any vertex in time proportional to its degree, but we *cannot* necessarily determine in constant time if two vertices are adjacent. (Is there a pointer from this record to that record? Can we get from this configuration to that configuration in one move?) Moreover, we usually don't have the luxury of reorganizing the pointers in each record or the moves out of a given configuration into a more efficient data structure. Thus, a standard adjacency list, with neighbors stored in linked lists, is the appropriate model data structure.

In the rest of this book, unless explicitly stated otherwise, all time bounds for graph algorithms assume that the input graph is represented by a standard adjacency list. Similarly, unless explicitly stated otherwise, when an exercise asks you to design and analyze a graph algorithm, you should assume that the input graph is represented in a standard adjacency list.

5.5 Whatever-First Search

So far we have only discussed *local* operations on graphs; arguably the most fundamental *global* question we can ask about graphs is ***reachability***. Given a graph G and a vertex s in G, the reachability question asks which vertices are reachable from s; that is, for which vertices v is there a path from s to v? For now, let's consider only undirected graphs; I'll consider directed graphs briefly at the end of this section. For undirected graphs, the vertices reachable from s are precisely the vertices in the same component as s.

Perhaps the most natural reachability algorithm—at least for people like us who are used to thinking recursively—is ***depth-first search***. This algorithm can be written either recursively or iteratively. It's exactly the same algorithm either way; the only difference is that we can actually see the "recursion" stack in the non-recursive version.

```
RecursiveDFS(v):
  if v is unmarked
      mark v
      for each edge vw
          RecursiveDFS(w)
```

```
IterativeDFS(s):
  Push(s)
  while the stack is not empty
      v ← Pop
      if v is unmarked
          mark v
          for each edge vw
              Push(w)
```

Depth-first search is just one (perhaps the most common) species of a general family of graph traversal algorithms that I call ***whatever-first search***. The generic traversal algorithm stores a set of candidate edges in some data structure that I'll call a "bag". The only important properties of a "bag" are that we can put stuff into it and then later take stuff back out. A stack is a particular type of bag, but certainly not the only one. Here is the generic algorithm:

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge vw
              put w into the bag
```

I claim that WhateverFirstSearch marks every node reachable from s and nothing else. The algorithm clearly marks each vertex in G *at most* once. To show that it visits every node in a connected graph *at least* once, we modify the algorithm slightly; the modifications are in bold red. Instead of keeping vertices in the bag, the modified algorithm stores pairs of vertices. This modification allows us to remember, whenever we visit a vertex v for the first time, which previously-visited neighbor vertex put v into the bag. We call this earlier vertex the *parent* of v.

