Mille viae ducunt homines per saecula Romam.
[A thousand roads lead men forever to Rome.]

— Alain de Lille *Liber Parabolarum* (1175)

I study my Bible as I gather apples.
First I shake the whole tree, that the ripest might fall.
Then I climb the tree and shake each limb,
and then each branch and then each twig,
and then I look under each leaf.

— attributed to Martin Luther (c. 1500)

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, for the solution is based on reason alone, and its discovery does not depend on any mathematical principle. Because of this, I do not know why even questions which bear so little relationship to mathematics are solved more quickly by mathematicians than by others.

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

CHAPTER 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 (more rarely) ***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 400000km of public roads across Europe, western and central Asia, and northern Africa during the height of the Roman empire. Travelers would carry *itinerariae*, which were either maps or simple lists of the cities, milestones, and branches along local roads. The largest surviving *itineraria* is the *Tabula*

Peutingeriana, a 13th-century scroll copy of a 5th-century revision of a 1st-century survey of the Roman *cursus publicus*, stretching from Scotland to southern India, commissioned during the reign of Augustus Caesar. The *Tabula Peutingeriana* is not a geographically accurate map, 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.

Figure 5.1. A small excerpt of Konrad Miller's 1872 restoriation 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 (*Matricorum*, bottom center). The name *Parisi* in red refers erroneously to the Celtic tribe; in fact the Parisii territory was nowhere near *Veteribus*.

Graphs are also used in astronomical charts to indicate constellations and other structures. One of the oldest known examples of such a chart is a scroll drawn during the early Tang Dynasty circa 650AD and discovered in 1907 in Dunhuang, a town on the ancient Silk Road in northern China. The Dunhuang star chart indicates the positions and magnitudes of 1339 stars, grouped into 257 asterisms (constellations), with each asterism indicated by lines joining nearby stars. Thus, each component of the graph is an asterism. A similar but less detailed star chart, discovered only in 1998, was painted on the wall of the Kitora Tomb in Asuka, Japan, in the late 7th or early 8th century.In both of these maps, the graph structure is imposed primarily as a mnemonic device; the components of the graph *are* the asterisms.[1]

Perhaps the oldest classical use of graphs—and specifically trees—is in representing genealogies. One of the oldest examples is the Tree of Jesse, which traces the patrilineal line of descent from Jesse, the father of King David, to Jesus in the books of Matthew and Luke. This lineage is illustrated as a tree (a literal plant, with leaves and branches) in hundreds of bibles, psalters, paintings, stained glass windows, and stone carvings, starting as early as the 11th century.

More complex family "trees" have been used for centuries to settle legal questions about marriage, inheritance, and royal succession. For example, civil law in the Roman

[1]The use of graphs to indicate constellations is a relatively modern practice in Western astronomy. Medieval and Renaissance European star charts indicated constellations by overlaying a realistic mnemonic image over each constellation—a bear for Ursa Major, a swan for Cygnus, a hunter for Orion, and so on.

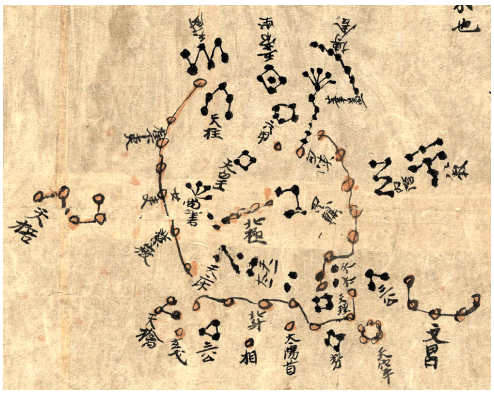


Figure 5.2. An exceprt of the Dunhuang star chart, showing constellations near the north celestial pole. The Northern Dipper—known in America as the Big Dipper—can be seen near the bottom of the image.

empire, later adopted as anon 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.[2]

