Graphs are one of the unifying themes of computer science—an abstract representation that describes the organization of transportation systems, human interactions, and telecommunication networks. That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.

More precisely, a graph $G = (V, E)$ consists of a set of vertices $V$ together with a set $E$ of vertex pairs or edges. Graphs are important because they can be used to represent essentially any relationship. For example, graphs can model a network of roads, with cities as vertices and roads between cities as edges, as shown in Figure 5.1. Electronic circuits can also be modeled as graphs, with junctions as vertices and components as edges.

![Figure 5.1: Modeling road networks and electronic circuits as graphs](image-url)
The key to solving many algorithmic problems is to think of them in terms of graphs. Graph theory provides a language for talking about the properties of relationships, and it is amazing how often messy applied problems have a simple description and solution in terms of classical graph properties.

Designing truly novel graph algorithms is a very difficult task. The key to using graph algorithms effectively in applications lies in correctly modeling your problem so you can take advantage of existing algorithms. Becoming familiar with many different algorithmic graph problems is more important than understanding the details of particular graph algorithms, particularly since Part II of this book will point you to an implementation as soon as you know the name of your problem.

Here we present basic data structures and traversal operations for graphs, which will enable you to cobble together solutions for basic graph problems. Chapter 6 will present more advanced graph algorithms that find minimum spanning trees, shortest paths, and network flows, but we stress the primary importance of correctly modeling your problem. Time spent browsing through the catalog now will leave you better informed of your options when a real job arises.

5.1 Flavors of Graphs

A graph $G = (V, E)$ is defined on a set of vertices $V$, and contains a set of edges $E$ of ordered or unordered pairs of vertices from $V$. In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges. In analyzing the source code of a computer program, the vertices may represent lines of code, with an edge connecting lines $x$ and $y$ if $y$ is the next statement executed after $x$. In analyzing human interactions, the vertices typically represent people, with edges connecting pairs of related souls.

Several fundamental properties of graphs impact the choice of the data structures used to represent them and algorithms available to analyze them. The first step in any graph problem is determining the flavors of graphs you are dealing with:

- **Undirected vs. Directed** – A graph $G = (V, E)$ is undirected if edge $(x, y) \in E$ implies that $(y, x)$ is also in $E$. If not, we say that the graph is directed. Road networks between cities are typically undirected, since any large road has lanes going in both directions. Street networks within cities are almost always directed, because there are at least a few one-way streets lurking somewhere. Program-flow graphs are typically directed, because the execution flows from one line into the next and changes direction only at branches. Most graphs of graph-theoretic interest are undirected.

- **Weighted vs. Unweighted** – Each edge (or vertex) in a weighted graph $G$ is assigned a numerical value, or weight. The edges of a road network graph might be weighted with their length, drive-time, or speed limit, depending upon the
application. In *unweighted* graphs, there is no cost distinction between various edges and vertices.

The difference between weighted and unweighted graphs becomes particularly apparent in finding the shortest path between two vertices. For unweighted graphs, the shortest path must have the fewest number of edges, and can be found using a breadth-first search as discussed in this chapter. Shortest paths in weighted graphs requires more sophisticated algorithms, as discussed in Chapter 6.

- **Simple vs. Non-simple** – Certain types of edges complicate the task of working with graphs. A *self-loop* is an edge \((x, x)\) involving only one vertex. An edge \((x, y)\) is a *multiedge* if it occurs more than once in the graph.

Both of these structures require special care in implementing graph algorithms. Hence any graph that avoids them is called *simple*. 

![Diagram of graphs](image-url)
• **Sparse vs. Dense:** Graphs are *sparse* when only a small fraction of the possible vertex pairs \( \binom{n}{2} \) for a simple, undirected graph on \( n \) vertices) actually have edges defined between them. Graphs where a large fraction of the vertex pairs define edges are called *dense*. There is no official boundary between what is called sparse and what is called dense, but typically dense graphs have a quadratic number of edges, while sparse graphs are linear in size.

Sparse graphs are usually sparse for application-specific reasons. Road networks must be sparse graphs because of road junctions. The most ghastly intersection I’ve ever heard of was the endpoint of only seven different roads. Junctions of electrical components are similarly limited to the number of wires that can meet at a point, perhaps except for power and ground.

• **Cyclic vs. Acyclic** – An *acyclic* graph does not contain any cycles. *Trees* are connected, acyclic undirected graphs. Trees are the simplest interesting graphs, and are inherently recursive structures because cutting any edge leaves two smaller trees.

Directed acyclic graphs are called *DAGs*. They arise naturally in scheduling problems, where a directed edge \((x, y)\) indicates that activity \(x\) must occur before \(y\). An operation called *topological sorting* orders the vertices of a DAG to respect these precedence constraints. Topological sorting is typically the first step of any algorithm on a DAG, as will be discussed in Section 5.10.1 (page 179).

• **Embedded vs. Topological** – A graph is *embedded* if the vertices and edges are assigned geometric positions. Thus, any drawing of a graph is an *embedding*, which may or may not have algorithmic significance.

Occasionally, the structure of a graph is completely defined by the geometry of its embedding. For example, if we are given a collection of points in the plane, and seek the minimum cost tour visiting all of them (i.e., the traveling salesman problem), the underlying topology is the *complete graph* connecting each pair of vertices. The weights are typically defined by the Euclidean distance between each pair of points.

Grids of points are another example of topology from geometry. Many problems on an \( n \times m \) grid involve walking between neighboring points, so the edges are implicitly defined from the geometry.

• **Implicit vs. Explicit** – Certain graphs are not explicitly constructed and then traversed, but built as we use them. A good example is in backtrack search. The vertices of this implicit search graph are the states of the search vector, while edges link pairs of states that can be directly generated from each other. Because you do not have to store the entire graph, it is often easier to work with an implicit graph than explicitly construct it prior to analysis.
5.1 FLAVORS OF GRAPHS

- Labeled vs. Unlabeled – Each vertex is assigned a unique name or identifier in a labeled graph to distinguish it from all other vertices. In unlabeled graphs, no such distinctions have been made.

Graphs arising in applications are often naturally and meaningfully labeled, such as city names in a transportation network. A common problem is that of isomorphism testing—determining whether the topological structure of two graphs are identical if we ignore any labels. Such problems are typically solved using backtracking, by trying to assign each vertex in each graph a label such that the structures are identical.

5.1.1 The Friendship Graph

To demonstrate the importance of proper modeling, let us consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends. Such graphs are called social networks and are well defined on any set of people—be they the people in your neighborhood, at your school/business, or even spanning the entire world. An entire science analyzing social networks has sprung up in recent years, because many interesting aspects of people and their behavior are best understood as properties of this friendship graph.

Most of the graphs that one encounters in real life are sparse. The friendship graph is a good example. Even the most gregarious person on earth knows an insignificant fraction of the world’s population.

We use this opportunity to demonstrate the graph theory terminology described above. “Talking the talk” proves to be an important part of “walking the walk”:

- If I am your friend, does that mean you are my friend? – This question really asks whether the graph is directed. A graph is undirected if edge $(x, y)$ always implies $(y, x)$. Otherwise, the graph is said to be directed. The “heard-of” graph is directed, since I have heard of many famous people who have never heard of me! The “had-sex-with” graph is presumably undirected, since the critical operation always requires a partner. I’d like to think that the “friendship” graph is also an undirected graph.
• *How close a friend are you?* – In *weighted* graphs, each edge has an associated numerical attribute. We could model the strength of a friendship by associating each edge with an appropriate value, perhaps from -10 (enemies) to 10 (blood brothers). The edges of a road network graph might be weighted with their length, drive-time, or speed limit, depending upon the application. A graph is said to be *unweighted* if all edges are assumed to be of equal weight.

• *Am I my own friend?* – This question addresses whether the graph is *simple*, meaning it contains no loops and no multiple edges. An edge of the form \((x, x)\) is said to be a *loop*. Sometimes people are friends in several different ways. Perhaps \(x\) and \(y\) were college classmates and now work together at the same company. We can model such relationships using *multiedges*—multiple edges \((x, y)\) perhaps distinguished by different labels.

Simple graphs really are often simpler to work with in practice. Therefore, we might be better off declaring that no one is their own friend.

• *Who has the most friends?* – The *degree* of a vertex is the number of edges adjacent to it. The most popular person defines the vertex of highest degree in the friendship graph. Remote hermits are associated with degree-zero vertices.

In *dense* graphs, most vertices have high degrees, as opposed to *sparse* graphs with relatively few edges. In a *regular graph*, each vertex has exactly the same degree. A regular friendship graph is truly the ultimate in social-ism.

• *Do my friends live near me?* – Social networks are not divorced from geography. Many of your friends are your friends only because they happen to live near you (e.g., neighbors) or used to live near you (e.g., college roommates).

Thus, a full understanding of social networks requires an *embedded* graph, where each vertex is associated with the point on this world where they live. This geographic information may not be explicitly encoded, but the fact that the graph is inherently embedded in the plane shapes our interpretation of any analysis.

• *Oh, you also know her?* – Social networking services such as Myspace and LinkedIn are built on the premise of *explicitly* defining the links between members and their member-friends. Such graphs consist of directed edges from person/vertex \(x\) professing his friendship to person/vertex \(y\).

That said, the complete friendship graph of the world is represented *implicitly*. Each person knows who their friends are, but cannot find out about other people’s friendships except by asking them. The “six degrees of separation” theory argues that there is a short path linking every two people in the world (e.g., Skiena and the President) but offers us no help in actually finding this path. The shortest such path I know of contains three hops (Steven Skiena \(\rightarrow\) Bob McGrath \(\rightarrow\) John Marberger \(\rightarrow\) George W. Bush), but there could
be a shorter one (say, if he went to college with my dentist). The friendship
graph is stored implicitly, so I have no way of easily checking.

- *Are you truly an individual, or just one of the faceless crowd?* – This question
boils down to whether the friendship graph is labeled or unlabeled. Does each
vertex have a name/label which reflects its identity, and is this label important
for our analysis?

Much of the study of social networks is unconcerned with labels on graphs.
Often the index number given a vertex in the graph data structure serves
as its label, perhaps for convenience or the need for anonymity. You may
assert that you are a name, not a number—but try protesting to the guy
who implements the algorithm. Someone studying how an infectious disease
spreads through a graph may label each vertex with whether the person is
healthy or sick, it being irrelevant what their name is.

*Take-Home Lesson:* Graphs can be used to model a wide variety of structures
and relationships. Graph-theoretic terminology gives us a language to talk
about them.

## 5.2 Data Structures for Graphs

Selecting the right graph data structure can have an enormous impact on perfor-
manence. Your two basic choices are adjacency matrices and adjacency lists, illus-
trated in Figure 5.4. We assume the graph $G = (V, E)$ contains $n$ vertices and $m$
edges.

- **Adjacency Matrix:** We can represent $G$ using an $n \times n$ matrix $M$, where
element $M[i, j] = 1$ if $(i, j)$ is an edge of $G$, and 0 if it isn’t. This allows fast
answers to the question “is $(i, j)$ in $G$?”, and rapid updates for edge insertion
and deletion. It may use excessive space for graphs with many vertices and
relatively few edges, however.
5. GRAPH TRAVERSAL

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster to test if ((x, y)) is in graph?</td>
<td>adjacency matrices</td>
</tr>
<tr>
<td>Faster to find the degree of a vertex?</td>
<td>adjacency lists</td>
</tr>
<tr>
<td>Less memory on small graphs?</td>
<td>adjacency lists ((m + n)) vs. (n^2)</td>
</tr>
<tr>
<td>Less memory on big graphs?</td>
<td>adjacency matrices (O(1)) vs. (O(d))</td>
</tr>
<tr>
<td>Edge insertion or deletion?</td>
<td>adjacency lists (\Theta(m + n)) vs. (\Theta(n^2))</td>
</tr>
<tr>
<td>Faster to traverse the graph?</td>
<td>adjacency lists</td>
</tr>
<tr>
<td>Better for most problems?</td>
<td>adjacency lists</td>
</tr>
</tbody>
</table>

Figure 5.5: Relative advantages of adjacency lists and matrices.

Consider a graph that represents the street map of Manhattan in New York City. Every junction of two streets will be a vertex of the graph. Neighboring junctions are connected by edges. How big is this graph? Manhattan is basically a grid of 15 avenues each crossing roughly 200 streets. This gives us about 3,000 vertices and 6,000 edges, since each vertex neighbors four other vertices and each edge is shared between two vertices. This modest amount of data should easily and efficiently be stored, yet an adjacency matrix would have \(3,000 \times 3,000 = 9,000,000\) cells, almost all of them empty!

There is some potential to save space by packing multiple bits per word or simulating a triangular matrix on undirected graphs. But these methods lose the simplicity that makes adjacency matrices so appealing and, more critically, remain inherently quadratic on sparse graphs.

- **Adjacency Lists**: We can more efficiently represent sparse graphs by using linked lists to store the neighbors adjacent to each vertex. Adjacency lists require pointers but are not frightening once you have experience with linked structures.

Adjacency lists make it harder to verify whether a given edge \((i, j)\) is in \(G\), since we must search through the appropriate list to find the edge. However, it is surprisingly easy to design graph algorithms that avoid any need for such queries. Typically, we sweep through all the edges of the graph in one pass via a breadth-first or depth-first traversal, and update the implications of the current edge as we visit it. Table 5.5 summarizes the tradeoffs between adjacency lists and matrices.

**Take-Home Lesson**: Adjacency lists are the right data structure for most applications of graphs.

We will use adjacency lists as our primary data structure to represent graphs. We represent a graph using the following data type. For each graph, we keep a
count of the number of vertices, and assign each vertex a unique identification number from 1 to \texttt{nvertices}. We represent the edges using an array of linked lists:

\begin{verbatim}
#define MAXV 1000 /* maximum number of vertices */
typedef struct {
    int y; /* adjacency info */
    int weight; /* edge weight, if any */
    struct edgenode *next; /* next edge in list */
} edgenode;

typedef struct {
    edgenode *edges[MAXV+1]; /* adjacency info */
    int degree[MAXV+1]; /* outdegree of each vertex */
    int nvertices; /* number of vertices in graph */
    int nedges; /* number of edges in graph */
    bool directed; /* is the graph directed? */
} graph;
\end{verbatim}

We represent directed edge \((x, y)\) by an \texttt{edgenode y} in \(x\)’s adjacency list. The degree field of the \texttt{graph} counts the number of meaningful entries for the given vertex. An undirected edge \((x, y)\) appears twice in any adjacency-based graph structure, once as \(y\) in \(x\)’s list, and once as \(x\) in \(y\)’s list. The boolean flag \texttt{directed} identifies whether the given graph is to be interpreted as directed or undirected.

To demonstrate the use of this data structure, we show how to read a graph from a file. A typical graph format consists of an initial line featuring the number of vertices and edges in the graph, followed by a listing of the edges at one vertex pair per line.

\begin{verbatim}
initialize_graph(graph *g, bool directed) {
    int i; /* counter */

    g -> nvertices = 0;
    g -> nedges = 0;
    g -> directed = directed;

    for (i=1; i<=MAXV; i++) g->degree[i] = 0;
    for (i=1; i<=MAXV; i++) g->edges[i] = NULL;
}
\end{verbatim}

Actually reading the graph requires inserting each edge into this structure:
read_graph(graph *g, bool directed)
{
    int i;          /* counter */
    int m;          /* number of edges */
    int x, y;       /* vertices in edge (x,y) */

    initialize_graph(g, directed);

    scanf("%d %d", &(g->nvertices), &m);

    for (i=1; i<=m; i++) {
        scanf("%d %d", &x, &y);
        insert_edge(g, x, y, directed);
    }
}

The critical routine is insert_edge. The new edgenode is inserted at the beginning of the appropriate adjacency list, since order doesn’t matter. We parameterize our insertion with the directed Boolean flag, to identify whether we need to insert two copies of each edge or only one. Note the use of recursion to solve this problem:

insert_edge(graph *g, int x, int y, bool directed)
{
    edgenode *p;          /* temporary pointer */

    p = malloc(sizeof(edgenode)); /* allocate edgenode storage */

    p->weight = NULL;
    p->y = y;
    p->next = g->edges[x];

    g->edges[x] = p;            /* insert at head of list */
    g->degree[x] ++;

    if (directed == FALSE)
        insert_edge(g, y, x, TRUE);
    else
        g->nedges ++;
}

Printing the associated graph is just a matter of two nested loops, one through vertices, the other through adjacent edges:
5.3 War Story: I was a Victim of Moore's Law

I am the author of a popular library of graph algorithms called Combinatorica (see www.combinatorica.com), which runs under the computer algebra system Mathematica. Efficiency is a great challenge in Mathematica, due to its applicative model of computation (it does not support constant-time write operations to arrays) and the overhead of interpretation (as opposed to compilation). Mathematica code is typically 1,000 to 5,000 times slower than C code.

Such slowdowns can be a tremendous performance hit. Even worse, Mathematica was a memory hog, needing a then-outrageous 4MB of main memory to run effectively when I completed Combinatorica in 1990. Any computation on large structures was doomed to thrash in virtual memory. In such an environment, my graph package could only hope to work effectively on very small graphs.

One design decision I made as a result was to use adjacency matrices as the basic Combinatorica graph data structure instead of lists. This may sound peculiar. If pressed for memory, wouldn’t it pay to use adjacency lists and conserve every last byte? Yes, but the answer is not so simple for very small graphs. An adjacency list representation of a weighted $n$-vertex, $m$-edge graph should use about $n + 2m$ words to represent; the $2m$ comes from storing the endpoint and weight components of...
each edge. Thus, the space advantages of adjacency lists kick in when $n + 2m$ is substantially smaller than $n^2$. The adjacency matrix is still manageable in size for $n \leq 100$ and, of course, half the size of adjacency lists on dense graphs.

