However, Strassen [Str69] discovered a divide-and-conquer algorithm that manipulates the products of seven $n/2 \times n/2$ matrix products to yield the product of two $n \times n$ matrices. This yields a time-complexity recurrence $T(n) = 7T(n/2) + O(n^2)$. In fact, this recurrence evaluates to $T(n) = O(n^{2.81})$, which seems impossible to predict *without* solving the recurrence.

4.10.3 Solving Divide-and-Conquer Recurrences (*)

In fact, divide-and-conquer recurrences of the form $T(n) = aT(n/b) + f(n)$ are generally easy to solve, because the solutions typically fall into one of three distinct cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$.

Although this looks somewhat frightening, it really isn't difficult to apply. The issue is identifying which case of the so-called *master theorem* holds for your given recurrence. Case 1 holds for heap construction and matrix multiplication, while Case 2 holds mergesort and binary search. Case 3 generally arises for clumsier algorithms, where the cost of combining the subproblems dominates everything.

The master theorem can be thought of as a black-box piece of machinery, invoked as needed and left with its mystery intact. However, with a little study, the reason why the master theorem works can become apparent.

Figure 4.9 shows the recursion tree associated with a typical $T(n) = aT(n/b) + f(n)$ divide-and-conquer algorithm. Each problem of size n is decomposed into a problems of size n/b. Each subproblem of size k takes $O(f(k))$ time to deal with internally, between partitioning and merging. The total time for the algorithm is the sum of these internal costs, plus the overhead of building the recursion tree. The height of this tree is $h = \log_b n$ and the number of leaf nodes $a^h = a^{\log_b n}$, which happens to simplify to $n^{\log_b a}$ with some algebraic manipulation.

The three cases of the master theorem correspond to three different costs which might be dominant as a function of a, b, and $f(n)$:

- *Case 1: Too many leaves* – If the number of leaf nodes outweighs the sum of the internal evaluation cost, the total running time is $O(n^{\log_b a})$.
- *Case 2: Equal work per level* – As we move down the tree, each problem gets smaller but there are more of them to solve. If the sum of the internal evaluation costs at each level are equal, the total running time is the cost per level ($n^{\log_b a}$) times the number of levels ($\log_b n$), for a total running time of $O(n^{\log_b a} \lg n)$.

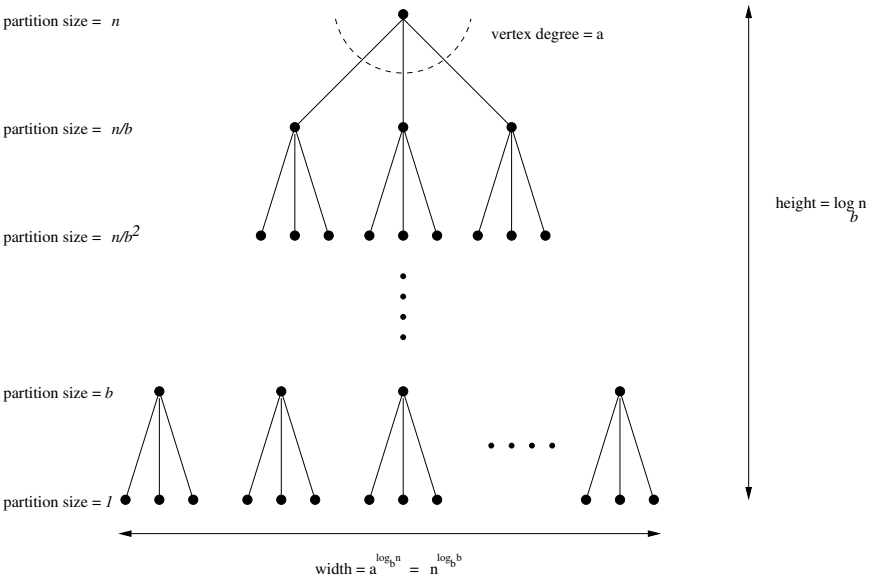

Figure 4.9: The recursion tree resulting from decomposing each problem of size n into a problems of size n/b

- *Case 3: Too expensive a root* – If the internal evaluation costs grow rapidly enough with n, then the cost of the root evaluation may dominate. If so, the the total running time is $O(f(n))$.