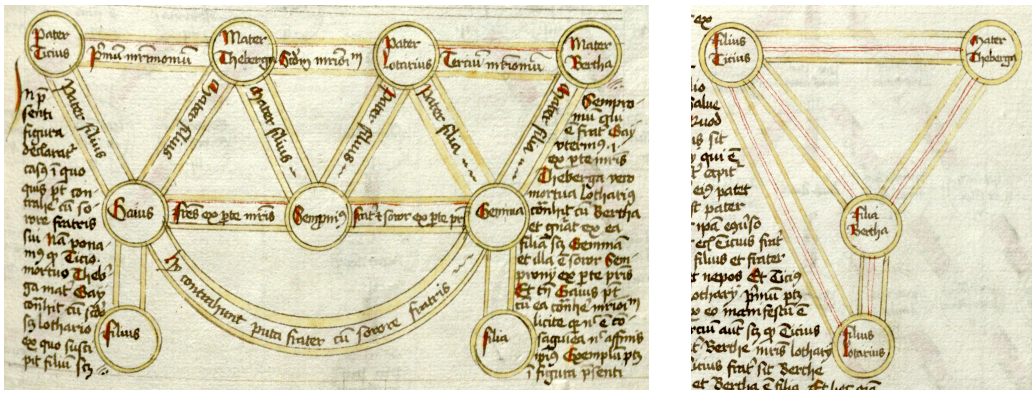


Figure 5.3. Two diagrams describing a complex marriage case, from an anonymous 15th-century treatise on Johannes Andreae's *Super arboribus consanguinitatis et affinitatis*, an early 14th-century treatise on canon law. From a gallery of "Legal Trees" published by the Yale Law Library under a Creative Commons Licence.

[2]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.

Of course, there are many other familiar examples of graphs, like mazes (introduced in their modern form by Giovanni Fontana circa 1420), Leonard Euler's well-known partial[3] solution to the Bridges of Königsburg puzzle (1735) or his less well-known solution the knight's tour problem (1759), 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, Kirchoff, 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 Pierce in 1886, and cast into their modern form by Claude Shannon in 1937), and yeah, okay, if you insist, the internet.

The word "graph" for the abstract mathematical was coined by J. J. 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 Kirchoff and Karl von Staudt ten years earlier. The first textbook on graph theory was published by Dénes Kőnig in 1936.

5.2 Vocabulistics

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***[4] or ***nodes***, and E is a set of pairs of elements of V, which we call ***edges*** or (more rarely) ***arcs***. 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.

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 between 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.

[3] Euler dismissed the final step of his proof—actually finding an Euler tour of a graph when every vertex has even degree—as obvious. In fact, Euler missed the requirement that the graph must be connected.

[4] The singular of the English word "vertices" is ***vertex***. Simiarly, the singular of "matrices" is ***matrix***, and the singular of "indices" is ***index***. Unless you're speaking Italian, there is no such 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! You *will* lose points for using any of these so-called words. If you have trouble remembering this rule, just use the word "node".

(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. 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*** or ***in-neighbor*** of v and v a ***successor*** or ***out-neighbor*** of u. The ***in-degree*** of a node is the number of predecessors; the ***out-degree*** is the 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 is a sequence of edges, where each successive pair of edges shares one vertex. 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. A disconnected graph consists of several ***components***, which are its maximal connected subgraphs; two vertices are in the same component if and only if there is a path between them.[5]

A ***cycle*** is a path that starts and ends at the same vertex and has at least one edge. 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 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 connected component of G.

We require slightly different definitions for directed graphs. A ***directed walk*** is a sequence of *directed* edges, where the head of each edge is the tail of the next; a ***directed path*** is a directed walk without repeated vertices. 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 Abstract 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

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

vertices. A graph is ***planar*** if it has a drawing where no two edges cross; such a drawing is also called an ***embedding***.[6] 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!

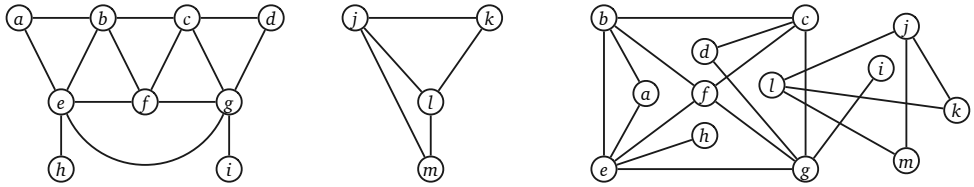


Figure 5.4. Two drawings of the same disconnected planar graph with 13 vertices, 19 edges, and two components.

However, drawings are not the only useful representation of graphs. For example, the ***intersection graph*** of a collection of objects has a node for every object and an edge for every intersecting pair. 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 are intervals on the real line, with an edge between any two intervals that overlap.