```
WhateverFirstSearch(s):
  put (∅, s) in bag
  while the bag is not empty
      take (p, v) from the bag                  (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge vw                      (†)
              put (v, w) into the bag          (⋆⋆)
```

Lemma 5.1. *WhateverFirstSearch(s) marks every vertex reachable from s and only those vertices. Moreover, the set of all pairs $(v, parent(v))$ with $parent(v) \neq \varnothing$ defines a spanning tree of the component containing s.*

Proof: First we argue that the algorithm marks every vertex v that is reachable from s, by induction on the shortest-path distance from s to v. The algorithm marks s. Let v be any other vertex reachable from s, and let $s \to \cdots \to u \to v$ be any path from s to v with the minimum number of edges. (There must be such a path, because v is reachable from s.) The prefix path $s \to \cdots \to u$ is shorter than the shortest path from s to u, so the inductive hypothesis implies that the algorithm marks u. When the algorithm marks u, it must immediately put the pair (u, v) into the bag, so it must later take (u, v) out of the bag, at which point the algorithm immediately marks v, unless it was already marked.

Every pair $(v, parent(v))$ with $parent(v) \neq \varnothing$ is actually an edge in the underlying graph G. We claim that for any marked vertex v, the path of parent edges $v \rightarrow parent(v) \rightarrow parent(parent(v)) \rightarrow \cdots$ eventually leads back to s; we prove this claim by induction on the order in which vertices are marked. Trivially s is reachable from s, so let v be any other marked vertex. The parent of v must be marked before v is marked, so the inductive hypothesis implies that the parent path $parent(v) \rightarrow parent(parent(v)) \rightarrow \cdots$ leads to s; adding one more parent edge $s \rightarrow parent(s)$ establishes the claim.

The previous claim implies that every vertex marked by the algorithm is reachable from s, and that the set of all parent edges forms a connected graph. Because every marked node except s has a unique parent, the number of parent edges is exactly one less than the number of marked vertices. We conclude that the parent edges form a tree. □

Analysis

The running time of the traversal algorithm depends on what data structure we use for the "bag", but we can make a few general observations. Let T is the time required to insert a single item into the bag or delete a single item from the bag. The for loop (†) is executed exactly once for each marked vertex, and therefore at most V times. Each edge uv in the component of s is put into the bag exactly twice; once as the pair (u, v) and once as the pair (v, u), so line (⋆⋆) is executed at most $2E$ times. Finally, we can't take more things out of the bag than we put in, so line (⋆) is executed at most $2E + 1$ times. Thus, assuming the underlying graph G is stored in a standard adjacency list, WhateverFirstSearch runs in ***$O(V + ET)$ time***. (If G is stored in an adjacency matrix, the running time of WhateverFirstSearch increases to $O(V^2 + ET)$.)

5.6 Important Variants

Stack: Depth-First

If we implement the "bag" using a *stack*, we recover our original depth-first search algorithm. Stacks support insertions (push) and deletions (pop) in $O(1)$ time each, so the algorithm runs in ***$O(V + E)$ time***. The spanning tree formed by the parent edges is called a ***depth-first spanning tree***. The exact shape of the tree depends on the start vertex and on the order that neighbors are visited inside the for loop (†), but in general, depth-first spanning trees are long and skinny. We will consider several important properties and applications of depth-first search in Chapter 6.

Queue: Breadth-First

If we implement the "bag" using a *queue*, we get a different graph-traversal algorithm called ***breadth-first search***. Queues support insertions (push) and deletions (pull) in $O(1)$ time each, so the algorithm runs in $\boldsymbol{O(V+E)}$ ***time***. In this case, the ***breadth-first spanning tree*** formed by the parent edges contains ***shortest paths*** from the start vertex s to every other vertex in its component; we will consider shortest paths in detail in Chapter 8. Again, the exact shape of a breadth-first spanning tree depends on the start vertex and on the order that neighbors are visited in the for loop (†), but in general, breadth-first spanning trees are short and bushy.

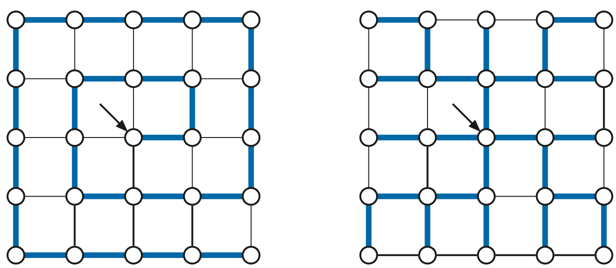


Figure 5.12. A depth-first spanning tree and a breadth-first spanning tree of the same graph, both starting at the center vertex.

Priority Queue: Best-First

Finally, if we implement the "bag" using a *priority queue*, we get yet another family of algorithms called ***best-first search***. Because the priority queue stores at most one copy of each edge, inserting an edge or extracting the minimum-priority edge requires $O(\log E)$ time, which implies that best-first search runs in $\boldsymbol{O(V+E\log E)}$ ***time***.

I describe best-first search as a "family of algorithms", rather than a single algorithm, because there are different methods to assign priorities to the edges, and these choices lead to different algorithmic behavior. I'll describe three well-known variants below, but there are many others. In all three examples, we assume that every edge uv or $u{\rightarrow}v$ in the input graph has a non-negative weight $w(uv)$ or $w(u{\rightarrow}v)$.

First, if the input graph is undirected and we use the weight of each edge as its priority, best-first search constructs the ***minimum spanning tree*** of the component of s. Surprisingly, as long as all the edge weights are distinct, the resulting tree does *not* depend on the start vertex or the order that neighbors are visited; in this case, the minimum spanning tree is actually unique. This instantiation of best-first search is commonly (but, as usual, incorrectly) known as *Prim's algorithm*; we'll discuss this and other minimum-spanning-trees in more detail in Chapter 7.

Define the *length* of a path to be the sum of the weights of its edges. We can also compute ***shortest paths*** in weighted graphs using best-first search, as follows. Every marked vertex v stores a distance $dist(v)$. Initially we set $dist(s) = 0$. For every other vertex v, when we set $parent(v) \leftarrow p$, we also set $dist(v) \leftarrow dist(p) + w(p \rightarrow v)$, and when we insert the edge $v \rightarrow w$ into the priority queue, we use the priority $dist(v) + w(v \rightarrow w)$. Assuming all edge weights are positive, $dist(v)$ is the length of the shortest path from s to v. This instantiation of best-first search is commonly (but, as usual, strictly speaking, incorrectly) known as *Dijkstra's algorithm*; we'll see this algorithm again in Chapter 8.

Finally, define the *width* of a path to be the *minimum* weight of any edge in the path. A simple modification of "Dijkstra's" best-first search algorithm computes ***widest paths*** from s to every other reachable vertex; widest paths are also called ***bottleneck shortest paths***. Every marked vertex v stores a value $width(v)$. Initially we set $width(s) = \infty$. For every other vertex v, when we set $parent(v) \leftarrow p$, we also set $width(v) \leftarrow \min\{width(p), w(p \rightarrow v)\}$, and when we insert the edge $v \rightarrow w$ into the priority queue, we use the priority $\min\{width(v), w(v \rightarrow w)\}$. Widest paths are useful in algorithms for computing *maximum flows*, which (you guessed it) we'll consider in Chapter 10.

Disconnected Graphs

WhateverFirstSearch(s) only visits the vertices reachable from a single start vertex s. To visit *every* vertex in G, we can use the following simple "wrapper" function.

```
WFSAll(G):
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            WhateverFirstSearch(v)
```

Wait, I hear you ask, why are you making this so complicated? Why not just[10] scan the vertex array?

```
MarkEveryVertexDuh(G):
    for all vertices v
        mark v
```

[10]This word is almost always a signal that you are missing something important.

Well, sure, *if* you have an complete list of vertices, then you can do that, but remember that not all graphs are represented so explicitly.[11] More importantly, even if we do have an explicit vertex list, the order in which this naive algorithm visits vertices is determined by their order in the *data structure*, not by the abstract structure of the graph.

In particular, unlike a naive scan through the vertices, WFSAll visits all the vertices in one component, and then all the vertices in the next component, and so on through each component of the input graph. This component-by-component traversal allows us, for example, to count the components of a disconnected graph using a single counter.

```
CountComponents(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          WhateverFirstSearch(v)
  return count
```

With just a bit more work, we can record which component contains each vertex, instead of merely marking it.