My more immediate concern was dealing with the overhead of using a slow interpreted language. Check out the benchmarks reported in Table 5.1. Two particularly complex but polynomial-time problems on 9 and 16 vertex graphs took several minutes to complete on my desktop machine in 1990! The quadratic-sized data structure certainly could not have had much impact on these running times, since $9 \times 9$ equals only 81. From experience, I knew the *Mathematica* programming language handled better to regular structures like adjacency matrices better than irregular-sized adjacency lists.

Still, *Combinatorica* proved to be a very good thing despite these performance problems. Thousands of people have used my package to do all kinds of interesting things with graphs. *Combinatorica* was never intended to be a high-performance algorithms library. Most users quickly realized that computations on large graphs were out of the question, but were eager to take advantage of *Combinatorica* as a mathematical research tool and prototyping environment. Everyone was happy.

But over the years, my users started asking why it took so long to do a modest-sized graph computation. The mystery wasn’t that my program was slow, because it had always been slow. The question was why did it take so many years for people to figure this out?
5.3 War Story: I Was a Victim of Moore's Law

The reason is that computers keep doubling in speed every two years or so. People’s expectation of how long something should take moves in concert with these technology improvements. Partially because of *Combinatorica*’s dependence on a quadratic-size graph data structure, it didn’t scale as well as it should on sparse graphs.

As the years rolled on, user demands become more insistent. *Combinatorica* needed to be updated. My collaborator, Sriram Pemmaraju, rose to the challenge. We (mostly he) completely rewrote *Combinatorica* to take advantage of faster graph data structures ten years after the initial version.

The new *Combinatorica* uses a list of edges data structure for graphs, largely motivated by increased efficiency. Edge lists are linear in the size of the graph (edges plus vertices), just like adjacency lists. This makes a huge difference on most graph related functions—for large enough graphs. The improvement is most dramatic in “fast” graph algorithms—those that run in linear or near linear-time, such as graph traversal, topological sort, and finding connected/biconnected components. The implications of this change is felt throughout the package in running time improvements and memory savings. *Combinatorica* can now work with graphs that are about 50-100 times larger than graphs that the old package could deal with.

Figure 5.7(l) plots the running time of the *MinimumSpanningTree* functions for both *Combinatorica* versions. The test graphs were sparse (grid graphs), designed to highlight the difference between the two data structures. Yes, the new version is much faster, but note that the difference only becomes important for graphs larger than the old *Combinatorica* was designed for. However, the relative difference in run time keeps growing with increasing n. Figure 5.7(r) plots the ratio of the running times as a function of graph size. The difference between linear size and quadratic size is asymptotic, so the consequences become ever more important as n gets larger.

What is the weird bump in running times that occurs around \( n \approx 250 \)? This likely reflects a transition between levels of the memory hierarchy. Such bumps are not uncommon in today’s complex computer systems. Cache performance in data structure design should be an important but not overriding consideration.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>command/machine</td>
<td>Sun-3</td>
<td>Sun-4</td>
<td>Sun-5</td>
<td>Ultra 5</td>
<td>SunBlade</td>
</tr>
<tr>
<td>PlanarQ[GridGraph[4,4]]</td>
<td>234.10</td>
<td>69.65</td>
<td>27.50</td>
<td>3.60</td>
<td>0.40</td>
</tr>
<tr>
<td>Length[Partitions[30]]</td>
<td>289.85</td>
<td>73.20</td>
<td>24.40</td>
<td>3.44</td>
<td>1.58</td>
</tr>
<tr>
<td>VertexConnectivity[GridGraph[3,3]]</td>
<td>239.67</td>
<td>47.76</td>
<td>14.70</td>
<td>2.00</td>
<td>0.91</td>
</tr>
<tr>
<td>RandomPartition[1000]</td>
<td>831.68</td>
<td>267.5</td>
<td>22.05</td>
<td>3.12</td>
<td>0.87</td>
</tr>
</tbody>
</table>

Table 5.1: Old *Combinatorica* benchmarks on low-end Sun workstations, from 1990 to today, (running time in seconds)
The asymptotic gains due to adjacency lists more than trumped any impact of the cache.

Two main lessons can be taken away from our experience developing Combinatorica:

- **To make a program run faster, just wait** – Sophisticated hardware eventually slithers down to everybody. We observe a speedup of more than 200-fold for the original version of Combinatorica as a consequence of 15 years of faster hardware. In this context, the further speedups we obtained from upgrading the package become particularly dramatic.

- **Asymptotics eventually do matter** – It was my mistake not to anticipate future developments in technology. While no one has a crystal ball, it is fairly safe to say that future computers will have more memory and run faster than today’s. This gives an edge to asymptotically more efficient algorithms/data structures, even if their performance is close on today’s instances. If the implementation complexity is not substantially greater, play it safe and go with the better algorithm.

### 5.4 War Story: Getting the Graph

“It takes five minutes just to read the data. We will never have time to make it do something interesting.”

The young graduate student was bright and eager, but green to the power of data structures. She would soon come to appreciate their power.
As described in a previous war story (see Section 3.6 (page 85)), we were experimenting with algorithms to extract triangular strips for the fast rendering of triangulated surfaces. The task of finding a small number of strips that cover each triangle in a mesh could be modeled as a graph problem. The graph has a vertex for every triangle of the mesh, with an edge between every pair of vertices representing adjacent triangles. This dual graph representation (see Figure 5.8) captures all the information needed to partition the triangulation into triangle strips.

The first step in crafting a program that constructs a good set of strips was to build the dual graph of the triangulation. This I sent the student off to do. A few days later, she came back and announced that it took over five CPU minutes just to construct this dual graph of a few thousand triangles.

“Nonsense!” I proclaimed. “You must be doing something very wasteful in building the graph. What format is the data in?”

“Well, it starts out with a list of the 3D coordinates of the vertices used in the model and then follows with a list of triangles. Each triangle is described by a list of three indices into the vertex coordinates. Here is a small example:”

```
VERTICES 4
0.000000 240.000000 0.000000
204.000000 240.000000 0.000000
204.000000 0.000000 0.000000
0.000000 0.000000 0.000000

TRIANGLES 2
0 1 3
1 2 3
```

“I see. So the first triangle uses all but the third point, since all the indices start from zero. The two triangles must share an edge formed by points 1 and 3.”

“Yeah, that’s right,” she confirmed.
“OK. Now tell me how you built your dual graph from this file.”

“Well, I can ignore the vertex information once I know how many vertices there are. The geometric position of the points doesn’t affect the structure of the graph. My dual graph is going to have as many vertices as the number of triangles. I set up an adjacency list data structure with that many vertices. As I read in each triangle, I compare it to each of the others to check whether it has two end points in common. If it does, I add an edge from the new triangle to this one.”

I started to sputter. “But that’s your problem right there! You are comparing each triangle against every other triangle, so that constructing the dual graph will be quadratic in the number of triangles. Reading the input graph should take linear time!”

“I’m not comparing every triangle against every other triangle. On average, it only tests against half or a third of the triangles.”

“Swell. But that still leaves us with an $O(n^2)$ algorithm. That is much too slow.”

She stood her ground. “Well, don’t just complain. Help me fix it!”

Fair enough. I started to think. We needed some quick method to screen away most of the triangles that would not be adjacent to the new triangle $(i, j, k)$. What we really needed was a separate list of all the triangles that go through each of the points $i$, $j$, and $k$. By Euler’s formula for planar graphs, the average point is incident on less than six triangles. This would compare each new triangle against fewer than twenty others, instead of most of them.

“We are going to need a data structure consisting of an array with one element for every vertex in the original data set. This element is going to be a list of all the triangles that pass through that vertex. When we read in a new triangle, we will look up the three relevant lists in the array and compare each of these against the new triangle. Actually, only two of the three lists need be tested, since any adjacent triangles will share two points in common. We will add an adjacency to our graph for every triangle-pair sharing two vertices. Finally, we will add our new triangle to each of the three affected lists, so they will be updated for the next triangle read.”

She thought about this for a while and smiled. “Got it, Chief. I’ll let you know what happens.”

The next day she reported that the graph could be built in seconds, even for much larger models. From here, she went on to build a successful program for extracting triangle strips, as reported in Section 3.6 (page 85).

The take-home lesson is that even elementary problems like initializing data structures can prove to be bottlenecks in algorithm development. Most programs working with large amounts of data have to run in linear or almost linear time. Such tight performance demands leave no room to be sloppy. Once you focus on the need for linear-time performance, an appropriate algorithm or heuristic can usually be found to do the job.
5.5 Traversing a Graph

Perhaps the most fundamental graph problem is to visit every edge and vertex in a graph in a systematic way. Indeed, all the basic bookkeeping operations on graphs (such as printing or copying graphs, and converting between alternate representations) are applications of graph traversal.

Mazes are naturally represented by graphs, where each graph vertex denotes a junction of the maze, and each graph edge denotes a hallway in the maze. Thus, any graph traversal algorithm must be powerful enough to get us out of an arbitrary maze. For efficiency, we must make sure we don’t get trapped in the maze and visit the same place repeatedly. For correctness, we must do the traversal in a systematic way to guarantee that we get out of the maze. Our search must take us through every edge and vertex in the graph.

The key idea behind graph traversal is to mark each vertex when we first visit it and keep track of what we have not yet completely explored. Although bread crumbs or unraveled threads have been used to mark visited places in fairy-tale mazes, we will rely on Boolean flags or enumerated types.

Each vertex will exist in one of three states:

- **undiscovered** – the vertex is in its initial, virgin state.
- **discovered** – the vertex has been found, but we have not yet checked out all its incident edges.
- **processed** – the vertex after we have visited all its incident edges.

Obviously, a vertex cannot be **processed** until after we discover it, so the state of each vertex progresses over the course of the traversal from **undiscovered** to **discovered** to **processed**.

We must also maintain a structure containing the vertices that we have discovered but not yet completely processed. Initially, only the single start vertex is considered to be discovered. To completely explore a vertex \( v \), we must evaluate each edge leaving \( v \). If an edge goes to an undiscovered vertex \( x \), we mark \( x \) **discovered** and add it to the list of work to do. We ignore an edge that goes to a **processed** vertex, because further contemplation will tell us nothing new about the graph. We can also ignore any edge going to a **discovered** but not **processed** vertex, because the destination already resides on the list of vertices to process.

Each undirected edge will be considered exactly twice, once when each of its endpoints is explored. Directed edges will be considered only once, when exploring the source vertex. Every edge and vertex in the connected component must eventually be visited. Why? Suppose that there exists a vertex \( u \) that remains unvisited, whose neighbor \( v \) was visited. This neighbor \( v \) will eventually be explored, after which we will certainly visit \( u \). Thus, we must find everything that is there to be found.

We describe the mechanics of these traversal algorithms and the significance of the traversal order below.
Figure 5.9: An undirected graph and its breadth-first search tree

### 5.6 Breadth-First Search

The basic breadth-first search algorithm is given below. At some point during the course of a traversal, every node in the graph changes state from *undiscovered* to *discovered*. In a breadth-first search of an undirected graph, we assign a direction to each edge, from the discoverer \( u \) to the discovered \( v \). We thus denote \( u \) to be the parent of \( v \). Since each node has exactly one parent, except for the root, this defines a tree on the vertices of the graph. This tree, illustrated in Figure 5.9, defines a shortest path from the root to every other node in the tree. This property makes breadth-first search very useful in shortest path problems.

\[
\text{BFS}(G, s)
\]

for each vertex \( u \in V[G] - \{s\} \) do

\begin{align*}
\text{state}[u] &= \text{“undiscovered”} \\
\text{p}[u] &= \text{nil}, \text{i.e. no parent is in the BFS tree}
\end{align*}

\begin{align*}
\text{state}[s] &= \text{“discovered”} \\
\text{p}[s] &= \text{nil} \\
Q &= \{s\}
\end{align*}

while \( Q \neq \emptyset \) do

\begin{align*}
\text{u} &= \text{dequeue}[Q] \\
\text{process vertex } u \text{ as desired}
\end{align*}

for each \( v \in \text{Adj}[u] \) do

\begin{align*}
\text{process edge } (u, v) \text{ as desired} \\
\text{if } \text{state}[v] = \text{“undiscovered”} \text{ then} \\
\text{state}[v] &= \text{“discovered”} \\
\text{p}[v] &= u \\
\text{enqueue}[Q, v]
\end{align*}

\begin{align*}
\text{state}[u] &= \text{“processed”}
\end{align*}

The graph edges that do not appear in the breadth-first search tree also have special properties. For undirected graphs, nontree edges can point only to vertices on the same level as the parent vertex, or to vertices on the level directly below
the parent. These properties follow easily from the fact that each path in the tree must be the shortest path in the graph. For a directed graph, a back-pointing edge \((u, v)\) can exist whenever \(v\) lies closer to the root than \(u\) does.

**Implementation**

Our breadth-first search implementation `bfs` uses two Boolean arrays to maintain our knowledge about each vertex in the graph. A vertex is **discovered** the first time we visit it. A vertex is considered **processed** after we have traversed all outgoing edges from it. Thus, each vertex passes from undiscovered to discovered to processed over the course of the search. This information could have been maintained using one enumerated type variable, but we used two Boolean variables instead.

```c
bool processed[MAXV+1]; /* which vertices have been processed */
bool discovered[MAXV+1]; /* which vertices have been found */
int parent[MAXV+1]; /* discovery relation */
```

Each vertex is initialized as undiscovered:

```c
initialize_search(graph *g) {
    int i; /* counter */
    for (i=1; i<=g->nvertices; i++) {
        processed[i] = discovered[i] = FALSE;
        parent[i] = -1;
    }
}
```

Once a vertex is discovered, it is placed on a queue. Since we process these vertices in first-in, first-out order, the oldest vertices are expanded first, which are exactly those closest to the root:

```c
bfs(graph *g, int start) {
    queue q; /* queue of vertices to visit */
    int v; /* current vertex */
    int y; /* successor vertex */
    edgenode *p; /* temporary pointer */
    init_queue(&q);
    enqueue(&q, start);
    discovered[start] = TRUE;
```
while (empty_queue(&q) == FALSE) {
    v = dequeue(&q);
    process_vertex_early(v);
    processed[v] = TRUE;
    p = g->edges[v];
    while (p != NULL) {
        y = p->y;
        if ((processed[y] == FALSE) || g->directed)
            process_edge(v,y);
        if (discovered[y] == FALSE) {
            enqueue(&q,y);
            discovered[y] = TRUE;
            parent[y] = v;
        }
        p = p->next;
    }
    process_vertex_late(v);
}

5.6.1 Exploiting Traversal

The exact behavior of bfs depends upon the functions process_vertex_early(), process_vertex_late(), and process_edge(). Through these functions, we can customize what the traversal does as it makes its official visit to each edge and each vertex. Initially, we will do all of vertex processing on entry, so process_vertex_late() returns without action:

process_vertex_late(int v)
{
}

By setting the active functions to

process_vertex_early(int v)
{
    printf("processed vertex %d\n",v);
}

process_edge(int x, int y)
{
    printf("processed edge (%d,%d)\n",x,y);
}
we print each vertex and edge exactly once. If we instead set `process_edge` to

```c
process_edge(int x, int y)
{
    nedges = nedges + 1;
}
```

we get an accurate count of the number of edges. Different algorithms perform
different actions on vertices or edges as they are encountered. These functions give
us the freedom to easily customize our response.

### 5.6.2 Finding Paths

The `parent` array set within `bfs()` is very useful for finding interesting paths
through a graph. The vertex that discovered vertex `i` is defined as `parent[i]`. Every vertex is discovered during the course of traversal, so except for the root
every node has a parent. The parent relation defines a tree of discovery with the
initial search node as the root of the tree.

Because vertices are discovered in order of increasing distance from the root,
this tree has a very important property. The unique tree path from the root to
each node \( x \in V \) uses the smallest number of edges (or equivalently, intermediate
nodes) possible on any root-to-\( x \) path in the graph.

We can reconstruct this path by following the chain of ancestors from \( x \) to the
root. Note that we have to work backward. We cannot find the path from the root to
\( x \), since that does not follow the direction of the parent pointers. Instead, we
must find the path from \( x \) to the root. Since this is the reverse of how we normally
want the path, we can either (1) store it and then explicitly reverse it using a stack,
or (2) let recursion reverse it for us, as follows:

```c
find_path(int start, int end, int parents[])
{
    if ((start == end) || (end == -1))
        printf("\n%d",start);
    else {
        find_path(start,parents[end],parents);
        printf(" %d",end);
    }
}
```

On our breadth-first search graph example (Figure 5.9) our algorithm generated
the following parent relation:

<table>
<thead>
<tr>
<th>vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
For the shortest path from 1 to 4, upper-right corner, this parent relation yields the path \{1, 5, 4\}.

There are two points to remember when using breadth-first search to find a shortest path from \(x\) to \(y\): First, the shortest path tree is only useful if BFS was performed with \(x\) as the root of the search. Second, BFS gives the shortest path only if the graph is unweighted. We will present algorithms for finding shortest paths in weighted graphs in Section 6.3.1 (page 206).

\section{Applications of Breadth-First Search}

Most elementary graph algorithms make one or two traversals of the graph while we update our knowledge of the graph. Properly implemented using adjacency lists, any such algorithm is destined to be linear, since BFS runs in \(O(n + m)\) time on both directed and undirected graphs. This is optimal, since it is as fast as one can hope to \textit{read} any \(n\)-vertex, \(m\)-edge graph.

The trick is seeing when traversal approaches are destined to work. We present several examples below.

\subsection{Connected Components}

The “six degrees of separation” theory argues that there is always a short path linking every two people in the world. We say that a graph is \textit{connected} if there is a path between any two vertices. If the theory is true, it means the friendship graph must be connected.