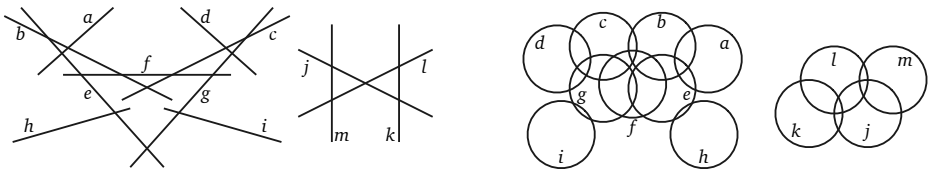


Figure 5.5. The graph in Figure 5.4 is also the intersection graph of (a) a set of line segments and (b) a set of circles.

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 $0, 1, 2, \ldots, n$, and the edges are the pairs $(i-1) \to i$ and $(i-2) \to i$ for every integer i between 2 and n.

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

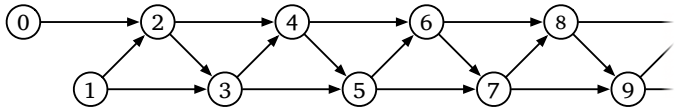


Figure 5.6. The dependency graph of the Fibonacci recurrence.

As a more complex example, consider the following recurrence, which solves a certain sequence-alignment problem called *edit distance*; see the dynamic programming notes for details:

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

The dependency graph of this recurrence is an $m \times n$ grid of vertices (i, j) connected by vertical edges $(i-1, j) \to (i, j)$, horizontal edges $(i, j-1) \to (i, j)$, and diagonal edges $(i-1, j-1) \to (i, j)$. 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.

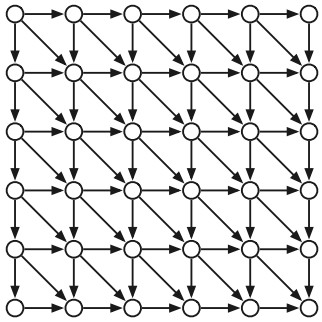


Figure 5.7. The dependency graph of the edit distance recurrence.

Another interesting example is the ***configuration graph*** of a game, puzzle, or mechanism like tic-tac-toe, checkers, the Rubik's Cube, the Towers 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 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.

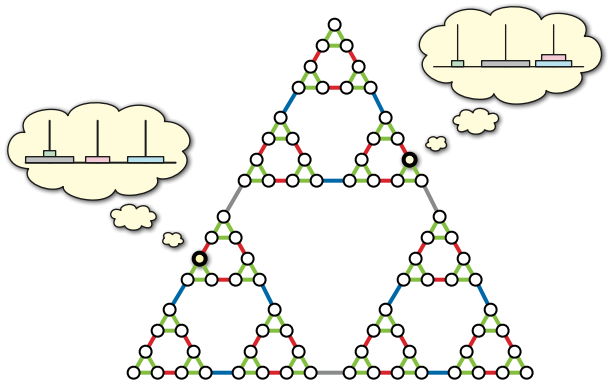


Figure 5.8. The configuration graph of the 4-disk Tower of Hanoi.

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 $a \in \Sigma$. Then basic questions about the language accepted by M can be phrased as questions about the graph G_M. For example, the language accepted by M is empty if and only if there is no path in G_M from the start state/vertex q_0 to an accepting state/vertex.

Finally, sometimes one graph can be used to implicitly represent other larger graphs. A good example of this implicit representation is the subset construction used to convert NFAs into DFAs. The subset construction can be generalized 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′***. Viewed in this light, the *incremental* subset construction used to convert NFAs to DFAs without unreachable states is just a breadth-first search of the implicitly-represented DFA.

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.

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).[7] 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 $O(V + E)$.

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 takes $O(1 + \deg(v))$ time; just scan v's neighbor list. Similarly, we can determine whether $u \to v$ 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 we locate the edge or when we fall of the end of a list.