```
CountAndLabel(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          LabelOne(v, count)
  return count
```

```
⟨⟨Label one component⟩⟩
LabelOne(v, count):
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          comp(v) ← count
          for each edge vw
              put w into the bag
```

WFSAll marks every vertex once, puts every edge into the bag once, and takes every edge out of the bag once, so the overall running time is $O(V + ET)$, where T is the time for a bag operation. In particular, if we run depth-first search or breadth-first search at every vertex, the resulting algorithm still requires only $O(V + E)$ time.

Moreover, because WhateverFirstSearch computes a spanning tree of one component, we can use WFSAll to compute a spanning *forest* of the entire

[11] On the other hand, if we store a time-stamp at every vertex indicating the last time it was "marked", then we can "unmark every vertex" in $O(1)$ time by recording the start time of our traversal, and considering a vertex "marked" if its time stamp is later than the recorded start time.

graph. In particular, best-first search with edge weights as priorities computes the minimum-weight spanning forest in $O(V + E \log E)$.

Shockingly, at least one *extremely* popular algorithms textbook claims that this wrapper can only be used with depth-first search.[12] This claim is flatly incorrect. In fact, the *very first* implementation of breadth-first search, written around 1945 by Konrad Zuse in his proto-language *Plankalkül*, was developed for the specific purpose of counting and labeling the components of an undirected graph.

Directed Graphs

Whatever-first search is easy to adapt to directed graphs; the only difference is that when we mark a vertex, we put all of its *out*-neighbors into the bag. In fact, if we are using standard adjacency lists or adjacency matrices, we do not have to change the code at all!

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge v→w
              put w into the bag
```

Our earlier proof implies that the algorithm marks every vertex reachable from s, and the directed edges $parent(v){\rightarrow}p$ define a rooted tree, with all edges directed away from the root s. However, even if the graph is connected, we no longer necessarily obtain a *spanning* tree of the graph, because reachability is no longer symmetric.

On the gripping hand, WhateverFirstSearch does define a spanning tree of the vertices reachable from s. Moreover, by varying the instantiation of the "bag", we can obtain a depth-first spanning tree, a breadth-first spanning tree, a minimum-weight directed spanning tree, a shortest-path tree, or a widest-path tree of those reachable vertices.

5.7 Graph Reductions: Flood Fill

One of the earliest modern examples of whatever-first search was proposed by Edward Moore in the mid-1950s. A *pixel map* is a two-dimensional array

[12] To quote directly: "Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may repeat from multiple sources."

whose value represent colors; the individual entries in the array are called *pixels*, an abbreviation of *picture elements*.[13] A *connected region* in a pixel map is a connected subset of pixels that all have the same color, where two pixels are considered adjacent if they are immediate horizontal or vertical neighbors. The *flood fill* operation, commonly represented by a paint can in raster-graphics editing software, changes every pixel in a connected region to a new color; the input to the operation consists of the indices i and j of one pixel in the target region and the new color.

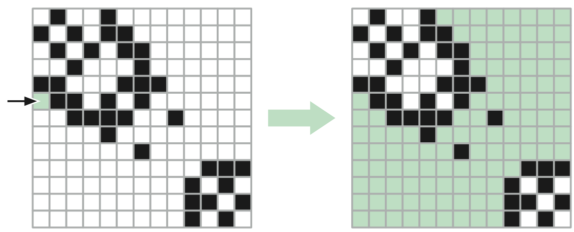


Figure 5.13. An example of flood fill

The flood-fill problem can be reduced to the reachability problem by chasing the definitions. We define an undirected graph $G = (V, E)$, whose vertices are the individual pixels, and whose edges connect neighboring pixels with the same color. Each connected region in the pixel map is a component of G; thus, the flood-fill problem ***reduces*** to a reachability problem in G. We can solve this reachability problem using whatever-first search in G, starting at the given pixel (i, j), with one minor modification; whenever we mark a vertex, we immediately change its color. For an $n \times n$ pixel map, the graph G has n^2 vertices and at most $2n^2$ edges, so whatever-first search runs in $O(V + E) = \boldsymbol{O(n^2)}$ ***time***.

This simple example demonstrates the essential ingredients of a ***reduction***. Rather than solving the flood-fill problem from scratch, we use an existing algorithm as a black-box subroutine. *How* whatever-first search works is utterly irrelevant here; all that matters is its *specification*: Given a graph G and a starting vertex s, mark every vertex in G that is reachable from s. Like any other subroutine, we still have to describe how to construct the input and how to use its output. We also have to analyze *our* resulting algorithm in terms of *our* input parameters, not the vertices and edges of whatever intermediate graph our algorithm constructs.

Now that we have an algorithm that works—but *only* now—we can apply two easy optimizations to make it faster, one practical and the other theoretical:

[13]Before the advent of modern raster display devices in the 1960s, pixels were more commonly known as *stitches* or *tesserae*, depending on whether they were made of thread or very small rocks. The word *pix* became a standard abbreviation for *picture(s)* in the early 20th century—not long after *sox* became a common plural of *sock*—supplanting the earlier colloquialism *piccy*. See also voxel (volume element), texel (texture element), and taxel (tactile element and/or badger).

- In an actual implementation, we would not actually build a separate graph data structure for G. Instead, we can use the pixel map directly *as though* it were a standard adjacency list, because we can list the same-color neighbors of any pixel in $O(1)$ time each. In particular, there is no need to separately "mark" vertices; we can use the color of the pixels instead.
- More careful analysis implies that the running time is proportional to the number of pixels in the region being filled—equivalently, the number of vertices in component of G containing vertex (i, j)—which could be considerably smaller than $O(n^2)$.

Exercises

Graphs

1. Prove that the following definitions are all equivalent.
   - A tree is a connected acyclic graph.
   - A tree is one component of a forest. (A forest is an acyclic graph.)
   - A tree is a connected graph with *at most* $V-1$ edges.
   - A tree is a minimally connected graph; removing any edge disconnects the graph.
   - A tree is an acyclic graph with *at least* $V-1$ edges.
   - A tree is a maximally acyclic graph; adding an edge between any two vertices creates a cycle.
   - A tree is a graph that contains a unique path between each pair of vertices.

2. Prove that any connected acyclic graph with $n \geq 2$ vertices has at least two vertices with degree 1. Do not use the words "tree" or "leaf", or any well-known properties of trees; your proof should follow entirely from the definitions of "connected" and "acyclic".