Chapter Notes

The most interesting sorting algorithms that have not been discussed in this section include *shellsort*, which is a substantially more efficient version of insertion sort, and *radix sort*, an efficient algorithm for sorting strings. You can learn more about these and every other sorting algorithm by browsing through Knuth [Knu98], with hundreds of pages of interesting material on sorting. This includes external sorting, the subject of this chapter's legal war story.

As implemented here, mergesort copies the merged elements into an auxiliary buffer to avoid overwriting the original elements to be sorted. Through clever but complicated buffer manipulation, mergesort can be implemented in an array without using much extra storage. Kronrod's algorithm for in-place merging is presented in [Knu98].

Randomized algorithms are discussed in greater detail in the books by Motwani and Raghavan [MR95] and Mitzenmacher and Upfal [MU05]. The problem of

nut and bolt sorting was introduced by [Raw92]. A complicated but deterministic $O(n \log n)$ algorithm is due to Komlos, Ma, and Szemeredi [KMS96].

Several other algorithms texts provide more substantive coverage of divide-and-conquer algorithms, including [CLRS01, KT06, Man89]. See [CLRS01] for an excellent overview of the master theorem.

4.11 Exercises

Applications of Sorting

4-1. *[3]* The Grinch is given the job of partitioning $2n$ players into two teams of n players each. Each player has a numerical rating that measures how good he/she is at the game. He seeks to divide the players as *unfairly* as possible, so as to create the biggest possible talent imbalance between team A and team B. Show how the Grinch can do the job in $O(n \log n)$ time.

4-2. *[3]* For each of the following problems, give an algorithm that finds the desired numbers within the given amount of time. To keep your answers brief, feel free to use algorithms from the book as subroutines. For the example, $S = \{6, 13, 19, 3, 8\}$, $19 - 3$ maximizes the difference, while $8 - 6$ minimizes the difference.

(a) Let S be an *unsorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *maximizes* $|x - y|$. Your algorithm must run in $O(n)$ worst-case time.

(b) Let S be a *sorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *maximizes* $|x - y|$. Your algorithm must run in $O(1)$ worst-case time.

(c) Let S be an *unsorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *minimizes* $|x - y|$, for $x \neq y$. Your algorithm must run in $O(n \log n)$ worst-case time.

(d) Let S be a *sorted* array of n integers. Give an algorithm that finds the pair $x, y \in S$ that *minimizes* $|x - y|$, for $x \neq y$. Your algorithm must run in $O(n)$ worst-case time.

4-3. *[3]* Take a sequence of $2n$ real numbers as input. Design an $O(n \log n)$ algorithm that partitions the numbers into n pairs, with the property that the partition minimizes the maximum sum of a pair. For example, say we are given the numbers (1,3,5,9). The possible partitions are ((1,3),(5,9)), ((1,5),(3,9)), and ((1,9),(3,5)). The pair sums for these partitions are (4,14), (6,12), and (10,8). Thus the third partition has 10 as its maximum sum, which is the minimum over the three partitions.

4-4. *[3]* Assume that we are given n pairs of items as input, where the first item is a number and the second item is one of three colors (red, blue, or yellow). Further assume that the items are sorted by number. Give an $O(n)$ algorithm to sort the items by color (all reds before all blues before all yellows) such that the numbers for identical colors stay sorted.

For example: (1,blue), (3,red), (4,blue), (6,yellow), (9,red) should become (3,red), (9,red), (1,blue), (4,blue), (6,yellow).

4-5. *[3]* The *mode* of a set of numbers is the number that occurs most frequently in the set. The set $(4, 6, 2, 4, 3, 1)$ has a mode of 4. Give an efficient and correct algorithm to compute the mode of a set of n numbers.