A \textit{connected component} of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices. The components are separate “pieces” of the graph such that there is no connection between the pieces. If we envision tribes in remote parts of the world that have yet not been encountered, each such tribe would form a separate connected component in the friendship graph. A remote hermit, or extremely unpleasant fellow, would represent a connected component of one vertex.

An amazing number of seemingly complicated problems reduce to finding or counting connected components. For example, testing whether a puzzle such as Rubik’s cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected.

Connected components can be found using breadth-first search, since the vertex order does not matter. We start from the first vertex. Anything we discover during this search must be part of the same connected component. We then repeat the search from any undiscovered vertex (if one exists) to define the next component, and so on until all vertices have been found:
connected_components(graph *g)
{
    int c; /* component number */
    int i; /* counter */

    initialize_search(g);

    c = 0;
    for (i=1; i<=g->nvertices; i++)
        if (discovered[i] == FALSE) {
            c = c+1;
            printf("Component %d:",c);
            bfs(g,i);
            printf("\n");
        }
}

process_vertex_early(int v)
{
    printf(" %d",v);
}

process_edge(int x, int y)
{
}

Observe how we increment a counter \( c \) denoting the current component number with each call to \( \text{bfs} \). We could have explicitly bound each vertex to its component number (instead of printing the vertices in each component) by changing the action of \( \text{process\_vertex\_early} \).

There are two distinct notions of connectivity for directed graphs, leading to algorithms for finding both weakly connected and strongly connected components. Both of these can be found in \( O(n + m) \) time, as discussed in Section 15.1 (page 477).

### 5.7.2 Two-Coloring Graphs

The \textit{vertex-coloring} problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color. We can easily avoid all conflicts by assigning each vertex a unique color. However, the goal is to use as few colors as possible. Vertex coloring problems often arise in scheduling applications, such as register allocation in compilers. See Section 16.7 (page 544) for a full treatment of vertex-coloring algorithms and applications.
A graph is bipartite if it can be colored without conflicts while using only two colors. Bipartite graphs are important because they arise naturally in many applications. Consider the "had-sex-with" graph in a heterosexual world. Men have sex only with women, and vice versa. Thus, gender defines a legal two-coloring, in this simple model.

But how can we find an appropriate two-coloring of a graph, thus separating the men from the women? Suppose we assume that the starting vertex is male. All vertices adjacent to this man must be female, assuming the graph is indeed bipartite.

We can augment breadth-first search so that whenever we discover a new vertex, we color it the opposite of its parent. We check whether any nondiscovery edge links two vertices of the same color. Such a conflict means that the graph cannot be two-colored. Otherwise, we will have constructed a proper two-coloring whenever we terminate without conflict.

twocolor(graph *g)
{
  int i; /* counter */

  for (i=1; i<=(g->nvertices); i++)
    color[i] = UNCOLORED;

  bipartite = TRUE;

  initialize_search(&g);

  for (i=1; i<=(g->nvertices); i++)
    if (discovered[i] == FALSE) {
      color[i] = WHITE;
      bfs(g,i);
    }
}

process_edge(int x, int y)
{
  if (color[x] == color[y]) {
    bipartite = FALSE;
    printf("Warning: not bipartite due to (%d,%d)\n",x,y);
  }

  color[y] = complement(color[x]);
}
complement(int color) {
    if (color == WHITE) return(BLACK);
    if (color == BLACK) return(WHITE);
    return(UNCOLORED);
}

We can assign the first vertex in any connected component to be whatever color/sex we wish. BFS can separate the men from the women, but we can’t tell them apart just by using the graph structure.

Take-Home Lesson: Breadth-first and depth-first searches provide mechanisms to visit each edge and vertex of the graph. They prove the basis of most simple, efficient graph algorithms.

5.8 Depth-First Search

There are two primary graph traversal algorithms: breadth-first search (BFS) and depth-first search (DFS). For certain problems, it makes absolutely no difference which you use, but in others the distinction is crucial.

The difference between BFS and DFS results is in the order in which they explore vertices. This order depends completely upon the container data structure used to store the discovered but not processed vertices.

- **Queue** – By storing the vertices in a first-in, first-out (FIFO) queue, we explore the oldest unexplored vertices first. Thus our explorations radiate out slowly from the starting vertex, defining a breadth-first search.

- **Stack** – By storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by lurching along a path, visiting a new neighbor if one is available, and backing up only when we are surrounded by previously discovered vertices. Thus, our explorations quickly wander away from our starting point, defining a depth-first search.

Our implementation of dfs maintains a notion of traversal time for each vertex. Our time clock ticks each time we enter or exit any vertex. We keep track of the entry and exit times for each vertex.

Depth-first search has a neat recursive implementation, which eliminates the need to explicitly use a stack:

\[\text{DFS}(G, u)\]
\[
\text{state}[u] = \text{“discovered”}
\]
\[
\text{process vertex } u \text{ if desired}
\]
\[
\text{entry}[u] = \text{time}
\]
The time intervals have interesting and useful properties with respect to depth-first search:

- **Who is an ancestor?** – Suppose that \( x \) is an ancestor of \( y \) in the DFS tree. This implies that we must enter \( x \) before \( y \), since there is no way we can be born before our own father or grandfather! We also must exit \( y \) before we exit \( x \), because the mechanics of DFS ensure we cannot exit \( x \) until after we have backed up from the search of all its descendants. Thus the time interval of \( y \) must be properly nested within ancestor \( x \).

- **How many descendants?** – The difference between the exit and entry times for \( v \) tells us how many descendents \( v \) has in the DFS tree. The clock gets incremented on each vertex entry and vertex exit, so half the time difference denotes the number of descendents of \( v \).

We will use these entry and exit times in several applications of depth-first search, particularly topological sorting and biconnected/strongly-connected components. We need to be able to take separate actions on each entry and exit, thus motivating distinct `process_vertex_early` and `process_vertex_late` routines called from `dfs`.

The other important property of a depth-first search is that it partitions the edges of an undirected graph into exactly two classes: *tree edges* and *back edges*. The tree edges discover new vertices, and are those encoded in the `parent` relation. Back edges are those whose other endpoint is an ancestor of the vertex being expanded, so they point back into the tree.

An amazing property of depth-first search is that all edges fall into these two classes. Why can’t an edge go to a brother or cousin node instead of an ancestor? All nodes reachable from a given vertex \( v \) are expanded before we finish with the traversal from \( v \), so such topologies are impossible for undirected graphs. This edge classification proves fundamental to the correctness of DFS-based algorithms.
Implementation

A depth-first search can be thought of as a breadth-first search with a stack instead of a queue. The beauty of implementing dfs recursively is that recursion eliminates the need to keep an explicit stack:

```c
dfs(graph *g, int v) {
    edgenode *p; /* temporary pointer */
    int y; /* successor vertex */

    if (finished) return; /* allow for search termination */

    discovered[v] = TRUE;
    time = time + 1;
    entry_time[v] = time;

    process_vertex_early(v);

    p = g->edges[v];
    while (p != NULL) {
        y = p->y;
        if (discovered[y] == FALSE) {
            parent[y] = v;
            process_edge(v, y);
            dfs(g, y);
        } else if ((!processed[y]) || (g->directed))
            process_edge(v, y);
    }
}
```
if (finished) return;
    p = p->next;
}

process_vertex_late(v);

time = time + 1;
exit_time[v] = time;
processed[v] = TRUE;

Depth-first search use essentially the same idea as backtracking, which we study in Section 7.1 (page 231). Both involve exhaustively searching all possibilities by advancing if it is possible, and backing up as soon as there is no unexplored possibility for further advancement. Both are most easily understood as recursive algorithms.

**Take-Home Lesson:** DFS organizes vertices by entry/exit times, and edges into tree and back edges. This organization is what gives DFS its real power.

### 5.9 Applications of Depth-First Search

As algorithm design paradigms go, a depth-first search isn’t particularly intimidating. It is surprisingly subtle, however meaning that its correctness requires getting details right.

The correctness of a DFS-based algorithm depends upon specifics of exactly when we process the edges and vertices. We can process vertex \( v \) either before we have traversed any of the outgoing edges from \( v \) (\textit{process vertex early}()) or after we have finished processing all of them (\textit{process vertex late}()). Sometimes we will take special actions at both times, say \textit{process vertex early}() to initialize a vertex-specific data structure, which will be modified on edge-processing operations and then analyzed afterwards using \textit{process vertex late}().

In undirected graphs, each edge \((x, y)\) sits in the adjacency lists of vertex \( x \) and \( y \). Thus there are two potential times to process each edge \((x, y)\), namely when exploring \( x \) and when exploring \( y \). The labeling of edges as tree edges or back edges occurs during the first time the edge is explored. This first time we see an edge is usually a logical time to do edge-specific processing. Sometimes, we may want to take different action the second time we see an edge.

But when we encounter edge \((x, y)\) from \( x \), how can we tell if we have previously traversed the edge from \( y \)? The issue is easy if vertex \( y \) is undiscovered: \((x, y)\)
becomes a tree edge so this must be the first time. The issue is also easy if $y$ has not been completely processed; since we explored the edge when we explored $y$ this must be the second time. But what if $y$ is an ancestor of $x$, and thus in a discovered state? Careful reflection will convince you that this must be our first traversal unless $y$ is the immediate ancestor of $x$—i.e., $(y, x)$ is a tree edge. This can be established by testing if $y == \text{parent}[x]$.

I find that the subtlety of depth-first search-based algorithms kicks me in the head whenever I try to implement one. I encourage you to analyze these implementations carefully to see where the problematic cases arise and why.

### 5.9.1 Finding Cycles

Back edges are the key to finding a cycle in an undirected graph. If there is no back edge, all edges are tree edges, and no cycle exists in a tree. But any back edge going from $x$ to an ancestor $y$ creates a cycle with the tree path from $y$ to $x$. Such a cycle is easy to find using $\text{dfs}$:

```c
process_edge(int x, int y)
{
    if (parent[x] != y) { /* found back edge! */
        printf("Cycle from %d to %d:",y,x);
        find_path(y,x,parent);
        printf("\n\n");
        finished = TRUE;
    }
}
```

The correctness of this cycle detection algorithm depends upon processing each undirected edge exactly once. Otherwise, a spurious two-vertex cycle $(x, y, x)$ could be composed from the two traversals of any single undirected edge. We use the $\text{finished}$ flag to terminate after finding the first cycle.

### 5.9.2 Articulation Vertices

Suppose you are a vandal seeking to disrupt the telephone network. Which station in Figure 5.11 should you choose to blow up to cause the maximum amount of
damage? Observe that there is a single point of failure—a single vertex whose
deletion disconnects a connected component of the graph. Such a vertex is called
an articulation vertex or cut-node. Any graph that contains an articulation vertex
is inherently fragile, because deleting that single vertex causes a loss of connectivity
between other nodes.

We presented a breadth-first search-based connected components algorithm in
Section 5.7.1 (page 166). In general, the connectivity of a graph is the smallest
number of vertices whose deletion will disconnect the graph. The connectivity is
one if the graph has an articulation vertex. More robust graphs without such a
vertex are said to be biconnected. Connectivity will be further discussed in the
catalog in Section 15.8 (page 505).

Testing for articulation vertices by brute force is easy. Temporarily delete each
vertex \( v \), and then do a BFS or DFS traversal of the remaining graph to establish
whether it is still connected. The total time for \( n \) such traversals is \( O(n(m + n)) \).
There is a clever linear-time algorithm, however, that tests all the vertices of a
connected graph using a single depth-first search.

What might the depth-first search tree tell us about articulation vertices? This
tree connects all the vertices of the graph. If the DFS tree represented the entirety
of the graph, all internal (nonleaf) nodes would be articulation vertices, since deleting
any one of them would separate a leaf from the root. Blowing up a leaf (such as
vertices 2 and 6 in Figure 5.12) cannot disconnect the tree, since it connects no
one but itself to the main trunk.

Figure 5.12: DFS tree of a graph containing two articulation vertices (namely 1 and 2). Back
dge (5, 2) keeps vertices 3 and 4 from being cut-nodes. Vertices 5 and 6 escape as leaves of
the DFS tree.
The root of the search tree is a special case. If it has only one child, it functions as a leaf. But if the root has two or more children, its deletion disconnects them, making the root an articulation vertex.

General graphs are more complex than trees. But a depth-first search of a general graph partitions the edges into tree edges and back edges. Think of these back edges as security cables linking a vertex back to one of its ancestors. The security cable from \( x \) back to \( y \) ensures that none of the vertices on the tree path between \( x \) and \( y \) can be articulation vertices. Delete any of these vertices, and the security cable will still hold all of them to the rest of the tree.

Finding articulation vertices requires maintaining the extent to which back edges (i.e., security cables) link chunks of the DFS tree back to ancestor nodes. Let \( \text{reachable}_\text{ancestor}[v] \) denote the earliest reachable ancestor of vertex \( v \), meaning the oldest ancestor of \( v \) that we can reach by a combination of tree edges and back edges. Initially, \( \text{reachable}_\text{ancestor}[v] = v \):

```c
int reachable_ancestor[MAXV+1]; /* earliest reachable ancestor of v */
int tree_out_degree[MAXV+1]; /* DFS tree outdegree of v */
```

We update \( \text{reachable}_\text{ancestor}[v] \) whenever we encounter a back edge that takes us to an earlier ancestor than we have previously seen. The relative age/rank of our ancestors can be determined from their entry time's:

```c
process_vertex_early(int v)
{
    reachable_ancestor[v] = v;
}
```

```c
process_edge(int x, int y)
{
    int class; /* edge class */

    class = edge_classification(x,y);

    if (class == TREE)
        tree_out_degree[x] = tree_out_degree[x] + 1;

    if ((class == BACK) && (parent[x] != y)) {
        if (entry_time[y] < entry_time[ reachable_ancestor[x] ] )
            reachable_ancestor[x] = y;
    }
}
```

The key issue is determining how the reachability relation impacts whether vertex \( v \) is an articulation vertex. There are three cases, as shown in Figure 5.13:
5. GRAPH TRAVERSAL

Figure 5.13: The three cases of articulation vertices: root, bridge, and parent cut-nodes

- **Root cut-nodes** – If the root of the DFS tree has two or more children, it must be an articulation vertex. No edges from the subtree of the second child can possibly connect to the subtree of the first child.

- **Bridge cut-nodes** – If the earliest reachable vertex from $v$ is $v$, then deleting the single edge ($parent[v], v$) disconnects the graph. Clearly $parent[v]$ must be an articulation vertex, since it cuts $v$ from the graph. Vertex $v$ is also an articulation vertex unless it is a leaf of the DFS tree. For any leaf, nothing falls off when you cut it.

- **Parent cut-nodes** – If the earliest reachable vertex from $v$ is the parent of $v$, then deleting the parent must sever $v$ from the tree unless the parent is the root.

The routine below systematically evaluates each of the three conditions as we back up from the vertex after traversing all outgoing edges. We use $entry\_time[v]$ to represent the age of vertex $v$. The reachability time $time\_v$ calculated below denotes the oldest vertex that can be reached using back edges. Getting back to an ancestor above $v$ rules out the possibility of $v$ being a cut-node:
process_vertex_late(int v)
{
    bool root; /* is the vertex the root of the DFS tree? */
    int time_v; /* earliest reachable time for v */
    int time_parent; /* earliest reachable time for parent[v] */

    if (parent[v] < 1) { /* test if v is the root */
        if (tree_out_degree[v] > 1)
            printf("root articulation vertex: %d \n",v);
            return;
    }

    root = (parent[parent[v]] < 1); /* is parent[v] the root? */
    if ((reachable_ancestor[v] == parent[v]) && (!root))
        printf("parent articulation vertex: %d \n",parent[v]);

    if (reachable_ancestor[v] == v) {
        printf("bridge articulation vertex: %d \n",parent[v]);

        if (tree_out_degree[v] > 0) /* test if v is not a leaf */
            printf("bridge articulation vertex: %d \n",v);
    }

    time_v = entry_time[reachable_ancestor[v]];
    time_parent = entry_time[ reachable_ancestor[parent[v]] ];

    if (time_v < time_parent)
        reachable_ancestor[parent[v]] = reachable_ancestor[v];
}

The last lines of this routine govern when we back up a node’s highest reachable
ancestor to its parent, namely whenever it is higher than the parent’s earliest
ancestor to date.

We can alternately talk about reliability in terms of edge failures instead of
vertex failures. Perhaps our vandal would find it easier to cut a cable instead of
blowing up a switching station. A single edge whose deletion disconnects the graph
is called a bridge; any graph without such an edge is said to be edge-biconnected.

Identifying whether a given edge (x, y) is a bridge is easily done in linear time
by deleting the edge and testing whether the resulting graph is connected. In fact
all bridges can be identified in the same $O(n+m)$ time. Edge (x, y) is a bridge if (1)
it is a tree edge, and (2) no back edge connects from y or below to x or above. This
can be computed with a minor modification of the reachable_ancestor function.
5.10 Depth-First Search on Directed Graphs

Depth-first search on an undirected graph proves useful because it organizes the edges of the graph in a very precise way, as shown in Figure 5.10.

When traversing undirected graphs, every edge is either in the depth-first search tree or a back edge to an ancestor in the tree. Let us review why. Suppose we encountered a “forward edge” \((x, y)\) directed toward a descendant vertex. In this case, we would have discovered \((x, y)\) while exploring \(y\), making it a back edge. Suppose we encounter a “cross edge” \((x, y)\), linking two unrelated vertices. Again, we would have discovered this edge when we explored \(y\), making it a tree edge.

For directed graphs, depth-first search labelings can take on a wider range of possibilities. Indeed, all four of the edge cases in Figure 5.14 can occur in traversing directed graphs. Still, this classification proves useful in organizing algorithms on directed graphs. We typically take a different action on edges from each different case.