3. A graph (V, E) is *bipartite* if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.
   (a) Prove that every tree is a bipartite graph.
   (b) Prove that a graph G is bipartite if and only if every cycle in G has an even number of edges.
   (c) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

4. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon i pecks pigeon j, which pecks pigeon k, which pecks pigeon ℓ, which pecks pigeon i.

   (a) Prove that any finite population of pigeons can be placed in a procession (perhaps a parade?) so that each pigeon pecks the preceding pigeon's posterior. Pretty please.

   (b) Suppose you are given a directed graph representing the pecking relationships among a set of n pigeons. The graph contains one vertex per pigeon, and it contains an edge $i \to j$ if and only if pigeon i pecks pigeon j. Describe and analyze an algorithm to compute a pecking order for the pigeons, as guaranteed by part (a).

   (c) Prove that for any set of at least three pigeons, either the pecking order described in part (a) is unique, or there are three pigeons i, j, and k, such that pigeon i pecks pigeon j, which pecks pigeon k, which pecks pigeon i.

5. An ***Euler tour*** of a graph G is a closed walk through G that traverses every edge of G exactly once.

   (a) Prove that if a connected graph G has an Euler tour, then every vertex in G has even degree. (Euler proved this.)

   (b) Prove that if every vertex in a connected graph G has even degree, then G has an Euler tour. (Euler did *not* prove this.)

   (c) Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such tour exists. (Euler vaguely waved his hands at this.)

6. The d-dimensional hypercube is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.

   (a) A Hamiltonian cycle in a graph G is a cycle of edges in G that enters each vertex of G exactly once. Prove that for all $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

   (b) Which hypercubes have an Euler tour (a closed walk that traverses every edge exactly once)? *[Hint: This is very easy.]*

Traversal Algorithms

7. Recall that a directed graph G is *strongly connected* if, for any two vertices u and v, there is a path in G from u to v and a path in G from v to u.

   Describe an algorithm to determine, given an *undirected* graph G as input, whether it is possible to direct each edge of G so that the resulting directed graph is strongly connected.

8. Let G be a connected graph, and let T be a depth-first spanning tree of G rooted at some node v. Prove that if T is also a breadth-first spanning tree of G rooted at v, then $G = T$.

9. Professors Epprich and Goodstein propose the following optimization of the generic whatever-first search algorithm. Instead of checking whether the vertices we take out of the bag are marked, their algorithm checks before it even puts the vertex into the bag, thereby ensuring that each vertex is put into the bag at most once. Their algorithm also assigns the parent of each vertex when that vertex is marked.

```
EAGERWFS(s):
  mark s
  put s into the bag
  while the bag is not empty
      take v from the bag
      for each edge vw
          if w is unmarked
              mark w
              parent(w) ← v
              put w into the bag
```

   (a) Prove that EagerWFS(s) marks every node reachable from s and nothing else. Equivalently, prove that the parent edges $v \to parent(v)$ computed by EagerWFS(s) define a spanning tree of the component containing s.

   (b) Prove that if the bag is implemented as a queue, EagerWFS is equivalent to breadth-first search, meaning the two algorithms mark the same vertices in the same order and construct the same spanning tree. *[Hint: What is the definition of a queue?]*

   (c) Prove that EagerWFS is *never* equivalent to depth-first search, no matter what data structure is used as the bag (and thus, in particular, when the bag is a stack).

   Neither EagerWFS nor RecursiveDFS specify the order that edges vw at each vertex v are considered, and different edge orders may lead to different spanning trees. Thus, you need to argue, for some explicit graph G, that no spanning tree of G produced by RecursiveDFS can be constructed by EagerWFS (using *any* bag data structure), or vice versa.

10. One of the earliest published descriptions of whatever-first search as a generic class of algorithms was by Edsger Dijkstra, Leslie Lamport, Alain Martin, Carel Scholten, and Elisabeth Steffens in 1975, as part of the design of an automatic garbage collector. Instead of maintaining marked and unmarked vertices, their algorithm maintains a color for each vertex, which is either white, gray, or black. As usual, in the following algorithm, we imagine a fixed underlying graph G.

```
ThreeColorSearch(s):
  color all nodes white
  color s gray
  while at least one vertex is gray
      ThreeColorStep( )
```

```
ThreeColorStep( ):
  v ← any gray vertex
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
```

(a) Prove that ThreeColorSearch maintains the following invariant at all times: No black vertex is a neighbor of a white vertex. *[Hint: This should be easy.]*

(b) Prove that after ThreeColorSearch(s) terminates, all vertices reachable from s are black, all vertices not reachable from s are white, and that the parent edges $v \to parent(v)$ define a rooted spanning tree of the component containing s.

[Hint: Intuitively, black nodes are "marked" and gray nodes are "in the bag". Unlike our formulation of WhateverFirstSearch, however, the three-color algorithm is not required to process all edges out of a node at the same time.]

(c) Prove that the following variant of ThreeColorSearch, which maintains the set of gray vertices in a standard stack, is equivalent to depth-first search. *[Hint: The order of the last two lines of ThreeColorStackStep matters!]*

```
ThreeColorStackSearch(s):
  color all nodes white
  color s gray
  push s onto the stack
  while at least one vertex is gray
      ThreeColorStackStep( )
```

```
ThreeColorStackStep( ):
  pop v from the stack
  if v has no white neighbors
      color v black
  else
      w ← any white neighbor of v
      parent(w) ← v
      color w gray
      push v onto the stack
      push w onto the stack
```

(d) Prove that the following variant of ThreeColorSearch, which maintains the set of gray vertices in a standard queue, is ***not*** equivalent to breadth-first search. *[Hint: The order of the last two lines of* THREECOLORQUEUESTEP *doesn't matter!]*

```
THREECOLORQUEUESEARCH(s):
    color all nodes white
    color s gray
    push s into the queue
    while at least one vertex is gray
        THREECOLORQUEUESTEP()
```

```
THREECOLORQUEUESTEP():
    pull v from the queue
    if v has no white neighbors
        color v black
    else
        w ← any white neighbor of v
        parent(w) ← v
        color w gray
        push v into the queue
        push w into the queue
```

♥(e) Now suppose that another process is adding edges to G while THREECOLORSEARCH is running. These new edges could violate the color invariant described in part (a) and therefore destroy the correctness of the algorithm—in particular, when THREECOLORSEARCH terminates, some vertices reachable from s could be white. This would be disastrous if we are relying on "white" to mean "unreachable and therefore safe to delete".