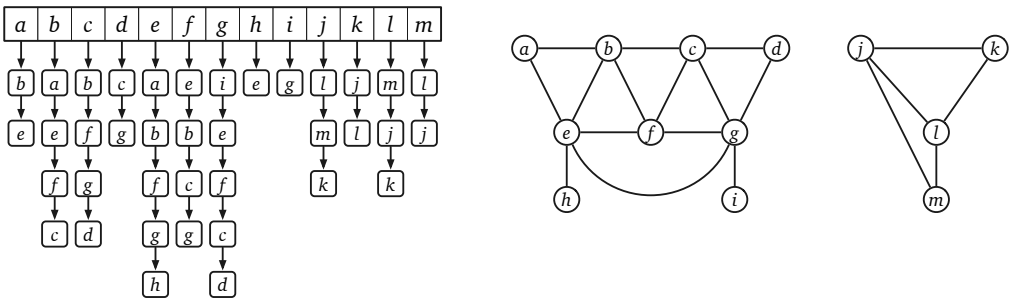


Figure 5.9. An adjacency list for our example graph.

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

[7] 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.

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.[8]

Adjacency Matrices

The other standard data structure for graphs is the ***adjacency matrix***.[9] 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.

- If the graph is undirected, then $A[u, v] := \left[uv \in E\right]$ for all vertices u and v.
- If the graph is directed, then $A[u, v] := \left[u{\to}v \in E\right]$ for all vertices u and v.

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

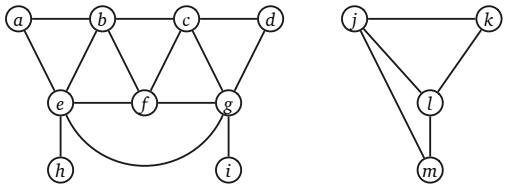


Figure 5.10. An adjacency matrix for our example graph.

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.

[8] 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. I discuss hashing in gory detail in a different chapter.

[9] See footnote 1.

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.[10]

	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. Analysis of basic operations on standard graph data structures.

At this point, 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 some more principled arguments for storing neighbors in linked lists. 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 pointless.

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 it 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.

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

- Similarly, we can apply any graph algorithm to a configuration graph *as though* it were given to us 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?) Thus, a standard adjacency list, with neighbors stored in linked lists, is the appropriate model data structure.

By default, all time bounds for graph algorithms assume that the graph is represented by 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 w is there a path from v to w? 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 me 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 highlighted in 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 1. *WhateverFirstSearch(s) marks every vertex reachable from s and only those vertices. Moreover, the set of pairs $(v, parent(v))$ with $parent(v) \neq \emptyset$ 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 put immediately (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 $parent(v) \neq \emptyset$ is actually an edge in the underlying graph G. We claim that for any marked vertex v, the path of parent edges $v \to parent(v) \to parent(parent(v)) \to \cdots$ 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 $parent(v) \to parent(parent(v)) \to \cdots$ leads to s; adding one more parent edge $s \to parent(s)$ 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. Thus, 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 $(\star\star)$ is executed at most $2E$ times. Finally, we can't take more things out of the bag than we put in, so line $(\star)$ is executed at most $2E + 1$ times. Thus, assuming the underlying graph G is stored in a standard adjacency list, WhateverFirstSearch runs in $\boldsymbol{O(V + ET)}$ ***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 $\boldsymbol{O(V + E)}$ ***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.

Queue: Breadth-First

If we implement the "bag" using a *queue*, we get a different graph-traversal algorithm called ***breadth-first search***. Stacks support insertions (push) and deletions (pop) 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 connected component; we'll discuss shortest paths in more detail in a later chapter. Again, 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.

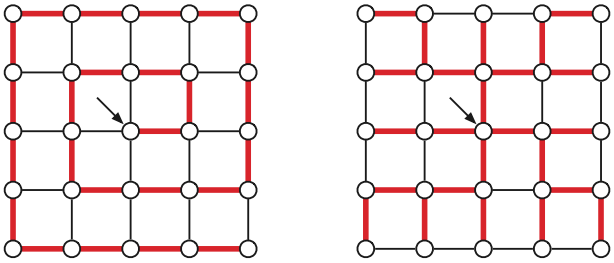


Figure 5.11. A depth-first spanning tree and a breadth-first spanning tree of the same graph, both starting at the center vertex.

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 behavior by the algorithm. I'll describe three well-known variants below, but there are many others. In all three examples, we assume that every edge uv in the input graph has a weight $w(uv)$.

For example, if 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 known as *Prim's algorithm*; we'll discuss this and other minimum-spanning-trees in more detail in a later chapter.