The correct labeling of each edge can be readily determined from the state, discovery time, and parent of each vertex, as encoded in the following function:

```c
int edge_classification(int x, int y)
{
    if (parent[y] == x) return(TREE);
    if (discovered[y] && !processed[y]) return(BACK);
    if (processed[y] && (entry_time[y]>entry_time[x])) return(FORWARD);
    if (processed[y] && (entry_time[y]<entry_time[x])) return(CROSS);

    printf("Warning: unclassified edge (%d,%d)\n",x,y);
}
```
As with BFS, this implementation of the depth-first search algorithm includes places to optionally process each vertex and edge—say to copy them, print them, or count them. Both algorithms will traverse all edges in the same connected component as the starting point. Since we need to start with a vertex in each component to traverse a disconnected graph, we must start from any vertex remaining undiscovered after a component search. With the proper initialization, this completes the traversal algorithm:

\[
\text{DFS-graph}(G) \\
\quad \text{for each vertex } u \in V[G] \text{ do} \\
\quad \quad \text{state}[u] = \text{"undiscovered"} \\
\quad \quad \text{for each vertex } u \in V[G] \text{ do} \\
\quad \quad \quad \text{if state}[u] = \text{"undiscovered"} \text{ then} \\
\quad \quad \quad \quad \text{initialize new component, if desired} \\
\quad \quad \quad \text{DFS}(G, u)
\]

I encourage the reader to convince themselves of the correctness of these four conditions. What I said earlier about the subtlety of depth-first search goes double for directed graphs.

### 5.10.1 Topological Sorting

Topological sorting is the most important operation on directed acyclic graphs (DAGs). It orders the vertices on a line such that all directed edges go from left to right. Such an ordering cannot exist if the graph contains a directed cycle, because there is no way you can keep going right on a line and still return back to where you started from!

Each DAG has at least one topological sort. The importance of topological sorting is that it gives us an ordering to process each vertex before any of its successors. Suppose the edges represented precedence constraints, such that edge
(x, y) means job x must be done before job y. Then, any topological sort defines a legal schedule. Indeed, there can be many such orderings for a given DAG.

But the applications go deeper. Suppose we seek the shortest (or longest) path from x to y in a DAG. No vertex appearing after y in the topological order can contribute to any such path, because there will be no way to get back to y. We can appropriately process all the vertices from left to right in topological order, considering the impact of their outgoing edges, and know that we will have looked at everything we need before we need it. Topological sorting proves very useful in essentially any algorithmic problem on directed graphs, as discussed in the catalog in Section 15.2 (page 481).

Topological sorting can be performed efficiently using depth-first searching. A directed graph is a DAG if and only if no back edges are encountered. Labeling the vertices in the reverse order that they are marked processed finds a topological sort of a DAG. Why? Consider what happens to each directed edge \( \{x, y\} \) as we encounter it exploring vertex x:

- If y is currently **undiscovered**, then we start a DFS of y before we can continue with x. Thus y is marked **completed** before x is, and x appears before y in the topological order, as it must.

- If y is **discovered** but not **completed**, then \( \{x, y\} \) is a back edge, which is forbidden in a DAG.

- If y is **processed**, then it will have been so labeled before x. Therefore, x appears before y in the topological order, as it must.

Study the following implementation:

```c
process_vertex_late(int v)
{
    push(&sorted,v);
}

process_edge(int x, int y)
{
    int class; /* edge class */
    class = edge_classification(x,y);
    if (class == BACK)
        printf("Warning: directed cycle found, not a DAG\n");
}
```
topsort(graph *g)
{
    int i; /* counter */

    init_stack(&sorted);

    for (i=1; i<=g->nvertices; i++)
    if (discovered[i] == FALSE)
        dfs(g,i);

    print_stack(&sorted); /* report topological order */
}

We push each vertex on a stack as soon as we have evaluated all outgoing edges. The top vertex on the stack always has no incoming edges from any vertex on the stack. Repeatedly popping them off yields a topological ordering.

5.10.2 Strongly Connected Components

We are often concerned with strongly connected components— that is, partitioning a graph into chunks such that directed paths exist between all pairs of vertices within a given chunk. A directed graph is strongly connected if there is a directed path between any two vertices. Road networks should be strongly connected, or else there will be places you can drive to but not drive home from without violating one-way signs.

It is straightforward to use graph traversal to test whether a graph \( G = (V, E) \) is strongly connected in linear time. First, do a traversal from some arbitrary vertex \( v \). Every vertex in the graph had better be reachable from \( v \) (and hence discovered on the BFS or DFS starting from \( v \)), otherwise \( G \) cannot possibly be strongly connected. Now construct a graph \( G' = (V, E') \) with the same vertex and edge set as \( G \) but with all edges reversed— i.e., directed edge \((x, y) \in E \) iff \((y, x) \in E' \). Thus, any path from \( v \) to \( z \) in \( G' \) corresponds to a path from \( z \) to \( v \) in \( G \). By doing a DFS from \( v \) in \( G' \), we find all vertices with paths to \( v \) in \( G \). The graph is strongly connected iff all vertices in \( G \) can (1) reach \( v \) and (2) are reachable from \( v \).

Graphs that are not strongly connected can be partitioned into strongly connected components, as shown in Figure 5.16 (left). The set of such components and the weakly-connecting edges that link them together can be determined using DFS. The algorithm is based on the observation that it is easy to find a directed cycle using a depth-first search, since any back edge plus the down path in the DFS tree gives such a cycle. All vertices in this cycle must be in the same strongly connected component. Thus, we can shrink (contract) the vertices on this cycle down to a single vertex representing the component, and then repeat. This process terminates when no directed cycle remains, and each vertex represents a different strongly connected component.
Our approach to implementing this idea is reminiscent of finding biconnected components in Section 5.9.2 (page 173). We update our notion of the oldest reachable vertex in response to (1) nontree edges and (2) backing up from a vertex. Because we are working on a directed graph, we also must contend with forward edges (from a vertex to a descendant) and cross edges (from a vertex back to a nonancestor but previously discovered vertex). Our algorithm will peel one strong component off the tree at a time, and assign each of its vertices the number of the component it is in:

```c
strong_components(graph *g) {
    int i; /* counter */

    for (i=1; i<=(g->nvertices); i++) {
        low[i] = i;
        scc[i] = -1;
    }
    components_found = 0;
    init_stack(&active);
    initialize_search(&g);

    for (i=1; i<=(g->nvertices); i++)
        if (discovered[i] == FALSE) {
            dfs(g,i);
        }
}
```

Figure 5.16: The strongly-connected components of a graph, with the associated DFS tree
Define \( \text{low}[v] \) to be the oldest vertex known to be in the same strongly connected component as \( v \). This vertex is not necessarily an ancestor, but may also be a distant cousin of \( v \) because of cross edges. Cross edges that point vertices from previous strongly connected components of the graph cannot help us, because there can be no way back from them to \( v \), but otherwise cross edges are fair game. Forward edges have no impact on reachability over the depth-first tree edges, and hence can be disregarded:

```c
int low[MAXV+1]; /* oldest vertex surely in component of \( v \) */
int scc[MAXV+1]; /* strong component number for each vertex */

process_edge(int x, int y)
{
    int class;  /* edge class */

    class = edge_classification(x,y);

    if (class == BACK) {
        if (entry_time[y] < entry_time[low[x]])
            low[x] = y;
    }

    if (class == CROSS) {
        if (scc[y] == -1) /* component not yet assigned */
            if (entry_time[y] < entry_time[low[x]])
                low[x] = y;
    }
}
```

A new strongly connected component is found whenever the lowest reachable vertex from \( v \) is \( v \). If so, we can clear the stack of this component. Otherwise, we give our parent the benefit of the oldest ancestor we can reach and backtrack:

```c
process_vertex_early(int v)
{
    push(&active,v);
}
```
5. GRAPH TRAVERSAL

process_vertex_late(int v)
{
    if (low[v] == v) { /* edge (parent[v],v) cuts off scc */
        pop_component(v);
    }

    if (entry_time[low[v]] < entry_time[low[parent[v]])
        low[parent[v]] = low[v];
}

pop_component(int v)
{
    int t; /* vertex placeholder */

    components_found = components_found + 1;

    scc[v] = components_found;
    while ((t = pop(&active)) != v) {
        scc[t] = components_found;
    }
}

Chapter Notes

Our treatment of graph traversal represents an expanded version of material from
Chapter 9 of [SR03]. The Combinatorica graph library discussed in the war story
is best described in the old [Ski90], and new editions [PS03] of the associated book.
Accessible introductions to the science of social networks include Barabasi [Bar03]
and Watts [Wat04].

5.11 Exercises

Simulating Graph Algorithms

5-1. [3] For the following graphs $G_1$ (left) and $G_2$ (right):
(a) Report the order of the vertices encountered on a breadth-first search starting from vertex A. Break all ties by picking the vertices in alphabetical order (i.e., A before Z).

(b) Report the order of the vertices encountered on a depth-first search starting from vertex A. Break all ties by picking the vertices in alphabetical order (i.e., A before Z).

5-2. [3] Do a topological sort of the following graph $G$:

Traversal

5-3. [3] Prove by induction that there is a unique path between any pair of vertices in a tree.

5-4. [3] Prove that in a breadth-first search on a undirected graph $G$, every edge is either a tree edge or a cross edge, where $x$ is neither an ancestor nor descendant of $y$, in cross edge $(x, y)$.

5-5. [3] Give a linear algorithm to compute the chromatic number of graphs where each vertex has degree at most 2. Must such graphs be bipartite?

5-6. [5] In breadth-first and depth-first search, an undiscovered node is marked discovered when it is first encountered, and marked processed when it has been completely
5. GRAPHS TRAVERSAL

(a) Describe a graph on \( n \) vertices and a particular starting vertex \( v \) such that \( \Theta(n) \) nodes are simultaneously in the discovered state during a breadth-first search starting from \( v \).

(b) Describe a graph on \( n \) vertices and a particular starting vertex \( v \) such that \( \Theta(n) \) nodes are simultaneously in the discovered state during a depth-first search starting from \( v \).

(c) Describe a graph on \( n \) vertices and a particular starting vertex \( v \) such that at some point \( \Theta(n) \) nodes remain undiscovered, while \( \Theta(n) \) nodes have been processed during a depth-first search starting from \( v \). (Note, there may also be discovered nodes.)

5-7. \([4]\) Given pre-order and in-order traversals of a binary tree, is it possible to reconstruct the tree? If so, sketch an algorithm to do it. If not, give a counterexample. Repeat the problem if you are given the pre-order and post-order traversals.

5-8. \([3]\) Present correct and efficient algorithms to convert an undirected graph \( G \) between the following graph data structures. You must give the time complexity of each algorithm, assuming \( n \) vertices and \( m \) edges.

(a) Convert from an adjacency matrix to adjacency lists.

(b) Convert from an adjacency list to an incidence matrix. An incidence matrix \( M \) has a row for each vertex and a column for each edge, such that \( M[i, j] = 1 \) if vertex \( i \) is part of edge \( j \), otherwise \( M[i, j] = 0 \).

(c) Convert from an incidence matrix to adjacency lists.

5-9. \([3]\) Suppose an arithmetic expression is given as a tree. Each leaf is an integer and each internal node is one of the standard arithmetical operations (+, −, *, /). For example, the expression \( 2 + 3 \times 4 + (3 \times 4)/5 \) is represented by the tree in Figure 5.17(a). Give an \( O(n) \) algorithm for evaluating such an expression, where there are \( n \) nodes in the tree.

5-10. \([5]\) Suppose an arithmetic expression is given as a DAG (directed acyclic graph) with common subexpressions removed. Each leaf is an integer and each internal

Figure 5.17: Expression \( 2 + 3 \times 4 + (3 \times 4)/5 \) as a tree and a DAG

searched. At any given moment, several nodes might be simultaneously in the discovered state.

(a) Describe a graph on \( n \) vertices and a particular starting vertex \( v \) such that \( \Theta(n) \) nodes are simultaneously in the discovered state during a breadth-first search starting from \( v \).

(b) Describe a graph on \( n \) vertices and a particular starting vertex \( v \) such that \( \Theta(n) \) nodes are simultaneously in the discovered state during a depth-first search starting from \( v \).

(c) Describe a graph on \( n \) vertices and a particular starting vertex \( v \) such that at some point \( \Theta(n) \) nodes remain undiscovered, while \( \Theta(n) \) nodes have been processed during a depth-first search starting from \( v \). (Note, there may also be discovered nodes.)

5-7. \([4]\) Given pre-order and in-order traversals of a binary tree, is it possible to reconstruct the tree? If so, sketch an algorithm to do it. If not, give a counterexample. Repeat the problem if you are given the pre-order and post-order traversals.

5-8. \([3]\) Present correct and efficient algorithms to convert an undirected graph \( G \) between the following graph data structures. You must give the time complexity of each algorithm, assuming \( n \) vertices and \( m \) edges.

(a) Convert from an adjacency matrix to adjacency lists.

(b) Convert from an adjacency list to an incidence matrix. An incidence matrix \( M \) has a row for each vertex and a column for each edge, such that \( M[i, j] = 1 \) if vertex \( i \) is part of edge \( j \), otherwise \( M[i, j] = 0 \).

(c) Convert from an incidence matrix to adjacency lists.

5-9. \([3]\) Suppose an arithmetic expression is given as a tree. Each leaf is an integer and each internal node is one of the standard arithmetical operations (+, −, *, /). For example, the expression \( 2 + 3 \times 4 + (3 \times 4)/5 \) is represented by the tree in Figure 5.17(a). Give an \( O(n) \) algorithm for evaluating such an expression, where there are \( n \) nodes in the tree.

5-10. \([5]\) Suppose an arithmetic expression is given as a DAG (directed acyclic graph) with common subexpressions removed. Each leaf is an integer and each internal
node is one of the standard arithmetical operations (+, −, *, /). For example, the expression $2 + 3 \times 4 + (3 \times 4)/5$ is represented by the DAG in Figure 5.17(b). Give an $O(n + m)$ algorithm for evaluating such a DAG, where there are $n$ nodes and $m$ edges in the DAG. Hint: modify an algorithm for the tree case to achieve the desired efficiency.


Algorithm Design

5-12. [5] The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, w) \in E^2$ iff there exists $v \in V$ such that $(u, v) \in E$ and $(v, w) \in E$; i.e., there is a path of exactly two edges from $u$ to $w$.

Give efficient algorithms for both adjacency lists and matrices.

5-13. [5] A vertex cover of a graph $G = (V, E)$ is a subset of vertices $V'$ such that each edge in $E$ is incident on at least one vertex of $V'$.

(a) Give an efficient algorithm to find a minimum-size vertex cover if $G$ is a tree.

(b) Let $G = (V, E)$ be a tree such that the weight of each vertex is equal to the degree of that vertex. Give an efficient algorithm to find a minimum-weight vertex cover of $G$.

(c) Let $G = (V, E)$ be a tree with arbitrary weights associated with the vertices. Give an efficient algorithm to find a minimum-weight vertex cover of $G$.

5-14. [3] A vertex cover of a graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that every edge in $E$ contains at least one vertex from $V'$. Delete all the leaves from any depth-first search tree of $G$. Must the remaining vertices form a vertex cover of $G$? Give a proof or a counterexample.

5-15. [5] A vertex cover of a graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that every edge in $E$ contains at least one vertex from $V'$. An independent set of graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that no edge in $E$ contains both vertices from $V'$.

An independent vertex cover is a subset of vertices that is both an independent set and a vertex cover of $G$. Give an efficient algorithm for testing whether $G$ contains an independent vertex cover. What classical graph problem does this reduce to?

5-16. [5] An independent set of an undirected graph $G = (V, E)$ is a set of vertices $U$ such that no edge in $E$ is incident on two vertices of $U$.

(a) Give an efficient algorithm to find a maximum-size independent set if $G$ is a tree.

(b) Let $G = (V, E)$ be a tree with weights associated with the vertices such that the weight of each vertex is equal to the degree of that vertex. Give an efficient algorithm to find a maximum independent set of $G$.

(c) Let $G = (V, E)$ be a tree with arbitrary weights associated with the vertices. Give an efficient algorithm to find a maximum independent set of $G$.
5-17. [5] Consider the problem of determining whether a given undirected graph \( G = (V, E) \) contains a triangle or cycle of length 3.

(a) Give an \( O(|V|^3) \) to find a triangle if one exists.

(b) Improve your algorithm to run in time \( O(|V| \cdot |E|) \). You may assume \( |V| \leq |E| \).

Observe that these bounds gives you time to convert between the adjacency matrix and adjacency list representations of \( G \).

5-18. [5] Consider a set of movies \( M_1, M_2, \ldots, M_k \). There is a set of customers, each one of which indicates the two movies they would like to see this weekend. Movies are shown on Saturday evening and Sunday evening. Multiple movies may be screened at the same time.

You must decide which movies should be televised on Saturday and which on Sunday, so that every customer gets to see the two movies they desire. Is there a schedule where each movie is shown at most once? Design an efficient algorithm to find such a schedule if one exists.

5-19. [5] The diameter of a tree \( T = (V, E) \) is given by

\[
\max_{u,v \in V} \delta(u,v)
\]

(where \( \delta(u,v) \) is the number of edges on the path from \( u \) to \( v \)). Describe an efficient algorithm to compute the diameter of a tree, and show the correctness and analyze the running time of your algorithm.

5-20. [5] Given an undirected graph \( G \) with \( n \) vertices and \( m \) edges, and an integer \( k \), give an \( O(m + n) \) algorithm that finds the maximum induced subgraph \( H \) of \( G \) such that each vertex in \( H \) has degree \( \geq k \), or prove that no such graph exists. An induced subgraph \( F = (U, R) \) of a graph \( G = (V, E) \) is a subset of \( U \) of the vertices \( V \) of \( G \), and all edges \( R \) of \( G \) such that both vertices of each edge are in \( U \).