However, if the other process explicitly preserves the color invariant, we can still use the three-color algorithm to safely identify unreachable vertices. We model the two concurrent algorithms as follows; the either/or choice in GARBAGECOLLECT and the choice of which vertices u and w to MUTATE are entirely out of the main algorithm's control.[14]

```
GARBAGECOLLECT(s):
    color all vertices white
    color s gray
    while at least one vertex is gray
        either
            COLLECTSTEP()
        or
            MUTATE()
```

```
COLLECTSTEP():
    v ← any gray vertex
    if v has no white neighbors
        color v black
    else
        w ← any white neighbor of v
        color w gray
```

[14]This is a *dramatic* oversimplification of the "mark and sweep" garbage-collection algorithms actually used in multi-threaded languages like Lua and Go. A more thorough discussion of multi-threaded dynamic memory management is unfortunately beyond the scope of this book, except for the First Commandment: **Thou Shalt Not Roll Thine Own Garbage Collector.**

```
MUTATE():
  u ← any vertex
  w ← any vertex
  if uw is not an edge
      add edge uw
      if u is black and w is white
          color u gray
      if u is white and w is black
          color w gray
```

Prove that GARBAGECOLLECT eventually terminates with every vertex reachable from s colored black and every vertex not reachable from s colored white.

♥(f) Suppose instead of recoloring black vertices gray, MUTATE maintains the color invariant by coloring some *white* vertices gray:

```
MUTATE():
  u ← any vertex
  w ← any vertex
  if uw is not an edge
      add edge uw
      if u is black and w is white
          color w gray
      if u is white and w is black
          color u gray
```

Prove that GARBAGECOLLECT eventually terminates with s colored black, every vertex reachable from a black vertex colored black, and every vertex not reachable from a black vertex colored white.

Reductions

11. A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

    Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the number maze in Figure 5.14, your algorithm should return the integer 8.

12. ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares,

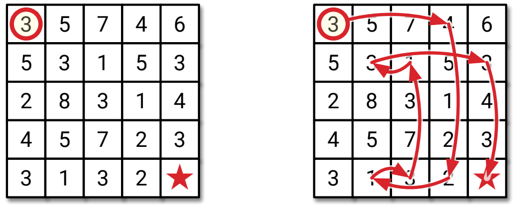


Figure 5.14. A 5 × 5 number maze that can be solved in eight moves.

numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either “snakes” (leading down) or “ladders” (leading up). Each square can be an endpoint of at most one snake or ladder.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k. If the token ends the move at the *top* end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the *bottom* end of a ladder, it climbs up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

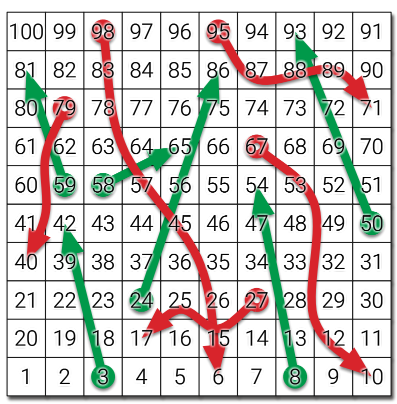


Figure 5.15. A Snakes and Ladders board. Upward straight arrows are ladders; downward wavy arrows are snakes.

13. The infamous Mongolian puzzle-warrior Vidrach Itky Leda invented the following puzzle in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

In a single turn, you may move either token up, right, down, or left *by a distance determined by the **other** token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move either token off the grid, and at the end of a move the two tokens cannot lie on the same square.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle in Figure 5.16, your algorithm would return the number 5.

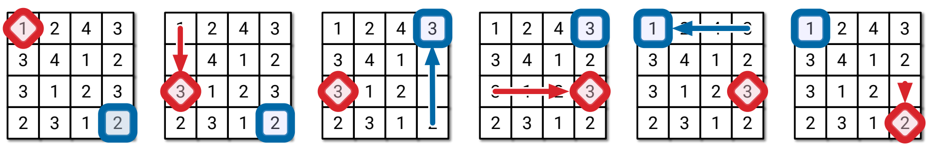
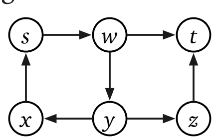


Figure 5.16. A five-move solution for a 4×4 Vidrach Itky Leda puzzle.

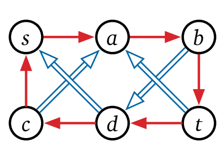
14. Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is divisible by 3.

    For example, given the graph shown below, with the indicated vertices s and t, your algorithm should return True, because the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.



15. Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red.

    For example, given the following graph as input, your algorithm should return the integer 7, because $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to t$ is the shortest legal walk from s to t.



16. Consider a directed graph G, where each edge is colored either red, white, or blue. A walk in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \cdots \to v_k$ is a French flag walk if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

    Describe an algorithm to find all vertices in G that can be reached from a given vertex v through a French flag walk.

17. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way e has an associated cost of $c(e)$ dollars, where $c(e)$ is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

    Judy wants to travel from galaxy s to galaxy t, but teleportation is not very pleasant and she would like to minimize the number of times she needs to teleport. However, she wants the total cost to be a multiple of five dollars, because carrying small change is not pleasant either.

    (a) Describe and analyze an algorithm to compute the smallest number of times Judy needs to teleport to travel from galaxy s to galaxy t so that the total cost is a multiple of five dollars.