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 $dist(v) \leftarrow dist(p) + w(pv)$, and when we insert (v, w) into the priority queue, we use the priority $dist(v) + w(vw)$. 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 known as *Dijkstra's algorithm*; we'll discuss this and other shortest-path algorithms in more detail in a later chapter.

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. 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 $width(v) \leftarrow \min\{width(p), w(pv)\}$, and when we insert the edge (v, w) into the priority queue, we use the priority $\min\{width(v), w(vw)\}$.

Disconnected Graphs

WHATEVERFIRSTSEARCH(s) only visits the vertices reachable from 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 we doing something so complicated? Why can't we just scan the vertex array?

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

Well, sure, but then the order that we're visiting the vertices is determined by the data structure rather than the connectivity structure of the graph. 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 scan 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 labels 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 computes a spanning *forest* of the entire graph. In particular, best-first search with edge weights as priorities computes the minimum-weight spanning forest in $O(V + E \log E)$.

Surprisingly, at least one *extremely* popular algorithms textbook claims that this wrapper can only be used with depth-first search.[11] They're wrong.

Directed Graphs

Whatever-first search can be adapted trivially to directed graphs; the only change 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 $parent(v)\to p$ 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. Moreoverover, if we use the wrapper function WFSALL, the resulting parent pointers *do* define a spanning tree for each (weakly) connected component of any directed graph.

[11] 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."

5.7 Graph Reductions: Flood Fill

One of the earliest modern examples of whatever-first search, dating back at least to Edward Moore in the mid-1950s, is the flood fill problem. A *pixel map* is a two-dimensional array whose value represent colors; the individual entries in the array are called *pixels*, an abbreviation of *picture elements*.[12] A *connected region* in a pixel map is a connected subset of pixels all with the same color, where two pixels are connected if they are immediate horizontal or vertical neighbors. The *flood fill* operation, commonly represented with 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.

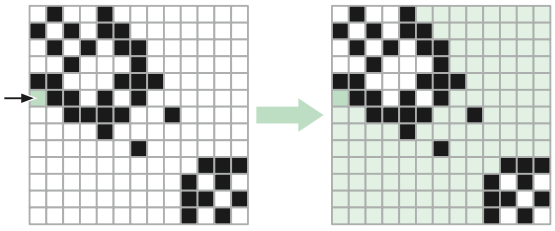


Figure 5.12. An example of flood fill

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.

In an actual implementation, we would not actually build a separate graph data structure for G; rather, because we can list the same-color neighbors of any pixel in $O(1)$ time each, we would use the pixel map itself as a representation of the graph *as though* it were a standard adjacency list. 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***. More careful analysis implies that the running time is proportional to the number of pixels in the region being filled, which could be considerably smaller than $O(n^2)$.

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. How whatever-first search works is utterly unimportant here; all that matters is its *specification*: Given a graph G and a starting vertex s, mark every vertex that is reachable from s. Like any other subroutine, we still have to describe how to construct

[12] 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 stones.

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.

Exercises

Sequences/Arrays

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 makes the graph disconnected.
   - 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.

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. 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 $G = T$.

4. 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) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

5. 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 A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.
   (a) Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. 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).

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

   (a) Prove that a connected graph G has an Euler tour if and only if every vertex has even degree.

   (b) Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such graph exists.

7. The d-dimensional hypercube is the graph defined as follows. There are $2d$ 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 visits every 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.]*

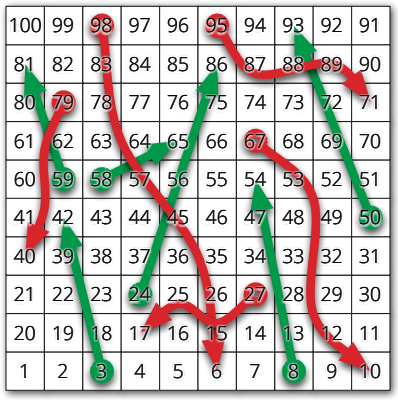
8. Recall that a directed graph G is *strongly connected* if, for any two vertices u and v, there is a path in G from u to v and a path in G from v to u.

   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.

9. ***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, 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.

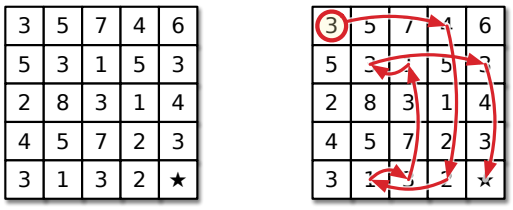
Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.