5-21. [6] Let \( v \) and \( w \) be two vertices in a directed graph \( G = (V, E) \). Design a linear-time algorithm to find the number of different shortest paths (not necessarily vertex disjoint) between \( v \) and \( w \). Note: the edges in \( G \) are unweighted.

5-22. [6] Design a linear-time algorithm to eliminate each vertex \( v \) of degree 2 from a graph by replacing edges \((u, v)\) and \((v, w)\) by an edge \((u, w)\). We also seek to eliminate multiple copies of edges by replacing them with a single edge. Note that removing multiple copies of an edge may create a new vertex of degree 2, which has to be removed, and that removing a vertex of degree 2 may create multiple edges, which also must be removed.

Directed Graphs

5-23. [5] Your job is to arrange \( n \) ill-behaved children in a straight line, facing front. You are given a list of \( m \) statements of the form “\( i \) hates \( j \)”. If \( i \) hates \( j \), then you do not want put \( i \) somewhere behind \( j \), because then \( i \) is capable of throwing something at \( j \).

(a) Give an algorithm that orders the line, (or says that it is not possible) in \( O(m + n) \) time.
(b) Suppose instead you want to arrange the children in rows such that if $i$ hates $j$, then $i$ must be in a lower numbered row than $j$. Give an efficient algorithm to find the minimum number of rows needed, if it is possible.

5-24. [3] Adding a single directed edge to a directed graph can reduce the number of weakly connected components, but by at most how many components? What about the number of strongly connected components?

5-25. [5] An arborescence of a directed graph $G$ is a rooted tree such that there is a directed path from the root to every other vertex in the graph. Give an efficient and correct algorithm to test whether $G$ contains an arborescence, and its time complexity.

5-26. [5] A mother vertex in a directed graph $G = (V, E)$ is a vertex $v$ such that all other vertices $G$ can be reached by a directed path from $v$.

(a) Give an $O(n + m)$ algorithm to test whether a given vertex $v$ is a mother of $G$, where $n = |V|$ and $m = |E|$.

(b) Give an $O(n + m)$ algorithm to test whether graph $G$ contains a mother vertex.

5-27. [9] A tournament is a directed graph formed by taking the complete undirected graph and assigning arbitrary directions on the edges—i.e., a graph $G = (V, E)$ such that for all $u, v \in V$, exactly one of $(u, v)$ or $(v, u)$ is in $E$. Show that every tournament has a Hamiltonian path—that is, a path that visits every vertex exactly once. Give an algorithm to find this path.

Articulation Vertices

5-28. [5] An articulation vertex of a graph $G$ is a vertex whose deletion disconnects $G$. Let $G$ be a graph with $n$ vertices and $m$ edges. Give a simple $O(n + m)$ algorithm for finding a vertex of $G$ that is not an articulation vertex—i.e., whose deletion does not disconnect $G$.

5-29. [5] Following up on the previous problem, give an $O(n + m)$ algorithm that finds a deletion order for the $n$ vertices such that no deletion disconnects the graph. (Hint: think DFS/BFS.)

5-30. [3] Suppose $G$ is a connected undirected graph. An edge $e$ whose removal disconnects the graph is called a bridge. Must every bridge $e$ be an edge in a depth-first search tree of $G$? Give a proof or a counterexample.

Interview Problems

5-31. [3] Which data structures are used in depth-first and breath-first search?

5-32. [4] Write a function to traverse binary search tree and return the $i$th node in sorted order.

Programming Challenges

These programming challenge problems with robot judging are available at http://www.programming-challenges.com or http://online-judge.uva.es.

5-1. “Bicoloring” – Programming Challenges 110901, UVA Judge 10004.
5-4. “Edit Step Ladders” – Programming Challenges 110905, UVA Judge 10029.
The data structures and traversal algorithms of Chapter 5 provide the basic building blocks for any computation on graphs. However, all the algorithms presented there dealt with unweighted graphs—i.e., graphs where each edge has identical value or weight.

There is an alternate universe of problems for weighted graphs. The edges of road networks are naturally bound to numerical values such as construction cost, traversal time, length, or speed limit. Identifying the shortest path in such graphs proves more complicated than breadth-first search in unweighted graphs, but opens the door to a wide range of applications.

The graph data structure from Chapter 5 quietly supported edge-weighted graphs, but here we make this explicit. Our adjacency list structure consists of an array of linked lists, such that the outgoing edges from vertex $x$ appear in the list $\text{edges}[x]$:

```c
typedef struct {
    edgenode *edges[MAXV+1]; /* adjacency info */
    int degree[MAXV+1];     /* outdegree of each vertex */
    int nvertices;          /* number of vertices in graph */
    int nedges;             /* number of edges in graph */
    int directed;           /* is the graph directed? */
} graph;
```

Each $\text{edgenode}$ is a record containing three fields, the first describing the second endpoint of the edge ($y$), the second enabling us to annotate the edge with a weight ($\text{weight}$), and the third pointing to the next edge in the list ($\text{next}$):
typedef struct {
    int y;       /* adjacency info */
    int weight;  /* edge weight, if any */
    struct edgenode *next; /* next edge in list */
} edgenode;

We now describe several sophisticated algorithms using this data structure, including minimum spanning trees, shortest paths, and maximum flows. That these optimization problems can be solved efficiently is quite worthy of our respect. Recall that no such algorithm exists for the first weighted graph problem we encountered, namely the traveling salesman problem.

6.1 Minimum Spanning Trees

A spanning tree of a graph \( G = (V, E) \) is a subset of edges from \( E \) forming a tree connecting all vertices of \( V \). For edge-weighted graphs, we are particularly interested in the minimum spanning tree—the spanning tree whose sum of edge weights is as small as possible.

Minimum spanning trees are the answer whenever we need to connect a set of points (representing cities, homes, junctions, or other locations) by the smallest amount of roadway, wire, or pipe. Any tree is the smallest possible connected graph in terms of number of edges, while the minimum spanning tree is the smallest connected graph in terms of edge weight. In geometric problems, the point set \( p_1, \ldots, p_n \) defines a complete graph, with edge \((v_i, v_j)\) assigned a weight equal to the distance from \( p_i \) to \( p_j \). An example of a geometric minimum spanning tree is illustrated in Figure 6.1. Additional applications of minimum spanning trees are discussed in Section 15.3 (page 484).

A minimum spanning tree minimizes the total length over all possible spanning trees. However, there can be more than one minimum spanning tree in a graph. Indeed, all spanning trees of an unweighted (or equally weighted) graph \( G \) are minimum spanning trees, since each contains exactly \( n - 1 \) equal-weight edges. Such a spanning tree can be found using depth-first or breadth-first search. Finding a minimum spanning tree is more difficult for general weighted graphs, however two different algorithms are presented below. Both demonstrate the optimality of certain greedy heuristics.

6.1.1 Prim’s Algorithm

Prim’s minimum spanning tree algorithm starts from one vertex and grows the rest of the tree one edge at a time until all vertices are included.

Greedy algorithms make the decision of what to do next by selecting the best local option from all available choices without regard to the global structure. Since we seek the tree of minimum weight, the natural greedy algorithm for minimum
spanning tree repeatedly selects the smallest weight edge that will enlarge the number of vertices in the tree.

Prim-MST(G)

Select an arbitrary vertex \( s \) to start the tree from.

While (there are still nontree vertices)

Select the edge of minimum weight between a tree and nontree vertex

Add the selected edge and vertex to the tree \( T_{\text{prim}} \).

Prim’s algorithm clearly creates a spanning tree, because no cycle can be introduced by adding edges between tree and nontree vertices. However, why should it be of minimum weight over all spanning trees? We have seen ample evidence of other natural greedy heuristics that do not yield a global optimum. Therefore, we must be particularly careful to demonstrate any such claim.

We use proof by contradiction. Suppose that there existed a graph \( G \) for which Prim’s algorithm did not return a minimum spanning tree. Since we are building the tree incrementally, this means that there must have been some particular instant where we went wrong. Before we inserted edge \((x, y)\), \( T_{\text{prim}} \) consisted of a set of edges that was a subtree of some minimum spanning tree \( T_{\text{min}} \), but choosing edge \((x, y)\) fatally took us away from a minimum spanning tree (see Figure 6.2(a)).

But how could we have gone wrong? There must be a path \( p \) from \( x \) to \( y \) in \( T_{\text{min}} \), as shown in Figure 6.2(b). This path must use an edge \((v_1, v_2)\), where \( v_1 \) is in \( T_{\text{prim}} \), but \( v_2 \) is not. This edge \((v_1, v_2)\) must have weight at least that of \((x, y)\), or Prim’s algorithm would have selected it before \((x, y)\) when it had the chance. Inserting \((x, y)\) and deleting \((v_1, v_2)\) from \( T_{\text{min}} \) leaves a spanning tree no larger than before, meaning that Prim’s algorithm did not make a fatal mistake in selecting edge \((x, y)\). Therefore, by contradiction, Prim’s algorithm must construct a minimum spanning tree.
Implementation

Prim’s algorithm grows the minimum spanning tree in stages, starting from a given vertex. At each iteration, we add one new vertex into the spanning tree. A greedy algorithm suffices for correctness: we always add the lowest-weight edge linking a vertex in the tree to a vertex on the outside. The simplest implementation of this idea would assign each vertex a Boolean variable denoting whether it is already in the tree (the array intree in the code below), and then searches all edges at each iteration to find the minimum weight edge with exactly one intree vertex.

Our implementation is somewhat smarter. It keeps track of the cheapest edge linking every nontree vertex in the tree. The cheapest such edge over all remaining non-tree vertices gets added in each iteration. We must update the costs of getting to the non-tree vertices after each insertion. However, since the most recently inserted vertex is the only change in the tree, all possible edge-weight updates must come from its outgoing edges:

```c
prim(graph *g, int start)
{
    int i; /* counter */
    edgenode *p; /* temporary pointer */
    bool intree[MAXV+1]; /* is the vertex in the tree yet? */
    int distance[MAXV+1]; /* cost of adding to tree */
    int v; /* current vertex to process */
    int w; /* candidate next vertex */
    int weight; /* edge weight */
    int dist; /* best current distance from start */

    for (i=1; i<=g->nvertices; i++) {
        intree[i] = FALSE;
```
6.1 MINIMUM SPANNING TREES

```c

distance[i] = MAXINT;
pARENT[i] = -1;
}

distance[start] = 0;
v = start;

while (intree[v] == FALSE) {
    intree[v] = TRUE;
p = g->edges[v];
    while (p != NULL) {
        w = p->y;
        weight = p->weight;
        if ((distance[w] > weight) && (intree[w] == FALSE)) {
            distance[w] = weight;
pARENT[w] = v;
        }
p = p->next;
    }
    v = 1;
dist = MAXINT;
    for (i=1; i<=g->nvertices; i++)
        if ((intree[i] == FALSE) && (dist > distance[i])) {
            dist = distance[i];
v = i;
        }
}
```

**Analysis**

Prim’s algorithm is correct, but how efficient is it? This depends on which data structures are used to implement it. In the pseudocode, Prim’s algorithm makes $n$ iterations sweeping through all $m$ edges on each iteration—yielding an $O(mn)$ algorithm.

But our implementation avoids the need to test all $m$ edges on each pass. It only considers the $\leq n$ cheapest known edges represented in the parent array and the $\leq n$ edges out of new tree vertex $v$ to update parent. By maintaining a Boolean flag along with each vertex to denote whether it is in the tree or not, we test whether the current edge joins a tree with a non-tree vertex in constant time.

The result is an $O(n^2)$ implementation of Prim’s algorithm, and a good illustration of power of data structures to speed up algorithms. In fact, more sophisticated
priority-queue data structures lead to an $O(m + n \lg n)$ implementation, by making it faster to find the minimum cost edge to expand the tree at each iteration.

The minimum spanning tree itself or its cost can be reconstructed in two different ways. The simplest method would be to augment this procedure with statements that print the edges as they are found or totals the weight of all selected edges. Alternately, the tree topology is encoded by the `parent` array, so it plus the original graph describe everything about the minimum spanning tree.

### 6.1.2 Kruskal’s Algorithm

Kruskal’s algorithm is an alternate approach to finding minimum spanning trees that proves more efficient on sparse graphs. Like Prim’s, Kruskal’s algorithm is greedy. Unlike Prim’s, it does not start with a particular vertex.

Kruskal’s algorithm builds up connected components of vertices, culminating in the minimum spanning tree. Initially, each vertex forms its own separate component in the tree-to-be. The algorithm repeatedly considers the lightest remaining edge and tests whether its two endpoints lie within the same connected component. If so, this edge will be discarded, because adding it would create a cycle in the tree-to-be. If the endpoints are in different components, we insert the edge and merge the two components into one. Since each connected component is always a tree, we need never explicitly test for cycles.

Kruskal-MST($G$)

Put the edges in a priority queue ordered by weight.

`count = 0`

`while (count < n - 1) do`

`get next edge (v, w)`

`if (component(v) \neq component(w))`

`add to $T_{kruskal}$`

`merge component(v) and component(w)`
This algorithm adds \( n - 1 \) edges without creating a cycle, so it clearly creates a spanning tree for any connected graph. But why must this be a minimum spanning tree? Suppose it wasn’t. As with the correctness proof of Prim’s algorithm, there must be some graph on which it fails. In particular, there must be a single edge \((x, y)\) whose insertion first prevented the tree \( T_{\text{kruskal}} \) from being a minimum spanning tree \( T_{\text{min}} \). Inserting this edge \((x, y)\) into \( T_{\text{min}} \) will create a cycle with the path from \( x \) to \( y \). Since \( x \) and \( y \) were in different components at the time of inserting \((x, y)\), at least one edge (say \((v_1, v_2)\)) on this path would have been evaluated by Kruskal’s algorithm later than \((x, y)\). But this means that \( w(v_1, v_2) \geq w(x, y) \), so exchanging the two edges yields a tree of weight at most \( T_{\text{min}} \). Therefore, we could not have made a fatal mistake in selecting \((x, y)\), and the correctness follows.

What is the time complexity of Kruskal’s algorithm? Sorting the \( m \) edges takes \( O(m \log m) \) time. The for loop makes \( m \) iterations, each testing the connectivity of two trees plus an edge. In the most simple-minded approach, this can be implemented by breadth-first or depth-first search in a sparse graph with at most \( n \) edges and \( n \) vertices, thus yielding an \( O(mn) \) algorithm.

However, a faster implementation results if we can implement the component test in faster than \( O(n) \) time. In fact, a clever data structure called union-find, can support such queries in \( O(\log n) \) time. Union-find is discussed in the next section. With this data structure, Kruskal’s algorithm runs in \( O(m \log m) \) time, which is faster than Prim’s for sparse graphs. Observe again the impact that the right data structure can have when implementing a straightforward algorithm.

**Implementation**

The implementation of the main routine follows fairly directly from the pseudocode:
kruskal(graph *g)
{
    int i; /* counter */
    set_union s; /* set union data structure */
    edge_pair e[MAXV+1]; /* array of edges data structure */
    bool weight_compare();

    set_union_init(&s, g->nvertices);

to_edge_array(g, e); /* sort edges by increasing cost */
qsort(&e,g->nedges,sizeof(edge_pair),weight_compare);

    for (i=0; i<(g->nedges); i++) {
        if (!same_component(s,e[i].x,e[i].y)) {
            printf("edge (%d,%d) in MST\n",e[i].x,e[i].y);
            union_sets(&s,e[i].x,e[i].y);
        }
    }
}

6.1.3 The Union-Find Data Structure

A set partition is a partitioning of the elements of some universal set (say the integers 1 to $n$) into a collection of disjointed subsets. Thus, each element must be in exactly one subset. Set partitions naturally arise in graph problems such as connected components (each vertex is in exactly one connected component) and vertex coloring (a person may be male or female, but not both or neither). Section 14.6 (page 456) presents algorithms for generating set partitions and related objects.

The connected components in a graph can be represented as a set partition. For Kruskal’s algorithm to run efficiently, we need a data structure that efficiently supports the following operations:

- **Same component($v_1,v_2$)** – Do vertices $v_1$ and $v_2$ occur in the same connected component of the current graph?

- **Merge components($C_1,C_2$)** – Merge the given pair of connected components into one component in response to an edge between them.

The two obvious data structures for this task each support only one of these operations efficiently. Explicitly labeling each element with its component number enables the same component test to be performed in constant time, but updating the component numbers after a merger would require linear time. Alternately, we can treat the merge components operation as inserting an edge in a graph, but
then we must run a full graph traversal to identify the connected components on demand.

The union-find data structure represents each subset as a “backwards” tree, with pointers from a node to its parent. Each node of this tree contains a set element, and the name of the set is taken from the key at the root. For reasons that will become clear, we will also maintain the number of elements in the subtree rooted in each vertex \( v \):

```c
typedef struct {
    int p[SET_SIZE+1]; /* parent element */
    int size[SET_SIZE+1]; /* number of elements in subtree i */
    int n; /* number of elements in set */
} set_union;
```

We implement our desired component operations in terms of two simpler operations, `union` and `find`:

- **Find(\( i \))** – Find the root of tree containing element \( i \), by walking up the parent pointers until there is nowhere to go. Return the label of the root.

- **Union(\( i,j \))** – Link the root of one of the trees (say containing \( i \)) to the root of the tree containing the other (say \( j \)) so `find(\( i \))` now equals `find(\( j \))`.

We seek to minimize the time it takes to execute any sequence of unions and finds. Tree structures can be very unbalanced, so we must limit the height of our trees. Our most obvious means of control is the decision of which of the two component roots becomes the root of the combined component on each `union`.