    (b) Solve part (a), but now assume that Judy has a coupon that allows her to use exactly one teleport-way for free.

18. Three Seashells is a solitaire game, played on a connected undirected graph G. Initially, three tokens are placed on distinct start vertices a, b, c. In each turn, you *must* move *all three* tokens, by moving each token along an edge from its current vertex to an adjacent vertex. At the end of each turn, the three tokens *must* lie on three different vertices. Your goal is to move the tokens onto three goal vertices x, y, z; it does not matter which token ends up on which goal vertex.

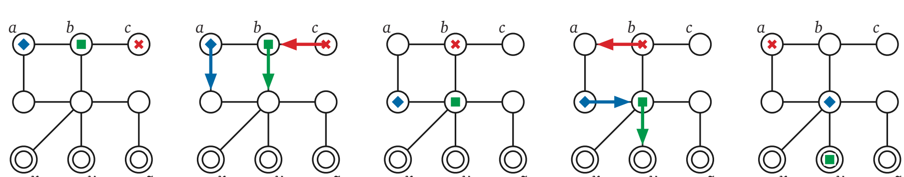


Figure 5.17. The initial configuration of the Three Seashells puzzle and the first two turns of a solution.

    Describe and analyze an algorithm to determine whether a given Three Seashells puzzle is solvable. Your input consists of the graph G, the start vertices a, b, c, and the goal vertices x, y, z. Your output is a single bit: TRUE or FALSE.

19. Let G be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G, and we want to move the coins so that they lie on the same vertex using as few moves as possible. At every step, each coin *must* move to an adjacent vertex.

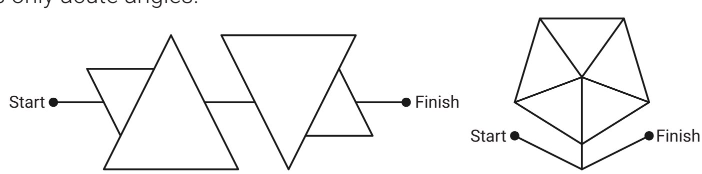
    (a) Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable. The input to your algorithm consists of a graph $G = (V, E)$ and two vertices $u, v \in V$ (which may or may not be distinct).

    (b) Now suppose there are three coins. Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable.

    (c) Finally, suppose there are *forty-two* coins. Describe and analyze an algorithm to determine whether it is possible to move all 42 coins to the same vertex. Again, *every* coin must move at *every* step. For full credit, your algorithm should run in $O(V + E)$ time.

20. One of my daughter's elementary-school math workbooks[15] contains several puzzles of the following type:

    > Complete each angle maze below by tracing a path from start to finish that has only acute angles.



Describe and analyze an algorithm to solve arbitrary acute-angle mazes.

You are given a connected undirected graph G, whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges, and the first maze above has 18 vertices and 21 edges. You are also given two vertices Start and Finish.

Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Formally, a walk through G is valid if, for any two consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = \pi$ or $0 < \angle uvw < \pi/2$. Assume you have a subroutine that can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

[15] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See https://www.beastacademy.com/resources/printables.php for several more examples.

21. Suppose you are given a set of n horizontal and vertical line segments and two points s and t in the plane. Describe an efficient algorithm to determine if there is a path from s to t that does not intersect any of the given line segments.

Each horizontal line segment is specified by its left and right x-coordinates and its unique y-coordinate; similarly, each vertical line segment is specified by its unique x-coordinate and its top and bottom y-coordinates. Finally, the points s and t are each specified by their x- and y-coordinates.

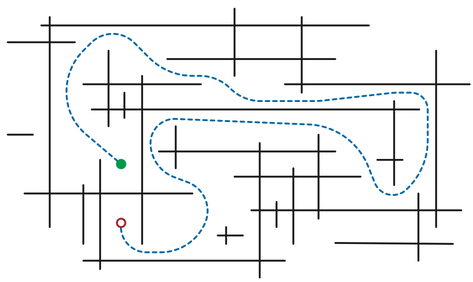


Figure 5.18. A path between two points in a maze of horizontal and vertical line segments.

22. Every cheesy romance movie has a scene where the romantic couple, after a long and frustrating separation, suddenly see each other across a long distance, and then slowly approach one another with unwavering eye contact as the music rolls in and the rain lifts and the sun shines through the clouds and the music swells and everyone starts dancing with rainbows and kittens and chocolate unicorns and. . . .[16]

Suppose a romantic couple—in grand computer science tradition, named Alice and Bob—enters their favorite park at the east and west entrances and immediately establish eye-contact. They can't just run directly to each other; instead, they must stay on the path that zig-zags through the park between the east and west entrances. To maintain the proper dramatic tension, Alice and Bob must traverse the path so that they always lie on a direct east-west line.

We can describe the zigzag path as two arrays $X[0..n]$ and $Y[0..n]$, containing the x- and y-coordinates of the corners of the path, in order from the southwest endpoint to the southeast endpoint. The X array is sorted in increasing order, and $Y[0] = Y[n]$. The path is a sequence of straight line segments connecting these corners.

(a) Suppose $Y[0] = Y[n] = 0$ and $Y[i] > 0$ for every other index i; that is, the endpoints of the path are strictly below every other point on the path.

[16] Fun fact: Damien Chazelle, the director of *Whiplash* and *La La Land*, is the son of Princeton computer science professor and electric guitarist Bernard Chazelle.