4-6. *[3]* Given two sets S_1 and S_2 (each of size n), and a number x, describe an $O(n \log n)$ algorithm for finding whether there exists a pair of elements, one from S_1 and one from S_2, that add up to x. (For partial credit, give a $\Theta(n^2)$ algorithm for this problem.)

4-7. *[3]* Outline a reasonable method of solving each of the following problems. Give the order of the worst-case complexity of your methods.

(a) You are given a pile of thousands of telephone bills and thousands of checks sent in to pay the bills. Find out who did not pay.

(b) You are given a list containing the title, author, call number and publisher of all the books in a school library and another list of 30 publishers. Find out how many of the books in the library were published by each company.

(c) You are given all the book checkout cards used in the campus library during the past year, each of which contains the name of the person who took out the book. Determine how many distinct people checked out at least one book.

4-8. *[4]* Given a set of S containing n real numbers, and a real number x. We seek an algorithm to determine whether two elements of S exist whose sum is exactly x.

(a) Assume that S is unsorted. Give an $O(n \log n)$ algorithm for the problem.

(b) Assume that S is sorted. Give an $O(n)$ algorithm for the problem.

4-9. *[4]* Give an efficient algorithm to compute the union of sets A and B, where $n = \max(|A|, |B|)$. The output should be an array of distinct elements that form the union of the sets, such that they appear more than once in the union.

(a) Assume that A and B are unsorted. Give an $O(n \log n)$ algorithm for the problem.

(b) Assume that A and B are sorted. Give an $O(n)$ algorithm for the problem.

4-10. *[5]* Given a set S of n integers and an integer T, give an $O(n^{k-1} \log n)$ algorithm to test whether k of the integers in S add up to T.

4-11. *[6]* Design an $O(n)$ algorithm that, given a list of n elements, finds all the elements that appear more than $n/2$ times in the list. *Then*, design an $O(n)$ algorithm that, given a list of n elements, finds all the elements that appear more than $n/4$ times.

Heaps

4-12. *[3]* Devise an algorithm for finding the k smallest elements of an unsorted set of n integers in $O(n + k \log n)$.

4-13. *[5]* You wish to store a set of n numbers in either a max-heap or a sorted array. For each application below, state which data structure is better, or if it does not matter. Explain your answers.

(a) Want to find the maximum element quickly.

(b) Want to be able to delete an element quickly.

(c) Want to be able to form the structure quickly.

(d) Want to find the minimum element quickly.

4-14. *[5]* Give an $O(n \log k)$-time algorithm that merges k sorted lists with a total of n elements into one sorted list. (Hint: use a heap to speed up the elementary $O(kn)$-time algorithm).

4-15. *[5]* (a) Give an efficient algorithm to find the second-largest key among n keys. You can do better than $2n - 3$ comparisons.

(b) Then, give an efficient algorithm to find the third-largest key among n keys. How many key comparisons does your algorithm do in the worst case? Must your algorithm determine which key is largest and second-largest in the process?

Quicksort

4-16. *[3]* Use the partitioning idea of quicksort to give an algorithm that finds the *median* element of an array of n integers in expected $O(n)$ time. (Hint: must you look at both sides of the partition?)

4-17. *[3]* The *median* of a set of n values is the $\lceil n/2 \rceil$th smallest value.

(a) Suppose quicksort always pivoted on the median of the current sub-array. How many comparisons would Quicksort make then in the worst case?

(b) Suppose quicksort were always to pivot on the $\lceil n/3 \rceil$th smallest value of the current sub-array. How many comparisons would be made then in the worst case?

4-18. *[5]* Suppose an array A consists of n elements, each of which is *red*, *white*, or *blue*. We seek to sort the elements so that all the *reds* come before all the *whites*, which come before all the *blues* The only operation permitted on the keys are

- *Examine(A,i)* – report the color of the ith element of A.
- *Swap(A,i,j)* – swap the ith element of A with the jth element.

Find a correct and efficient algorithm for red-white-blue sorting. There is a linear-time solution.