To minimize the tree height, it is better to make the smaller tree the subtree of the bigger one. Why? The height of all the nodes in the root subtree stay the same, while the height of the nodes merged into this tree all increase by one. Thus, merging in the smaller tree leaves the height unchanged on the larger set of vertices.
**Implementation**

The implementation details are as follows:

```c
set_union_init(set_union *s, int n)
{
    int i; /* counter */

    for (i=1; i<=n; i++) {
        s->p[i] = i;
        s->size[i] = 1;
    }

    s->n = n;
}

int find(set_union *s, int x)
{
    if (s->p[x] == x)
        return(x);
    else
        return( find(s,s->p[x]) );
}

int union_sets(set_union *s, int s1, int s2)
{
    int r1, r2; /* roots of sets */

    r1 = find(s,s1);
    r2 = find(s,s2);

    if (r1 == r2) return; /* already in same set */
    if (s->size[r1] >= s->size[r2]) {
        s->size[r1] = s->size[r1] + s->size[r2];
        s->p[r2] = r1;
    } else {
        s->size[r2] = s->size[r1] + s->size[r2];
        s->p[r1] = r2;
    }
}

bool same_component(set_union *s, int s1, int s2)
{
    return ( find(s,s1) == find(s,s2) );
}
```
Analysis

On each union, the tree with fewer nodes becomes the child. But how tall can such a tree get as a function of the number of nodes in it? Consider the smallest possible tree of height \( h \). Single-node trees have height 1. The smallest tree of height-2 has two nodes; from the union of two single-node trees. When do we increase the height? Merging in single-node trees won’t do it, since they just become children of the rooted tree of height-2. Only when we merge two height-2 trees together do we get a tree of height-3, now with four nodes.

See the pattern? We must double the number of nodes in the tree to get an extra unit of height. How many doublings can we do before we use up all \( n \) nodes? At most, \( \log_2 n \) doublings can be performed. Thus, we can do both unions and finds in \( O(\log n) \), good enough for Kruskal’s algorithm. In fact, union-find can be done even faster, as discussed in Section 12.5 (page 385).

6.1.4 Variations on Minimum Spanning Trees

This minimum spanning tree algorithm has several interesting properties that help solve several closely related problems:

- **Maximum Spanning Trees** – Suppose an evil telephone company is contracted to connect a bunch of houses together; they will be paid a price proportional to the amount of wire they install. Naturally, they will build the most expensive spanning tree possible. The maximum spanning tree of any graph can be found by simply negating the weights of all edges and running Prim’s algorithm. The most negative tree in the negated graph is the maximum spanning tree in the original.

Most graph algorithms do not adapt so easily to negative numbers. Indeed, shortest path algorithms have trouble with negative numbers, and certainly do not generate the longest possible path using this technique.

- **Minimum Product Spanning Trees** – Suppose we seek the spanning tree that minimizes the product of edge weights, assuming all edge weights are positive. Since \( \log(a \cdot b) = \log(a) + \log(b) \), the minimum spanning tree on a graph whose edge weights are replaced with their logarithms gives the minimum product spanning tree on the original graph.

- **Minimum Bottleneck Spanning Tree** – Sometimes we seek a spanning tree that minimizes the maximum edge weight over all such trees. In fact, every minimum spanning tree has this property. The proof follows directly from the correctness of Kruskal’s algorithm.

Such bottleneck spanning trees have interesting applications when the edge weights are interpreted as costs, capacities, or strengths. A less efficient
but conceptually simpler way to solve such problems might be to delete all “heavy” edges from the graph and ask whether the result is still connected. These kind of tests can be done with simple BFS/DFS.

The minimum spanning tree of a graph is unique if all \( m \) edge weights in the graph are distinct. Otherwise the order in which Prim’s/Kruskal’s algorithm breaks ties determines which minimum spanning tree is returned.

There are two important variants of a minimum spanning tree that are not solvable with these techniques.

- **Steiner Tree** – Suppose we want to wire a bunch of houses together, but have the freedom to add extra intermediate vertices to serve as a shared junction. This problem is known as a *minimum Steiner tree*, and is discussed in the catalog in Section 16.10.

- **Low-degree Spanning Tree** – Alternately, what if we want to find the minimum spanning tree where the highest degree node in the tree is small? The lowest max-degree tree possible would be a simple path, and have \( n - 2 \) nodes of degree 2 with two endpoints of degree 1. A path that visits each vertex once is called a *Hamiltonian path*, and is discussed in the catalog in Section 16.5.

### 6.2 War Story: Nothing but Nets

I’d been tipped off about a small printed-circuit board testing company nearby in need of some algorithmic consulting. And so I found myself inside a nondescript building in a nondescript industrial park, talking with the president of Integri-Test and one of his lead technical people.

“We’re leaders in robotic printed-circuit board testing devices. Our customers have very high reliability requirements for their PC-boards. They must check that each and every board has no wire breaks before filling it with components. This means testing that each and every pair of points on the board that are supposed to be connected are connected.”

“How do you do the testing?” I asked.

“We have a robot with two arms, each with electric probes. The arms simultaneously contact both of the points to test whether two points are properly connected. If they are properly connected, then the probes will complete a circuit. For each net, we hold one arm fixed at one point and move the other to cover the rest of the points.”

“Wait!” I cried. “What is a net?”
Figure 6.6: An example net showing (a) the metal connection layer, (b) the contact points, (c) their minimum spanning tree, and (d) the points partitioned into clusters.
The president scribbled down some notes and then frowned. “Fine. Maybe you can order the points in a net better for us. But that’s not our real problem. When you watch our robot in action, the right arm sometimes has to run all the way to the right side of the board on a given net, while the left arm just sits there. It seems we would benefit by breaking nets into smaller pieces to balance things out.”

I sat down and thought. The left and right arms each have interlocking TSP problems to solve. The left arm would move between the leftmost points of each net, while the right arm to visits all the other points in each net as ordered by the left TSP tour. By breaking each net into smaller nets we would avoid making the right arm cross all the way across the board. Further, a lot of little nets meant there would be more points in the left TSP, so each left-arm movement was likely to be short, too.

“You are right. We should win if we can break big nets into small nets. We want the nets to be small, both in the number of points and in physical area. But we must be sure that if we validate the connectivity of each small net, we will have confirmed that the big net is connected. One point in common between two little nets is sufficient to show that the bigger net formed by the two little nets is connected, since current can flow between any pair of points.”

Now we had to break each net into overlapping pieces, where each piece was small. This is a clustering problem. Minimum spanning trees are often used for clustering, as discussed in Section 15.3 (page 484). In fact, that was the answer! We could find the minimum spanning tree of the net points and break it into little clusters whenever a spanning tree edge got too long. As shown in Figure 6.6(d), each cluster would share exactly one point in common with another cluster, with connectivity ensured because we are covering the edges of a spanning tree. The shape of the clusters will reflect the points in the net. If the points lay along a line across the board, the minimum spanning tree would be a path, and the clusters would be pairs of points. If the points all fell in a tight region, there would be one nice fat cluster for the right arm to scoot around.

So I explained the idea of constructing the minimum spanning tree of a graph. The boss listened, scribbled more notes, and frowned again.

“I like your clustering idea. But minimum spanning trees are defined on graphs. All you’ve got are points. Where do the weights of the edges come from?”

“Oh, we can think of it as a complete graph, where every pair of points are connected. The weight of the edge is defined as the distance between the two points. Or is it . . .?”

I went back to thinking. The edge cost should reflect the travel time between two points. While distance is related to travel time, it wasn’t necessarily the same thing.

“Hey. I have a question about your robot. Does it take the same amount of time to move the arm left-right as it does up-down?”

They thought a minute. “Yeah, it does. We use the same type of motor to control horizontal and vertical movements. Since the two motors for each arm are
independent, we can simultaneously move each arm both horizontally and vertically."

“So the time to move both one foot left and one foot up is exactly the same as just moving one foot left? This means that the weight for each edge should not be the Euclidean distance between the two points, but the biggest difference between either the $x-$ or $y$-coordinate. This is something we call the $L_{\infty}$ metric, but we can capture it by changing the edge weights in the graph. Anything else funny about your robots?” I asked.

“Well, it takes some time for the robot to come up to speed. We should probably also factor in acceleration and deceleration of the arms.”

“Darn right. The more accurately you can model the time your arm takes to move between two points, the better our solution will be. But now we have a very clean formulation. Let’s code it up and let’s see how well it works!”

They were somewhat skeptical whether this approach would do any good, but agreed to think about it. A few weeks later they called me back and reported that the new algorithm reduced the distance traveled by about $30\%$ over their previous approach, at a cost of a little more computational preprocessing. However, since their testing machine cost $\$200,000$ a pop and a PC cost $\$2,000$, this was an excellent tradeoff. It is particularly advantageous since the preprocessing need only be done once when testing multiple instances of a particular board design.

The key idea leading to the successful solution was modeling the job in terms of classical algorithmic graph problems. I smelled TSP the instant they started talking about minimizing robot motion. Once I realized that they were implicitly forming a star-shaped spanning tree to ensure connectivity, it was natural to ask whether the minimum spanning tree would perform any better. This idea led to clustering, and thus partitioning each net into smaller nets. Finally, by carefully designing our distance metric to accurately model the costs of the robot itself, we could incorporate such complicated properties (as acceleration) without changing our fundamental graph model or algorithm design.

*Take-Home Lesson:* Most applications of graphs can be reduced to standard graph properties where well-known algorithms can be used. These include minimum spanning trees, shortest paths, and other problems presented in the catalog.

### 6.3 Shortest Paths

A *path* is a sequence of edges connecting two vertices. Since movie director Mel Brooks ("The Producers") is my father’s sister’s husband’s cousin, there is a path in the friendship graph between me and him, shown in Figure 6.7—even though the two of us have never met. But if I were trying to impress how tight I am with Cousin Mel, I would be much better off saying that my Uncle Lenny grew up with him. I have a friendship path of length 2 to Cousin Mel through Uncle Lenny, while
the path is of length 4 by blood and marriage. This multiplicity of paths hints at why finding the shortest path between two nodes is important and instructive, even in nontransportation applications.

The shortest path from $s$ to $t$ in an unweighted graph can be constructed using a breadth-first search from $s$. The minimum-link path is recorded in the breadth-first search tree, and it provides the shortest path when all edges have equal weight.

However, BFS does not suffice to find shortest paths in weighted graphs. The shortest weighted path might use a large number of edges, just as the shortest route (timewise) from home to office may involve complicated shortcuts using backroads, as shown in Figure 6.8.

In this section, we will present two distinct algorithms for finding the shortest paths in weighted graphs.

### 6.3.1 Dijkstra’s Algorithm

Dijkstra’s algorithm is the method of choice for finding shortest paths in an edge- and/or vertex-weighted graph. Given a particular start vertex $s$, it finds the shortest path from $s$ to every other vertex in the graph, including your desired destination $t$.

Suppose the shortest path from $s$ to $t$ in graph $G$ passes through a particular intermediate vertex $x$. Clearly, this path must contain the shortest path from $s$ to $x$ as its prefix, because if not, we could shorten our $s$-to-$t$ path by using the shorter
s-to-x prefix. Thus, we must compute the shortest path from s to x before we find the path from s to t.

Dijkstra’s algorithm proceeds in a series of rounds, where each round establishes the shortest path from s to some new vertex. Specifically, x is the vertex that minimizes \( \text{dist}(s, v_i) + w(v_i, x) \) over all unfinished \( 1 \leq i \leq n \), where \( w(i, j) \) is the length of the edge from i to j, and \( \text{dist}(i, j) \) is the length of the shortest path between them.

This suggests a dynamic programming-like strategy. The shortest path from s to itself is trivial unless there are negative weight edges, so \( \text{dist}(s, s) = 0 \). If \( (s, y) \) is the lightest edge incident to s, then this implies that \( \text{dist}(s, y) = w(s, y) \). Once we determine the shortest path to a node x, we check all the outgoing edges of x to see whether there is a better path from s to some unknown vertex through x.

\[
\text{ShortestPath-Dijkstra}(G, s, t) \\
\text{known} = \{s\} \\
\text{for } i = 1 \text{ to } n, \text{dist}[i] = \infty \\
\text{for each edge } (s, v), \text{dist}[v] = w(s, v) \\
\text{last} = s \\
\text{while } (\text{last} \neq t) \\
\quad \text{select } v_{\text{next}}, \text{the unknown vertex minimizing } \text{dist}[v] \\
\quad \text{for each edge } (v_{\text{next}}, x), \text{dist}[x] = \min[\text{dist}[x], \text{dist}[v_{\text{next}}] + w(v_{\text{next}}, x)] \\
\quad \text{last} = v_{\text{next}} \\
\quad \text{known} = \text{known} \cup \{v_{\text{next}}\}
\]

The basic idea is very similar to Prim’s algorithm. In each iteration, we add exactly one vertex to the tree of vertices for which we know the shortest path from s. As in Prim’s, we keep track of the best path seen to date for all vertices outside the tree, and insert them in order of increasing cost.

The difference between Dijkstra’s and Prim’s algorithms is how they rate the desirability of each outside vertex. In the minimum spanning tree problem, all we cared about was the weight of the next potential tree edge. In shortest path, we want to include the closest outside vertex (in shortest-path distance) to s. This is a function of both the new edge weight and the distance from s to the tree vertex it is adjacent to.

**Implementation**

The pseudocode actually obscures how similar the two algorithms are. In fact, the change is very minor. Below, we give an implementation of Dijkstra’s algorithm based on changing exactly three lines from our Prim’s implementation—one of which is simply the name of the function!
dijkstra(graph *g, int start)  /* WAS prim(g,start) */
{
    int i;    /* counter */
edgenode *p;    /* temporary pointer */
bool intree[MAXV+1];    /* is the vertex in the tree yet? */
int distance[MAXV+1];    /* distance vertex is from start */
int v;    /* current vertex to process */
int w;    /* candidate next vertex */
int weight;    /* edge weight */
int dist;    /* best current distance from start */

    for (i=1; i<=g->nvertices; i++) {
        intree[i] = FALSE;
        distance[i] = MAXINT;
        parent[i] = -1;
    }

distance[start] = 0;
v = start;

    while (intree[v] == FALSE) {
        intree[v] = TRUE;
p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            weight = p->weight;
            /* CHANGED */            if (distance[w] > (distance[v]+weight)) {
                /* CHANGED */                distance[w] = distance[v]+weight;
                /* CHANGED */                parent[w] = v;
            }
p = p->next;
        }

        v = 1;
dist = MAXINT;
        for (i=1; i<=g->nvertices; i++)
            if ((intree[i] == FALSE) &
                (dist > distance[i])) {
                dist = distance[i];
v = i;
            }
    }
}
This algorithm finds more than just the shortest path from $s$ to $t$. It finds the shortest path from $s$ to all other vertices. This defines a shortest path spanning tree rooted in $s$. For undirected graphs, this would be the breadth-first search tree, but in general it provides the shortest path from $s$ to all other vertices.

**Analysis**

What is the running time of Dijkstra’s algorithm? As implemented here, the complexity is $O(n^2)$. This is the same running time as a proper version of Prim’s algorithm; except for the extension condition it is the same algorithm as Prim’s.

The length of the shortest path from start to a given vertex $t$ is exactly the value of distance[$t$]. How do we use dijkstra to find the actual path? We follow the backward parent pointers from $t$ until we hit start (or -1 if no such path exists), exactly as was done in the find_path() routine of Section 5.6.2 (page 165).

Dijkstra works correctly only on graphs without negative-cost edges. The reason is that midway through the execution we may encounter an edge with weight so negative that it changes the cheapest way to get from $s$ to some other vertex already in the tree. Indeed, the most cost-effective way to get from your house to your next-door neighbor would be repeatedly through the lobby of any bank offering you enough money to make the detour worthwhile.

Most applications do not feature negative-weight edges, making this discussion academic. Floyd’s algorithm, discussed below, works correctly unless there are negative cost cycles, which grossly distort the shortest-path structure. Unless that bank limits its reward to one per customer, you might so benefit by making an infinite number of trips through the lobby that you would never decide to actually reach your destination!

**Stop and Think: Shortest Path with Node Costs**

*Problem:* Suppose we are given a graph whose weights are on the vertices, instead of the edges. Thus, the cost of a path from $x$ to $y$ is the sum of the weights of all vertices on the path.

Give an efficient algorithm for finding shortest paths on vertex-weighted graphs.

---

*Solution:* A natural idea would be to adapt the algorithm we have for edge-weighted graphs (Dijkstra’s) to the new vertex-weighted domain. It should be clear that we can do it. We replace any reference to the weight of an edge with the weight of the destination vertex. This can be looked up as needed from an array of vertex weights.

However, my preferred approach would leave Dijkstra’s algorithm intact and instead concentrate on constructing an edge-weighted graph on which Dijkstra’s
algorithm will give the desired answer. Set the weight of each directed edge \((i,j)\) in the input graph to the cost of vertex \(j\). Dijkstra’s algorithm now does the job.

This technique can be extended to a variety of different domains, such as when there are costs on both vertices and edges.

### 6.3.2 All-Pairs Shortest Path

Suppose you want to find the “center” vertex in a graph—the one that minimizes the longest or average distance to all the other nodes. This might be the best place to start a new business. Or perhaps you need to know a graph’s diameter—the longest shortest-path distance over all pairs of vertices. This might correspond to the longest possible time it takes a letter or network packet to be delivered. These and other applications require computing the shortest path between all pairs of vertices in a given graph.

We could solve all-pairs shortest path by calling Dijkstra’s algorithm from each of the \(n\) possible starting vertices. But Floyd’s all-pairs shortest-path algorithm is a slick way to construct this \(n \times n\) distance matrix from the original weight matrix of the graph.