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

10. 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.

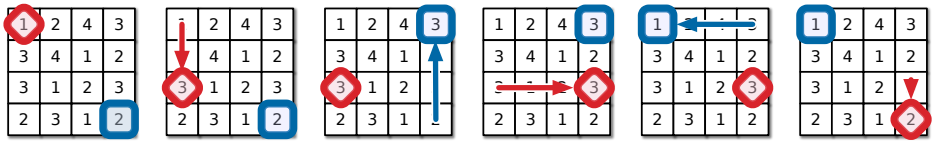


A 5 × 5 number maze that can be solved in eight moves.

11. 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,

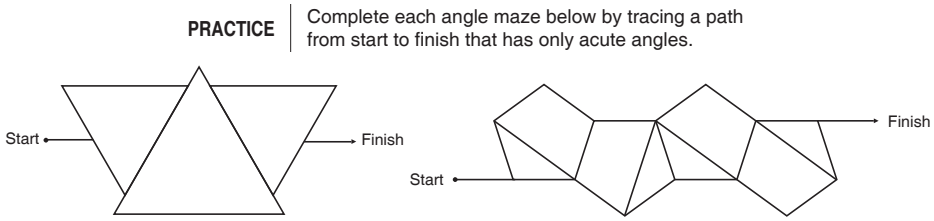
then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move either token off the grid, and at the end of a move the two tokens cannot lie on the same square.



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

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 above, your algorithm would return the number 5.

12. The following puzzles appear in my younger daughter's math workbook.[13]



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; the first maze above has 13 vertices and 15 edges. You are also given two vertices Start and Finish.

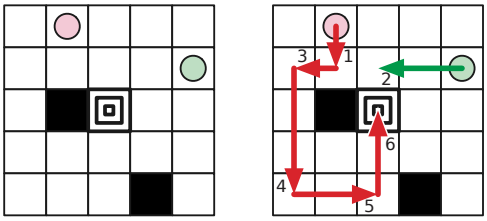
Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Formally, a walk through G is valid if, for any two consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = 180°$ or $0 < \angle uvw < 90°$. Assume you have a subroutine that can compute the angle between any two segments in $O(1)$ time. Do ***not*** assume that angles are integer multiples of 1°.

13. 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

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

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 below 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.



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

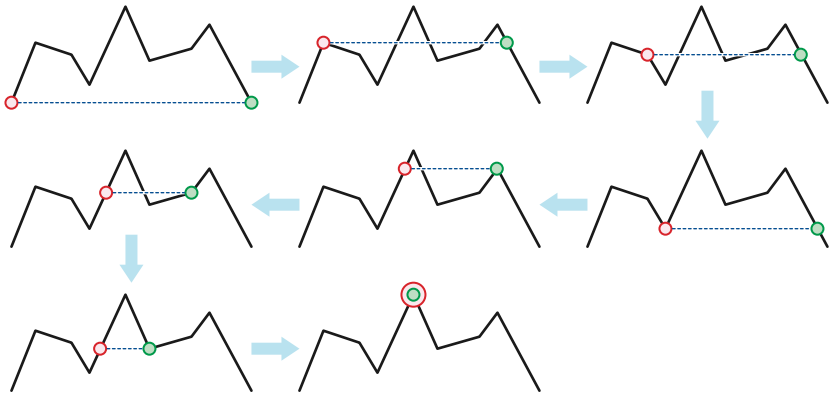
Describe and analyze an algorithm to determine whether an instance of this puzzle is solvable. Your input consist of the integer n, a list of obstacle locations, the target location, and the initial locations of the tokens. The output of your algorithm is a single boolean: TRUE if the given puzzle is solvable and FALSE otherwise. The running time of your algorithm should be a small polynomial in n.

14. Every cheesy romance 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 rainbows and kittens and chocolate unicorns and. . . .

    Suppose a romantic couple—in grand computer science tradition, named Alice and Bob—enters their favorite park at the southeast and southwest corners 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 part between the southeast and southwest entrances. To maintain the proper dramatic tension, Alice and Bob must traverse the path so that they lie on a direct east-west line.