4-19. *[5]* An *inversion* of a permutation is a pair of elements that are out of order.

(a) Show that a permutation of n items has at most $n(n-1)/2$ inversions. Which permutation(s) have exactly $n(n-1)/2$ inversions?

(b) Let P be a permutation and P^r be the reversal of this permutation. Show that P and P^r have a total of exactly $n(n-1)/2$ inversions.

(c) Use the previous result to argue that the expected number of inversions in a random permutation is $n(n-1)/4$.

4-20. *[3]* Give an efficient algorithm to rearrange an array of n keys so that all the negative keys precede all the nonnegative keys. Your algorithm must be in-place, meaning you cannot allocate another array to temporarily hold the items. How fast is your algorithm?

Other Sorting Algorithms

4-21. *[5]* Stable sorting algorithms leave equal-key items in the same relative order as in the original permutation. Explain what must be done to ensure that mergesort is a stable sorting algorithm.

4-22. *[3]* Show that n positive integers in the range 1 to k can be sorted in $O(n \log k)$ time. The interesting case is when $k << n$.

4-23. *[5]* We seek to sort a sequence S of n integers with many duplications, such that the number of distinct integers in S is $O(\log n)$. Give an $O(n \log \log n)$ worst-case time algorithm to sort such sequences.

4-24. *[5]* Let $A[1..n]$ be an array such that the first $n - \sqrt{n}$ elements are already sorted (though we know nothing about the remaining elements). Give an algorithm that sorts A in substantially better than $n \log n$ steps.

4-25. *[5]* Assume that the array $A[1..n]$ only has numbers from $\{1, \ldots, n^2\}$ but that at most $\log \log n$ of these numbers ever appear. Devise an algorithm that sorts A in substantially less than $O(n \log n)$.

4-26. *[5]* Consider the problem of sorting a sequence of n 0's and 1's using comparisons. For each comparison of two values x and y, the algorithm learns which of $x < y$, $x = y$, or $x > y$ holds.

(a) Give an algorithm to sort in $n - 1$ comparisons in the worst case. Show that your algorithm is optimal.

(b) Give an algorithm to sort in $2n/3$ comparisons in the average case (assuming each of the n inputs is 0 or 1 with equal probability). Show that your algorithm is optimal.

4-27. *[6]* Let P be a simple, but not necessarily convex, polygon and q an arbitrary point not necessarily in P. Design an efficient algorithm to find a line segment originating from q that intersects the maximum number of edges of P. In other words, if standing at point q, in what direction should you aim a gun so the bullet will go through the largest number of walls. A bullet through a vertex of P gets credit for only one wall. An $O(n \log n)$ algorithm is possible.

Lower Bounds

4-28. *[5]* In one of my research papers [Ski88], I discovered a comparison-based sorting algorithm that runs in $O(n \log(\sqrt{n}))$. Given the existence of an $\Omega(n \log n)$ lower bound for sorting, how can this be possible?

4-29. *[5]* Mr. B. C. Dull claims to have developed a new data structure for priority queues that supports the operations *Insert*, *Maximum*, and *Extract-Max*—all in $O(1)$ worst-case time. Prove that he is mistaken. (Hint: the argument does not involve a lot of gory details—just think about what this would imply about the $\Omega(n \log n)$ lower bound for sorting.)

Searching

4-30. *[3]* A company database consists of 10,000 sorted names, 40% of whom are known as good customers and who together account for 60% of the accesses to the database. There are two data structure options to consider for representing the database:

- Put all the names in a single array and use binary search.
- Put the good customers in one array and the rest of them in a second array. Only if we do not find the query name on a binary search of the first array do we do a binary search of the second array.

Demonstrate which option gives better expected performance. Does this change if linear search on an unsorted array is used instead of binary search for both options?

4-31. *[3]* Suppose you are given an array A of n sorted numbers that has been *circularly shifted* k positions to the right. For example, $\{35, 42, 5, 15, 27, 29\}$ is a sorted array that has been circularly shifted $k = 2$ positions, while $\{27, 29, 35, 42, 5, 15\}$ has been shifted $k = 4$ positions.