Floyd’s algorithm is best employed on an adjacency matrix data structure, which is no extravagance since we must store all \(n^2\) pairwise distances anyway. Our adjacency_matrix type allocates space for the largest possible matrix, and keeps track of how many vertices are in the graph:

```c
typedef struct {
    int weight[MAXV+1][MAXV+1]; /* adjacency/weight info */
    int nvertices; /* number of vertices in graph */
} adjacency_matrix;
```

The critical issue in an adjacency matrix implementation is how we denote the edges absent from the graph. A common convention for unweighted graphs denotes graph edges by 1 and non-edges by 0. This gives exactly the wrong interpretation if the numbers denote edge weights, for the non-edges get interpreted as a free ride between vertices. Instead, we should initialize each non-edge to \(\text{MAXINT}\). This way we can both test whether it is present and automatically ignore it in shortest-path computations, since only real edges will be used, provided \(\text{MAXINT}\) is less than the diameter of your graph.

There are several ways to characterize the shortest path between two nodes in a graph. The Floyd-Warshall algorithm starts by numbering the vertices of the graph from 1 to \(n\). We use these numbers not to label the vertices, but to order them. Define \(W[i,j]^k\) to be the length of the shortest path from \(i\) to \(j\) using only vertices numbered from 1, 2, ..., \(k\) as possible intermediate vertices.

What does this mean? When \(k = 0\), we are allowed no intermediate vertices, so the only allowed paths are the original edges in the graph. Thus the initial
all-pairs shortest-path matrix consists of the initial adjacency matrix. We will perform \( n \) iterations, where the \( k \)th iteration allows only the first \( k \) vertices as possible intermediate steps on the path between each pair of vertices \( x \) and \( y \).

At each iteration, we allow a richer set of possible shortest paths by adding a new vertex as a possible intermediary. Allowing the \( k \)th vertex as a stop helps only if there is a short path that goes through \( k \), so

\[
W[i, j]^k = \min(W[i, j]^{k-1}, W[i, k]^{k-1} + W[k, j]^{k-1})
\]

The correctness of this is somewhat subtle, and I encourage you to convince yourself of it. But there is nothing subtle about how simple the implementation is:

```c
floyd(adjacency_matrix *g)
{
    int i,j; /* dimension counters */
    int k; /* intermediate vertex counter */
    int through_k; /* distance through vertex k */

    for (k=1; k<=g->nvertices; k++)
        for (i=1; i<=g->nvertices; i++)
            for (j=1; j<=g->nvertices; j++) {
                through_k = g->weight[i][k]+g->weight[k][j];
                if (through_k < g->weight[i][j])
                    g->weight[i][j] = through_k;
            }
}
```

The Floyd-Warshall all-pairs shortest path runs in \( O(n^3) \) time, which is asymptotically no better than \( n \) calls to Dijkstra’s algorithm. However, the loops are so tight and the program so short that it runs better in practice. It is notable as one of the rare graph algorithms that work better on adjacency matrices than adjacency lists.

The output of Floyd’s algorithm, as it is written, does not enable one to reconstruct the actual shortest path between any given pair of vertices. These paths can be recovered if we retain a parent matrix \( P \) of our choice of the last intermediate vertex used for each vertex pair \((x, y)\). Say this value is \( k \). The shortest path from \( x \) to \( y \) is the concatenation of the shortest path from \( x \) to \( k \) with the shortest path from \( k \) to \( y \), which can be reconstructed recursively given the matrix \( P \). Note, however, that most all-pairs applications need only the resulting distance matrix. These jobs are what Floyd’s algorithm was designed for.
6.3.3 Transitive Closure

Floyd’s algorithm has another important application, that of computing transitive closure. In analyzing a directed graph, we are often interested in which vertices are reachable from a given node.

As an example, consider the blackmail graph, where there is a directed edge \((i, j)\) if person \(i\) has sensitive-enough private information on person \(j\) so that \(i\) can get \(j\) to do whatever he wants. You wish to hire one of these \(n\) people to be your personal representative. Who has the most power in terms of blackmail potential?

A simplistic answer would be the vertex of highest degree, but an even better representative would be the person who has blackmail chains leading to the most other parties. Steve might only be able to blackmail Miguel directly, but if Miguel can blackmail everyone else then Steve is the man you want to hire.

The vertices reachable from any single node can be computed using breadth-first or depth-first searches. But the whole batch can be computed using an all-pairs shortest-path. If the shortest path from \(i\) to \(j\) remains MAXINT after running Floyd’s algorithm, you can be sure no directed path exists from \(i\) to \(j\). Any vertex pair of weight less than MAXINT must be reachable, both in the graph-theoretic and blackmail senses of the word.

Transitive closure is discussed in more detail in the catalog in Section 15.5.

6.4 War Story: Dialing for Documents

I was part of a group visiting Periphonics, which was then an industry leader in building telephone voice-response systems. These are more advanced versions of the Press 1 for more options, Press 2 if you didn’t press 1 telephone systems that blight everyone’s lives. We were being given the standard tour when someone from our group asked, “Why don’t you guys use voice recognition for data entry. It would be a lot less annoying than typing things out on the keypad.”

The tour guide reacted smoothly. “Our customers have the option of incorporating speech recognition into our products, but very few of them do. User-independent, connected-speech recognition is not accurate enough for most applications. Our customers prefer building systems around typing text on the telephone keyboards.”

“Prefer typing, my pupik!” came a voice from the rear of our group. “I hate typing on a telephone. Whenever I call my brokerage house to get stock quotes some machine tells me to type in the three letter code. To make things worse, I have to hit two buttons to type in one letter, in order to distinguish between the three letters printed on each key of the telephone. I hit the 2 key and it says Press 1 for A, Press 2 for B, Press 3 for C. Pain in the neck if you ask me.”

“Maybe you don’t have to hit two keys for each letter!” I chimed in. “Maybe the system could figure out the correct letter from context!”
“There isn’t a whole lot of context when you type in three letters of stock market code.”

“Sure, but there would be plenty of context if we typed in English sentences. I’ll bet that we could reconstruct English text correctly if they were typed in a telephone at one keystroke per letter.”

The guy from Periphonics gave me a disinterested look, then continued the tour. But when I got back to the office, I decided to give it a try.

Not all letters are equally likely to be typed on a telephone. In fact, not all letters can be typed, since Q and Z are not labeled on a standard American telephone. Therefore, we adopted the convention that Q, Z, and “space” all sat on the * key. We could take advantage of the uneven distribution of letter frequencies to help us decode the text. For example, if you hit the 3 key while typing English, you more likely meant to type an E than either a D or F. By taking into account the frequencies of a window of three characters (trigrams), we could predict the typed text. This is what happened when I tried it on the Gettysburg Address:

```
enurraore ane reten yearr ain our ectherr arortght eostoi on ugis aootinent a oey oation aoncdiveve in licesty ane eedicatee un uhe rrrosisition uiat all oen are arectee e ual
ony ye are enichde in a irect ailti yar uestini yhethes uiat oatíoor o aoy oation ro aoncdivéeve ane ro eedicatee aan loni enure ye are oet on a irect aattlediele oe uiat yar ye iate aone un eedicate a rostion oe uiat eiele ar a einaal resitni rlace eor uio reir yin iere iate uhdís uiat uhe oation oggh live it is alaniethes cittini ane rrrores uiat ye rioule en uigir
att in a laries reore ye aan ouu eedicate ye aan oou aoorrearate ye aan oou alaloy ugis irome the arate oen litini ane eee ain rustgiiee iere iate aoorrearatee it eat aante our roor rows un ade or eearat the yople yill littl oote oor loni renences yiat ye ray iere att it aan oetes eoses iyi uhyf eie iere it is eor ur uhe litini rathes un ae eedicatee iere un uhe undiniside yopl yhici uhyf yin entght iere iate uiu ear ro onaky aeetancde it is rathes eor ur un ae iere eedicatee un uhe irect uarl renceiiini adeore ur uiat eron where ioooree eee ye uale inarareee eeuotion uo tiaat aaur eor yhici uhyf iere iate uhe lart eull oearure oe eeuotioo tiat ye iere iggily rerolue uiat where eee riall oou iate eide io
```

The trigram statistics did a decent job of translating it into Greek, but a terrible job of transcribing English. One reason was clear. This algorithm knew nothing about English words. If we coupled it with a dictionary, we might be onto something. But two words in the dictionary are often represented by the exact same string of phone codes. For an extreme example, the code string “22737” collides with eleven distinct English words, including cases, cares, cards, capes, caper, and bases. For our next attempt, we reported the unambiguous characters of any words that collided in the dictionary, and used trigrams to fill in the rest of the characters. We were rewarded with:

```
eourscore and seven yearr ain our estherr brought forth on this continent azyoey nation conceiwe in liberty and dedicatee uo uhe proposition that all men are createe equal
ony ye are engagee in azipeat civil yar uestioi whether that nation or aoy nation ro conceiwee and ro dedicatee aanzee long endure ye are oet on azipeat battlefield oe that yar ye iate aone uo dedicate a rostion oe that field ar a final perthni place for those yin here iate their lives that uhe nation oight live it is altogether fittinizean proper that ye should en this
aut in a larges sense ye aan ouu dedicate ye aan ouu consecrate ye aan ouu hallow this ground the arate men litioi and deae yin strugglee here iate consecratee it ear above our roor power uo ade or detract the world will little oote oor long remember what ye
```
If you were a student of American history, maybe you could recognize it, but you certainly couldn’t read it. Somehow, we had to distinguish between the different dictionary words that got hashed to the same code. We could factor in the relative popularity of each word, but this still made too many mistakes.

At this point, I started working with Harald Rau on the project, who proved to be a great collaborator. First, he was a bright and persistent graduate student. Second, as a native German speaker, he believed every lie I told him about English grammar.

Harald built up a phone code reconstruction program along the lines of Figure 6.9. It worked on the input one sentence at a time, identifying dictionary words that matched each code string. The key problem was how to incorporate grammatical constraints.

“We can get good word-use frequencies and grammatical information from a big text database called the Brown Corpus. It contains thousands of typical English sentences, each parsed according to parts of speech. But how do we factor it all in?” Harald asked.

“Let’s think about it as a graph problem,” I suggested.

“Graph problem? What graph problem? Where is there even a graph?”

“Think of a sentence as a series of phone tokens, each representing a word in the sentence. Each phone token has a list of words from the dictionary that match it. How can we choose which one is right? Each possible sentence interpretation can be thought of as a path in a graph. The vertices of this graph are the complete set of possible word choices. There will be an edge from each possible choice for the $i$th word to each possible choice for the $(i + 1)$st word. The cheapest path across this graph defines the best interpretation of the sentence.”

“But all the paths look the same. They have the same number of edges. Wait. Now I see! We have to add weight to the edges to make the paths different.”

“Exactly! The cost of an edge will reflect how likely it is that we will travel through the given pair of words. Perhaps we can count how often that pair of words occurred together in previous texts. Or we can weigh them by the part of speech of each word. Maybe nouns don’t like to be next to nouns as much as they like being next to verbs.”

“It will be hard to keep track of word-pair statistics, since there are so many of them. But we certainly know the frequency of each word. How can we factor that into things?”
“We can pay a cost for walking through a particular vertex that depends upon the frequency of the word. Our best sentence will be given by the shortest path across the graph.”

“But how do we figure out the relative weights of these factors?”

“First try what seems natural to you and then we can experiment with it.”

Harald incorporated this shortest-path algorithm. With proper grammatical and statistical constraints, the system performed great. Look at the Gettysburg Address now, with all the reconstruction errors highlighted:

FOURSORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH ON THIS CONTINENT A NEW NATION CONCEIVED IN LIBERTY AND DEDICATED TO THE PROPOSITION THAT ALL MEN ARE CREATED EQUAL. NOW WE ARE
Figure 6.10: The minimum-cost path defines the best interpretation for a sentence

<table>
<thead>
<tr>
<th>Text</th>
<th>characters</th>
<th>characters correct</th>
<th>non-blanks correct</th>
<th>words correct</th>
<th>time per character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clinton Speeches</td>
<td>1,073,593</td>
<td>99.04%</td>
<td>98.86%</td>
<td>97.67%</td>
<td>0.97ms</td>
</tr>
<tr>
<td>Herland</td>
<td>278,670</td>
<td>98.24%</td>
<td>97.89%</td>
<td>97.02%</td>
<td>0.97ms</td>
</tr>
<tr>
<td>Moby Dick</td>
<td>1,123,581</td>
<td>96.85%</td>
<td>96.25%</td>
<td>94.75%</td>
<td>1.14ms</td>
</tr>
<tr>
<td>Bible</td>
<td>3,961,684</td>
<td>96.20%</td>
<td>95.39%</td>
<td>95.39%</td>
<td>1.33ms</td>
</tr>
<tr>
<td>Shakespeare</td>
<td>4,558,202</td>
<td>95.20%</td>
<td>94.21%</td>
<td>92.86%</td>
<td>0.99ms</td>
</tr>
</tbody>
</table>

Figure 6.11: Telephone-code reconstruction applied to several text samples

Engaged in a great civil war testing whether that nation or any nation so conceived and so dedicated can long endure. We are met on a great battlefield of that was. We have come to dedicate a portion of that field as a final serving place for those who here have their lives that the nation might live. It is altogether fitting and proper that we should do this. But in a larger sense we can not dedicate we can not consecrate we can not hallow this ground. The brave men living and dead who struggled here have consecrated it far above our poor power to add or detract. The world will little note nor long remember what we say here but it can never forget what they did here. It is for us the living rather to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us that from these honored dead we take increased devotion to that cause for which they here have devoted the last full measure of devotion that we here highly resolve that these dead shall not have died in vain that this nation under god shall have a new birth of freedom and that government of the people by the people for the people shall not perish from the earth.
While we still made a few mistakes, the results are clearly good enough for many applications. Periphonics certainly thought so, for they licensed our program to incorporate into their products. Figure 6.11 shows that we were able to reconstruct correctly over 99% of the characters in a megabyte of President Clinton’s speeches, so if Bill had phoned them in, we would certainly be able to understand what he was saying. The reconstruction time is fast enough, indeed faster than you can type it in on the phone keypad.

The constraints for many pattern recognition problems can be naturally formulated as shortest path problems in graphs. In fact, there is a particularly convenient dynamic programming solution for these problems (the Viterbi algorithm) that is widely used in speech and handwriting recognition systems. Despite the fancy name, the Viterbi algorithm is basically solving a shortest path problem on a DAG. Hunting for a graph formulation to solve any given problem is often a good idea.

### 6.5 Network Flows and Bipartite Matching

Edge-weighted graphs can be interpreted as a network of pipes, where the weight of edge \((i, j)\) determines the capacity of the pipe. Capacities can be thought of as a function of the cross-sectional area of the pipe. A wide pipe might be able to carry 10 units of flow in a given time, whereas a narrower pipe might only carry 5 units. The network flow problem asks for the maximum amount of flow which can be sent from vertices \(s\) to \(t\) in a given weighted graph \(G\) while respecting the maximum capacities of each pipe.
6.5.1 Bipartite Matching

While the network flow problem is of independent interest, its primary importance is in solving other important graph problems. A classic example is bipartite matching. A matching in a graph $G = (V, E)$ is a subset of edges $E' \subset E$ such that no two edges of $E'$ share a vertex. A matching pairs off certain vertices such that every vertex is in, at most, one such pair, as shown in Figure 6.12.

Graph $G$ is bipartite or two-colorable if the vertices can be divided into two sets, $L$ and $R$, such that all edges in $G$ have one vertex in $L$ and one vertex in $R$. Many naturally defined graphs are bipartite. For example, certain vertices may represent jobs to be done and the remaining vertices represent people who can potentially do them. The existence of edge $(j, p)$ means that job $j$ can be done by person $p$. Or let certain vertices represent boys and certain vertices represent girls, with edges representing compatible pairs. Matchings in these graphs have natural interpretations as job assignments or as marriages, and are the focus of Section 15.6 (page 498).

The largest bipartite matching can be readily found using network flow. Create a source node $s$ that is connected to every vertex in $L$ by an edge of weight 1. Create a sink node $t$ and connect it to every vertex in $R$ by an edge of weight 1. Finally, assign each edge in the bipartite graph $G$ a weight of 1. Now, the maximum possible flow from $s$ to $t$ defines the largest matching in $G$. Certainly we can find a flow as large as the matching by using only the matching edges and their source-to-sink connections. Further, there can be no greater possible flow. How can we ever hope to get more than one flow unit through any vertex?

6.5.2 Computing Network Flows

Traditional network flow algorithms are based on the idea of augmenting paths, and repeatedly finding a path of positive capacity from $s$ to $t$ and adding it to the flow. It can be shown that the flow through a network is optimal if and only if it contains no augmenting path. Since each augmentation adds to the flow, we must eventually find the global maximum.

The key structure is the residual flow graph, denoted as $R(G, f)$, where $G$ is the input graph and $f$ is the current flow through $G$. This directed, edge-weighted $R(G, f)$ contains the same vertices as $G$. For each edge $(i, j)$ in $G$ with capacity $c(i, j)$ and flow $f(i, j)$, $R(G, f)$ may contain two edges:

(i) an edge $(i, j)$ with weight $c(i, j) - f(i, j)$, if $c(i, j) - f(i, j) > 0$ and

(ii) an edge $(j, i)$ with weight $f(i, j)$, if $f(i, j) > 0$.

The presence of edge $(i, j)$ in the residual graph indicates that positive flow can be pushed from $i$ to $j$. The weight of the edge gives the exact amount that can be pushed. A path in the residual flow graph from $s$ to $t$ implies that more flow can be pushed from $s$ to $t$ and the minimum edge weight on this path defines the amount of extra flow that can be pushed.
Figure 6.13: Maximum $s - t$ flow in a graph $G$ (on left) showing the associated residual graph $R(G)$ and minimum $s - t$ cut (dotted line near $t$)

Figure 6.13 illustrates this idea. The maximum $s - t$ flow in graph $G$ is 7. Such a flow is revealed by the two directed $t$ to $s$ paths in the residual graph $R(G)$ of capacities 2 + 5, respectively. These flows completely saturate the capacity of the two edges incident to vertex $t$, so no augmenting path remains. Thus the flow is optimal. A set of edges whose deletion separates $s$ from $t$ (like the two edges incident to $t$) is called an $s$-$t$ cut. Clearly, no $s$ to $t$ flow can exceed the weight of the minimum such cut. In fact, a flow equal to the minimum cut is always possible.