    We can describe the zigzag path as two arrays $X[0..n]$ and $Y[0..n]$, which list the x- and y-coordinates of the corners of the path in order from the southwest endpoint to the southeast endpoint. Clearly $Y[0] = Y[n]$.

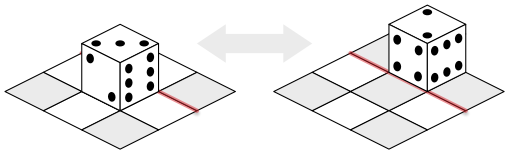
    (a) Suppose $Y[0] = Y[n] = 0$ and $Y[i] > 0$ for every other index i; that is, the endpoints of the path are strictly below every other point on the path. Prove that for any path P meeting these conditions, Alice and Bob can *always* meet. *[Hint: Describe a graph that models all possible locations of the couple along the path. What are the vertices of this graph? What are the edges? What can we say about the degrees of the vertices?]*



Alice and Bob meet.

(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.

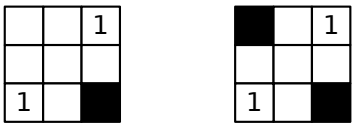
15. A *rolling die maze* is a puzzle involving a standard six-sided die (a cube with numbers on each side) and a grid of squares. You should imagine the grid lying on top of a table; the die always rests on and exactly covers one square. In a single step, you can *roll* the die 90 degrees around one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



Rolling a die.

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 of pips 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.

For example, here are two rolling die mazes. Black squares are blocked. The maze on the left can be solved by placing the die on the lower left square with 1 pip on the top face, and then rolling it north, then north, then east, then east. The maze on the right is not solvable.



Two rolling die mazes. Only the maze on the left is 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 only be solved if the initial position is chosen correctly.]

16. ***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.[14] 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.

    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.

    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. *[Hint: Build a graph. What are the vertices? What are the edges? What problem is this?]*

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

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)

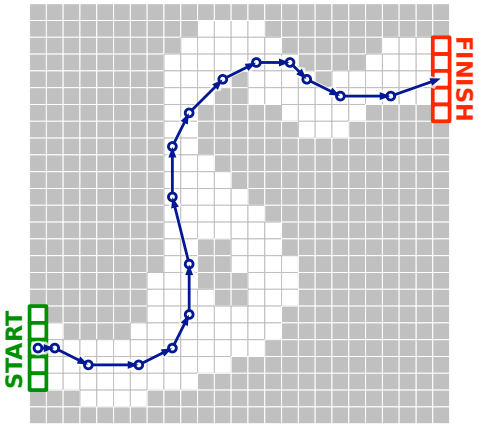


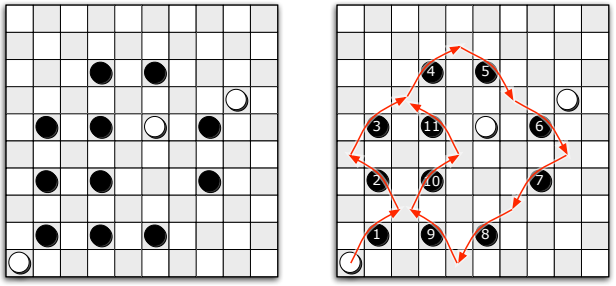
Figure 5.13. A 16-step Racetrack run, on a 25 × 25 track. This is *not* the shortest run on this track.

17. Draughts, also known in the US as "checkers", is a game played on an $m \times m$ grid of squares, alternately colored light and dark. 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.

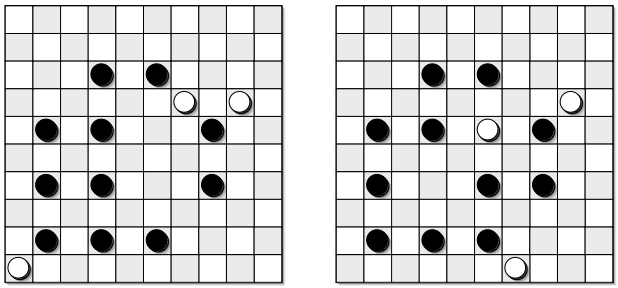
    Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[15] Pieces can be moved in any of the four diagonal directions. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square, or makes 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 6. Parity, parity, parity.]*

[15] Most other variants of draughts have "flying kings", which behave *very* differently than what's described here, and which make this problem *much* more difficult.



White wins in one turn.



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