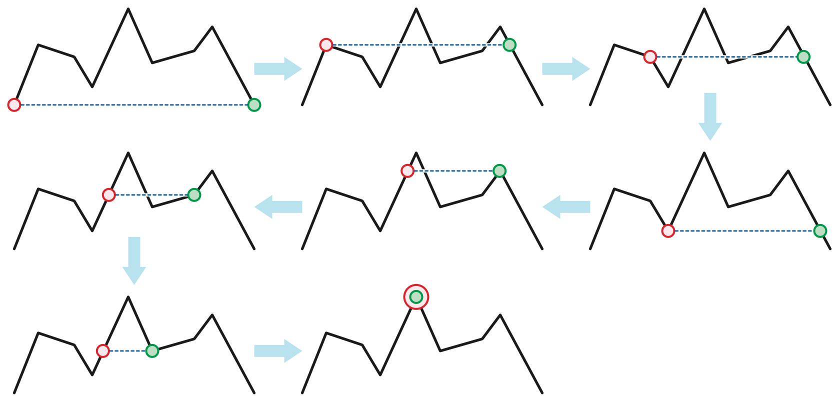


Figure 5.19. Alice and Bob meet. Alice walks backward in step 2, and Bob walks backward in steps 5 and 6.

Prove that for any path P meeting these conditions, Alice and Bob can *always* meet. *[Hint: Describe a graph that models all possible locations of the couple along the path. What are the vertices of this graph? What are the edges? Use the* ***Handshake Lemma****: Every graph has an even number of vertices with odd degree.]*

(b) If the endpoints of the path are *not* below every other vertex, Alice and Bob might still be able to meet, or they might not. Describe an algorithm to decide whether Alice and Bob can meet, without either breaking east-west eye contact or stepping off the path, given the arrays $X[0..n]$ and $Y[0..n]$ as input.

♥(c) Describe an algorithm for part (b) that runs in $O(n)$ time.

23. The famous puzzle-maker Kaniel the Dane invented a solitaire game played with two tokens on an $n \times n$ square grid. Some squares of the grid are marked as *obstacles*, and one grid square is marked as the *target*. In each turn, the player must move one of the tokens from is current position *as far as possible* upward, downward, right, or left, stopping just before the token hits (1) the edge of the board, (2) an obstacle square, or (3) the other token. The goal is to move either of the tokens onto the target square.

For example, we can solve the puzzle shown in Figure 5.20 by moving the red token down until it hits the obstacle, then moving the green token left until it hits the red token, and then moving the red token left, down, right, and up. The red token stops at the target on the 6th move *because* the green token is just above the target square.

Describe and analyze an algorithm to determine whether an instance of this puzzle is solvable. Your input consist of the integer n, a list of obstacle locations, the target location, and the initial locations of the tokens. The

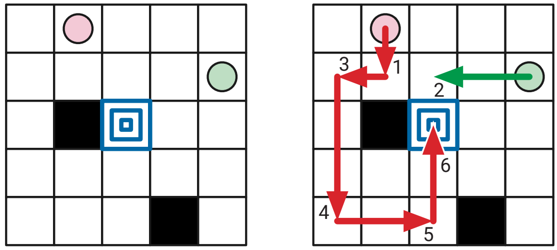


Figure 5.20. An instance of Kaniel the Dane's puzzle that can be solved in six moves. Circles indicate initial token positions; black squares are obstacles; the center square is the target.

output of your algorithm is a single boolean: TRUE if the given puzzle is solvable and FALSE otherwise. *[Hint: Don't forget about the time required to construct the graph.]*

♥24. ***Rectangle Walk*** is a new abstract puzzle game, available for only 99¢ on Steam, iOS, Android, Xbox One, Playstation 5, Nintendo Wii U, Atari 2600, Palm Pilot, Commodore 64, TRS-80, Sinclair ZX-1, DEC PDP-8, PLATO, Zuse Z3, Duramesc, Odhner Arithmometer, Analytical Engine, Jacquard Loom, Horologium Mirabile Lundense, Leibniz Stepped Reckoner, Al-Jazari's Robot Band, Yan Shi's Automaton, Antikythera Mechanism, Knotted Rope, Ishango Bone, and Pile of Rocks.

The game is played on an $n \times n$ grid of black and white squares. The player moves a rectangle through this grid, subject to the following conditions:

- The rectangle must be aligned with the grid; that is, the top, bottom, left, and right coordinates must be integers.
- The rectangle must fit within the $n \times n$ grid, and it must contain at least one grid cell.
- The rectangle must not contain a black square.
- In a single move, the player can replace the current rectangle r with any rectangle r' that either contains r or is contained in r.

Initially, the player's rectangle is a 1×1 square in the upper right corner. The player's goal is to reach a 1×1 square in the bottom left corner using as few moves as possible.

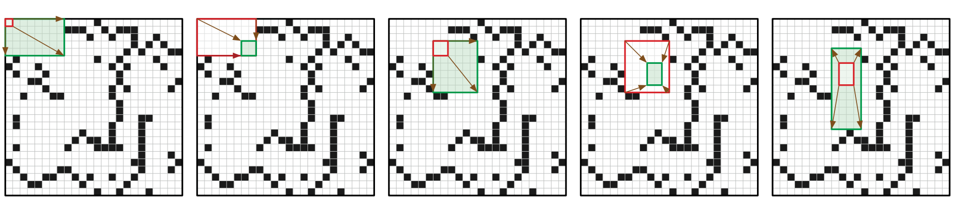


Figure 5.21. The first five steps of a Rectangle Walk.

Describe and analyze an algorithm to compute the length of the shortest Rectangle Walk in a given bitmap. Your input is an array $M[1..n, 1..n]$, where $M[i,j] = 1$ indicates a black square and $M[i,j] = 0$ indicates a white square. Assume that a valid rectangle walk exists; in particular, $M[1,1] = M[n,n] = 0$. For example, given the bitmap shown above, your algorithm should return the integer 18. *[Hint: Don't forget about the time required to construct the graph!!]*

25. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[17] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$

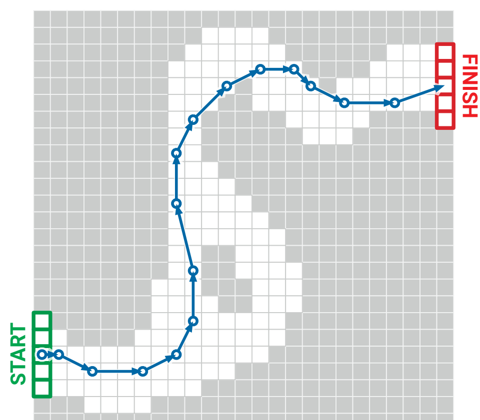


Figure 5.22. A 16-step Racetrack run, on a 25 × 25 track. This is *not* the shortest run on this track.

Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0,0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position inside the finishing area.

[17]The actual game is a bit more complicated than the version described here. See http://harmmade.com/vectorracer/ for an excellent online version.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the "starting area" is the first column, and the "finishing area" is the last column.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack.

26. A *rolling die maze* is a puzzle involving a standard six-sided die (a cube with numbers on each side) and a grid of squares. You should imagine the grid lying on a table; the die always rests on and exactly covers one square of the grid. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.

    Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

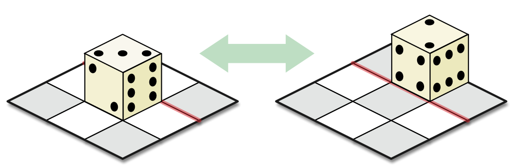


Figure 5.23. Rolling a die

Figure 5.24 shows four rolling die mazes. Assuming we use a standard die with 1 and 6 on opposite sides, only the first two mazes are solvable. For example, the first maze is solvable by by placing the die on the lower left square with 1 on the top face, and then rolling the die east, then north, then north, then east.

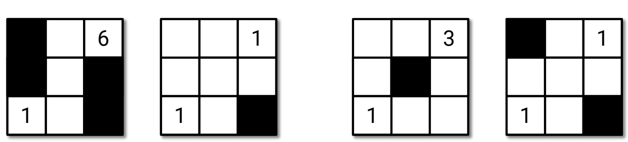


Figure 5.24. Four rolling die mazes; only the first two are solvable.

(a) Suppose the input is a two-dimensional array $L[1..n, 1..n]$, where each entry $L[i, j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked.

Describe and analyze a polynomial-time algorithm to determine whether the given rolling die maze is solvable.

♥(b) Now suppose the maze is specified *implicitly* by a list of labeled and blocked squares. Specifically, suppose the input consists of an integer M, specifying the height and width of the maze, and an array $S[1..n]$, where each entry $S[i]$ is a triple (x, y, L) indicating that square (x, y) has label L. As in the explicit encoding, label -1 indicates that the square is blocked; free squares are not listed in S at all. Describe and analyze an efficient algorithm to determine whether the given rolling die maze is solvable. For full credit, the running time of your algorithm should be polynomial in the input size n.

[Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can be solved only if the initial position is chosen correctly.]

♥27. Suppose you are given an arbitrary directed graph G in which each edge is colored either red or blue, along with two special vertices s and t.

(a) Describe an algorithm that either computes a walk from s to t such that the pattern of red and blue edges along the walk is a palindrome, or correctly reports that no such walk exists.

(b) Describe an algorithm that either computes the *shortest* walk from s to t such that the pattern of red and blue edges along the walk is a palindrome, or correctly reports that no such walk exists.

[Hint: Where did we last see palindromes?]

♠♥28. Draughts, also known in the United States as "checkers", is a game played on an $m \times m$ grid of squares, alternately colored light and dark.[18] The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size. Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player ("White") moves the white pieces; the other ("Black") moves the black pieces. A player loses when her last piece is taken off the board.

[18]The counting tables used by medieval English government accountants were covered by a green cloth with black squares in a checker pattern; disk-shaped counters were placed in these squares to represent values. For this reason, the British government's accountants have been collectively known since the 10th century as the *Exchequer*. The actual counting tables were used by the Exchequer to tally tax payments well into the 19th century.

Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[19] Pieces can be moved in any of the four diagonal directions. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square, or makes a series of *jumps* with one of her pieces. In each jump, the piece moves to an empty square two steps away in any diagonal direction, but only if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. All jumps in the same turn must be made with the same piece.

Describe an algorithm to decide whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board (m), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where n is the total number of pieces. *[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does **not** always find a winning sequence of jumps even when one exists. See problem 5. Parity, parity, parity.]*

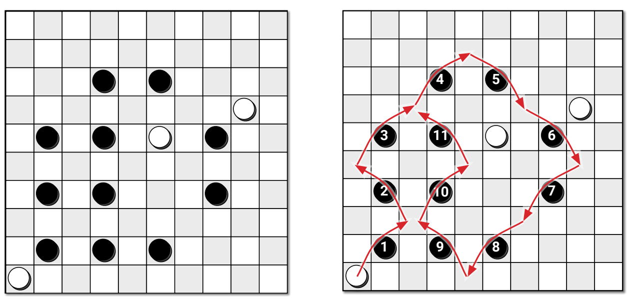


Figure 5.25. White wins in one turn.

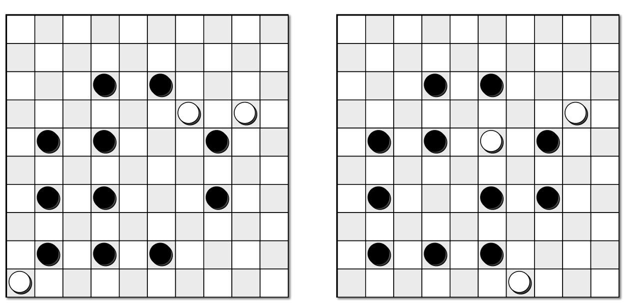


Figure 5.26. White cannot win in one turn from either of these positions.

[19]Most other variants of draughts have "flying kings", which behave *very* differently than kings in the British/American game, and which make this problem *much* more difficult, as we will see in Chapter 12.