**Take-Home Lesson:** The maximum flow from $s$ to $t$ always equals the weight of the minimum $s$-$t$ cut. Thus, flow algorithms can be used to solve general edge and vertex connectivity problems in graphs.

**Implementation**

We cannot do full justice to the theory of network flows here. However, it is instructive to see how augmenting paths can be identified and the optimal flow computed.

For each edge in the residual flow graph, we must keep track of both the amount of flow currently going through the edge, as well as its remaining *residual* capacity. Thus, we must modify our `edge` structure to accommodate the extra fields:

```c
typedef struct {
    int v;          /* neighboring vertex */
    int capacity;   /* capacity of edge */
    int flow;       /* flow through edge */
    int residual;   /* residual capacity of edge */
    struct edgenode *next; /* next edge in list */
} edgenode;
```
We use a breadth-first search to look for any path from source to sink that increases the total flow, and use it to augment the total. We terminate with the optimal flow when no such augmenting path exists.

```c
netflow(flow_graph *g, int source, int sink) {
    int volume; /* weight of the augmenting path */
    add_residual_edges(g);
    initialize_search(g);
    bfs(g, source);
    volume = path_volume(g, source, sink, parent);
    while (volume > 0) {
        augment_path(g, source, sink, parent, volume);
        initialize_search(g);
        bfs(g, source);
        volume = path_volume(g, source, sink, parent);
    }
}
```

Any augmenting path from source to sink increases the flow, so we can use `bfs` to find such a path in the appropriate graph. We only consider network edges that have remaining capacity or, in other words, positive residual flow. The predicate below helps `bfs` distinguish between saturated and unsaturated edges:

```c
bool valid_edge(edgenode *e) {
    if (e->residual > 0) return (TRUE);
    else return (FALSE);
}
```

Augmenting a path transfers the maximum possible volume from the residual capacity into positive flow. This amount is limited by the path-edge with the smallest amount of residual capacity, just as the rate at which traffic can flow is limited by the most congested point.
int path_volume(flow_graph *g, int start, int end, int parents[]) {
    edgenode *e; /* edge in question */
    edgenode *find_edge();

    if (parents[end] == -1) return(0);

    e = find_edge(g, parents[end], end);

    if (start == parents[end])
        return(e->residual);
    else
        return( min(path_volume(g,start,parents[end],parents),
                     e->residual) );
}

edgenode *find_edge(flow_graph *g, int x, int y)
{
    edgenode *p; /* temporary pointer */

    p = g->edges[x];

    while (p != NULL) {
        if (p->v == y) return(p);
        p = p->next;
    }

    return(NULL);
}

Sending an additional unit of flow along directed edge \((i, j)\) reduces the residual capacity of edge \((i, j)\) but increases the residual capacity of edge \((j, i)\). Thus, the act of augmenting a path requires modifying both forward and reverse edges for each link on the path.
augment_path(flow_graph *g, int start, int end, int parents[], int volume) {
    edgenode *e; /* edge in question */
    edgenode *find_edge();

    if (start == end) return;

    e = find_edge(g,parents[end],end);
    e->flow += volume;
    e->residual -= volume;

    e = find_edge(g,end,parents[end]);
    e->residual += volume;

    augment_path(g,start,parents[end],parents,volume);
}

Initializing the flow graph requires creating directed flow edges \((i, j)\) and \((j, i)\) for each network edge \(e = (i, j)\). Initial flows are all set to 0. The initial residual flow of \((i, j)\) is set to the capacity of \(e\), while the initial residual flow of \((j, i)\) is set to 0.

Analysis

The augmenting path algorithm above eventually converges on the optimal solution. However, each augmenting path may add only a little to the total flow, so, in principle, the algorithm might take an arbitrarily long time to converge.

However, Edmonds and Karp [EK72] proved that always selecting a shortest unweighted augmenting path guarantees that \(O(n^3)\) augmentations suffice for optimization. In fact, the Edmonds-Karp algorithm is what is implemented above, since a breadth-first search from the source is used to find the next augmenting path.

6.6 Design Graphs, Not Algorithms

Proper modeling is the key to making effective use of graph algorithms. We have defined several graph properties, and developed algorithms for computing them. All told, about two dozen different graph problems are presented in the catalog, mostly in Sections 15 and 16. These classical graph problems provide a framework for modeling most applications.

The secret is learning to design graphs, not algorithms. We have already seen a few instances of this idea:
• The maximum spanning tree can be found by negating the edge weights of the input graph $G$ and using a minimum spanning tree algorithm on the result. The most negative weight spanning tree will define the maximum weight tree in $G$.

• To solve bipartite matching, we constructed a special network flow graph such that the maximum flow corresponds to a maximum cardinality matching.

The applications below demonstrate the power of proper modeling. Each arose in a real-world application, and each can be modeled as a graph problem. Some of the modelings are quite clever, but they illustrate the versatility of graphs in representing relationships. As you read a problem, try to devise an appropriate graph representation before peeking to see how we did it.

**Stop and Think: The Pink Panther’s Passport to Peril**

*Problem:* “I’m looking for an algorithm to design natural routes for video-game characters to follow through an obstacle-filled room. How should I do it?”

*Solution:* Presumably the desired route should look like a path that an intelligent being would choose. Since intelligent beings are either lazy or efficient, this should be modeled as a shortest path problem.

But what is the graph? One approach might be to lay a grid of points in the room. Create a vertex for each grid point that is a valid place for the character to stand; i.e., that does not lie within an obstacle. There will be an edge between any pair of nearby vertices, weighted proportionally to the distance between them. Although direct geometric methods are known for shortest paths (see Section 15.4 (page 489)), it is easier to model this discretely as a graph.

**Stop and Think: Ordering the Sequence**

*Problem:* “A DNA sequencing project generates experimental data consisting of small fragments. For each given fragment $f$, we know certain other fragments are forced to lie to the left of $f$, and certain other fragments are forced to be to the right of $f$. How can we find a consistent ordering of the fragments from left to right that satisfies all the constraints?”

*Solution:* Create a directed graph, where each fragment is assigned a unique vertex. Insert a directed edge $(l, f)$ from any fragment $l$ that is forced to be to the left of $f$. 

---
of $f$, and a directed edge $(f, r)$ to any fragment $r$ forced to be to the right of $f$. We seek an ordering of the vertices such that all the edges go from left to right. This is a topological sort of the resulting directed acyclic graph. The graph must be acyclic, because cycles make finding a consistent ordering impossible.

**Stop and Think: Bucketing Rectangles**

*Problem:* “In my graphics work I need to solve the following problem. Given an arbitrary set of rectangles in the plane, how can I distribute them into a minimum number of buckets such that no subset of rectangles in any given bucket intersects another? In other words, there can not be any overlapping area between two rectangles in the same bucket.”

*Solution:* We formulate a graph where each vertex is a rectangle, and there is an edge if two rectangles intersect. Each bucket corresponds to an independent set of rectangles, so there is no overlap between any two. A vertex coloring of a graph is a partition of the vertices into independent sets, so minimizing the number of colors is exactly what you want.

**Stop and Think: Names in Collision**

*Problem:* “In porting code from UNIX to DOS, I have to shorten several hundred file names down to at most 8 characters each. I can’t just use the first eight characters from each name, because “filename1” and “filename2” would be assigned the exact same name. How can I meaningfully shorten the names while ensuring that they do not collide?”

*Solution:* Construct a bipartite graph with vertices corresponding to each original file name $f_i$ for $1 \leq i \leq n$, as well as a collection of acceptable shortenings for each name $f_{i1}, \ldots, f_{ik}$. Add an edge between each original and shortened name. We now seek a set of $n$ edges that have no vertices in common, so each file name is mapped to a distinct acceptable substitute. Bipartite matching, discussed in Section 15.6 (page 498), is exactly this problem of finding an independent set of edges in a graph.
Stop and Think: Separate the Text

Problem: "We need a way to separate the lines of text in the optical character-recognition system that we are building. Although there is some white space between the lines, problems like noise and the tilt of the page makes it hard to find. How can we do line segmentation?

Solution: Consider the following graph formulation. Treat each pixel in the image as a vertex in the graph, with an edge between two neighboring pixels. The weight of this edge should be proportional to how dark the pixels are. A segmentation between two lines is a path in this graph from the left to right side of the page. We seek a relatively straight path that avoids as much blackness as possible. This suggests that the shortest path in the pixel graph will likely find a good line segmentation.

Take-Home Lesson: Designing novel graph algorithms is very hard, so don’t do it. Instead, try to design graphs that enable you to use classical algorithms to model your problem.

Chapter Notes

Network flows are an advanced algorithmic technique, and recognizing whether a particular problem can be solved by network flow requires experience. We point the reader to books by Cook and Cunningham [CC97] and Ahuja, Magnanti, and Orlin [AMO93] for more detailed treatments of the subject.

The augmenting path method for network flows is due to Ford and Fulkerson [FF62]. Edmonds and Karp [EK72] proved that always selecting a shortest geodesic augmenting path guarantees that $O(n^3)$ augmentations suffice for optimization.

The phone code reconstruction system that was the subject of the war story is described in more technical detail in [RS96].

6.7 Exercises

Simulating Graph Algorithms

6-1. [3] For the graphs in Problem 5-1:

(a) Draw the spanning forest after every iteration of the main loop in Kruskal’s algorithm.

(b) Draw the spanning forest after every iteration of the main loop in Prim’s algorithm.
(c) Find the shortest path spanning tree rooted in $A$.

(d) Compute the maximum flow from $A$ to $H$.

Minimum Spanning Trees

6-2. [3] Is the path between two vertices in a minimum spanning tree necessarily a shortest path between the two vertices in the full graph? Give a proof or a counterexample.

6-3. [3] Assume that all edges in the graph have distinct edge weights (i.e., no pair of edges have the same weight). Is the path between a pair of vertices in a minimum spanning tree necessarily a shortest path between the two vertices in the full graph? Give a proof or a counterexample.


6-5. [3] Does either Prim’s and Kruskal’s algorithm work if there are negative edge weights? Explain why or why not.

6-6. [5] Suppose we are given the minimum spanning tree $T$ of a given graph $G$ (with $n$ vertices and $m$ edges) and a new edge $e = (u, v)$ of weight $w$ that we will add to $G$. Give an efficient algorithm to find the minimum spanning tree of the graph $G + e$. Your algorithm should run in $O(n)$ time to receive full credit.

6-7. [5] (a) Let $T$ be a minimum spanning tree of a weighted graph $G$. Construct a new graph $G'$ by adding a weight of $k$ to every edge of $G$. Do the edges of $T$ form a minimum spanning tree of $G'$? Prove the statement or give a counterexample.

(b) Let $P = \{s, \ldots, t\}$ describe a shortest weighted path between vertices $s$ and $t$ of a weighted graph $G$. Construct a new graph $G'$ by adding a weight of $k$ to every edge of $G$. Does $P$ describe a shortest path from $s$ to $t$ in $G'$? Prove the statement or give a counterexample.

6-8. [5] Devise and analyze an algorithm that takes a weighted graph $G$ and finds the smallest change in the cost to a non-MST edge that would cause a change in the minimum spanning tree of $G$. Your algorithm must be correct and run in polynomial time.

6-9. [4] Consider the problem of finding a minimum weight connected subset $T$ of edges from a weighted connected graph $G$. The weight of $T$ is the sum of all the edge weights in $T$.

(a) Why is this problem not just the minimum spanning tree problem? Hint: think negative weight edges.

(b) Give an efficient algorithm to compute the minimum weight connected subset $T$.

6-10. [4] Let $G = (V, E)$ be an undirected graph. A set $F \subseteq E$ of edges is called a feedback-edge set if every cycle of $G$ has at least one edge in $F$.

(a) Suppose that $G$ is unweighted. Design an efficient algorithm to find a minimum-size feedback-edge set.
(b) Suppose that $G$ is a weighted undirected graph with positive edge weights. Design an efficient algorithm to find a minimum-weight feedback-edge set.

6-11. [5] Modify Prim’s algorithm so that it runs in time $O(n \log k)$ on a graph that has only $k$ different edges costs.

Union-Find

6-12. [5] Devise an efficient data structure to handle the following operations on a weighted directed graph:

(a) Merge two given components.
(b) Locate which component contains a given vertex $v$.
(c) Retrieve a minimum edge from a given component.

6-13. [5] Design a data structure that can perform a sequence of, $m$ union and find operations on a universal set of $n$ elements, consisting of a sequence of all unions followed by a sequence of all finds, in time $O(m + n)$.

Shortest Paths

6-14. [3] The single-destination shortest path problem for a directed graph seeks the shortest path from every vertex to a specified vertex $v$. Give an efficient algorithm to solve the single-destination shortest paths problem.

6-15. [3] Let $G = (V, E)$ be an undirected weighted graph, and let $T$ be the shortest-path spanning tree rooted at a vertex $v$. Suppose now that all the edge weights in $G$ are increased by a constant number $k$. Is $T$ still the shortest-path spanning tree from $v$?

6-16. [3] Answer all of the following:

(a) Give an example of a weighted connected graph $G = (V, E)$ and a vertex $v$, such that the minimum spanning tree of $G$ is the same as the shortest-path spanning tree rooted at $v$.

(b) Give an example of a weighted connected directed graph $G = (V, E)$ and a vertex $v$, such that the minimum-cost spanning tree of $G$ is very different from the shortest-path spanning tree rooted at $v$.

(c) Can the two trees be completely disjointed?

6-17. [3] Either prove the following or give a counterexample:

(a) Is the path between a pair of vertices in a minimum spanning tree of an undirected graph necessarily the shortest (minimum weight) path?

(b) Suppose that the minimum spanning tree of the graph is unique. Is the path between a pair of vertices in a minimum spanning tree of an undirected graph necessarily the shortest (minimum weight) path?

6-18. [5] In certain graph problems, vertices have can have weights instead of or in addition to the weights of edges. Let $C_v$ be the cost of vertex $v$, and $C_{(x,y)}$ the cost of the edge $(x, y)$. This problem is concerned with finding the cheapest path between vertices $a$ and $b$ in a graph $G$. The cost of a path is the sum of the costs of the edges and vertices encountered on the path.
(a) Suppose that each edge in the graph has a weight of zero (while non-edges have a cost of $\infty$). Assume that $C_v = 1$ for all vertices $1 \leq v \leq n$ (i.e., all vertices have the same cost). Give an efficient algorithm to find the cheapest path from $a$ to $b$ and its time complexity.

(b) Now suppose that the vertex costs are not constant (but are all positive) and the edge costs remain as above. Give an efficient algorithm to find the cheapest path from $a$ to $b$ and its time complexity.

(c) Now suppose that both the edge and vertex costs are not constant (but are all positive). Give an efficient algorithm to find the cheapest path from $a$ to $b$ and its time complexity.

6-19. [5] Let $G$ be a weighted directed graph with $n$ vertices and $m$ edges, where all edges have positive weight. A directed cycle is a directed path that starts and ends at the same vertex and contains at least one edge. Give an $O(n^3)$ algorithm to find a directed cycle in $G$ of minimum total weight. Partial credit will be given for an $O(n^2m)$ algorithm.

6-20. [5] Can we modify Dijkstra’s algorithm to solve the single-source longest path problem by changing minimum to maximum? If so, then prove your algorithm correct. If not, then provide a counterexample.

6-21. [5] Let $G = (V, E)$ be a weighted acyclic directed graph with possibly negative edge weights. Design a linear-time algorithm to solve the single-source shortest-path problem from a given source $v$.

6-22. [5] Let $G = (V, E)$ be a directed weighted graph such that all the weights are positive. Let $v$ and $w$ be two vertices in $G$ and $k \leq |V|$ be an integer. Design an algorithm to find the shortest path from $v$ to $w$ that contains exactly $k$ edges. Note that the path need not be simple.

6-23. [5] Arbitrage is the use of discrepancies in currency-exchange rates to make a profit. For example, there may be a small window of time during which 1 U.S. dollar buys 0.75 British pounds, 1 British pound buys 2 Australian dollars, and 1 Australian dollar buys 0.70 U.S. dollars. At such a time, a smart trader can trade one U.S. dollar and end up with $0.75 \times 2 \times 0.7 = 1.05$ U.S. dollars—a profit of 5%. Suppose that there are $n$ currencies $c_1, \ldots, c_n$ and an $n \times n$ table $R$ of exchange rates, such that one unit of currency $c_i$ buys $R[i, j]$ units of currency $c_j$. Devise and analyze an algorithm to determine the maximum value of

$$R[c_1, c_{i_1}] \cdot R[c_{i_1}, c_{i_2}] \cdots R[c_{i_{k-1}}, c_{i_k}] \cdot R[c_{i_k}, c_1]$$

Hint: think all-pairs shortest path.

Network Flow and Matching

6-24. [3] A matching in a graph is a set of disjoint edges—i.e., edges that do not share any vertices in common. Give a linear-time algorithm to find a maximum matching in a tree.

6-25. [5] An edge cover of an undirected graph $G = (V, E)$ is a set of edges such that each vertex in the graph is incident to at least one edge from the set. Give an efficient algorithm, based on matching, to find the minimum-size edge cover for $G$. 
Programming Challenges

These programming challenge problems with robot judging are available at http://www.programming-challenges.com or http://online-judge.uva.es.

6-1. “Freckles” – Programming Challenges 111001, UVA Judge 10034.
6-2. “Necklace” – Programming Challenges 111002, UVA Judge 10054.
6-3. “Railroads” – Programming Challenges 111004, UVA Judge 10039.