- Suppose you know what k is. Give an $O(1)$ algorithm to find the largest number in A.
- Suppose you *do not* know what k is. Give an $O(\lg n)$ algorithm to find the largest number in A. For partial credit, you may give an $O(n)$ algorithm.

4-32. *[3]* Consider the numerical 20 Questions game. In this game, Player 1 thinks of a number in the range 1 to n. Player 2 has to figure out this number by asking the fewest number of true/false questions. Assume that nobody cheats.

(a) What is an optimal strategy if n in known?

(b) What is a good strategy is n is not known?

4-33. *[5]* Suppose that you are given a sorted sequence of *distinct* integers $\{a_1, a_2, \ldots, a_n\}$. Give an $O(\lg n)$ algorithm to determine whether there exists an i index such as $a_i = i$. For example, in $\{-10, -3, 3, 5, 7\}$, $a_3 = 3$. In $\{2, 3, 4, 5, 6, 7\}$, there is no such i.

4-34. *[5]* Suppose that you are given a sorted sequence of *distinct* integers $\{a_1, a_2, \ldots, a_n\}$, drawn from 1 to m where $n < m$. Give an $O(\lg n)$ algorithm to find an integer $\leq m$ that is not present in a. For full credit, find the smallest such integer.

4-35. *[5]* Let M be an $n \times m$ integer matrix in which the entries of each row are sorted in increasing order (from left to right) and the entries in each column are in increasing order (from top to bottom). Give an efficient algorithm to find the position of an integer x in M, or to determine that x is not there. How many comparisons of x with matrix entries does your algorithm use in worst case?

Implementation Challenges

4-36. *[5]* Consider an $n \times n$ array A containing integer elements (positive, negative, and zero). Assume that the elements in each row of A are in strictly increasing order, and the elements of each column of A are in strictly decreasing order. (Hence there cannot be two zeroes in the same row or the same column.) Describe an efficient algorithm that counts the number of occurrences of the element 0 in A. Analyze its running time.

4-37. *[6]* Implement versions of several different sorting algorithms, such as selection sort, insertion sort, heapsort, mergesort, and quicksort. Conduct experiments to assess the relative performance of these algorithms in a simple application that reads a large text file and reports exactly one instance of each word that appears within it. This application can be efficiently implemented by sorting all the words that occur in the text and then passing through the sorted sequence to identify one instance of each distinct word. Write a brief report with your conclusions.

4-38. *[5]* Implement an external sort, which uses intermediate files to sort files bigger than main memory. Mergesort is a good algorithm to base such an implementation on. Test your program both on files with small records and on files with large records.

4-39. *[8]* Design and implement a parallel sorting algorithm that distributes data across several processors. An appropriate variation of mergesort is a likely candidate. Measure the speedup of this algorithm as the number of processors increases. Later, compare the execution time to that of a purely sequential mergesort implementation. What are your experiences?

Interview Problems

4-40. *[3]* If you are given a million integers to sort, what algorithm would you use to sort them? How much time and memory would that consume?

4-41. *[3]* Describe advantages and disadvantages of the most popular sorting algorithms.

4-42. *[3]* Implement an algorithm that takes an input array and returns only the unique elements in it.

4-43. *[5]* You have a computer with only 2Mb of main memory. How do you use it to sort a large file of 500 Mb that is on disk?

4-44. *[5]* Design a stack that supports push, pop, and retrieving the minimum element in constant time. Can you do this?

4-45. *[5]* Given a search string of three words, find the smallest snippet of the document that contains all three of the search words—i.e. , the snippet with smallest number of words in it. You are given the index positions where these words in occur search strings, such as *word1: (1, 4, 5)*, *word2: (4, 9, 10)*, and *word3: (5, 6, 15)*. Each of the lists are in sorted order, as above.

4-46. *[6]* You are given 12 coins. One of them is heavier or lighter than the rest. Identify this coin in just three weighings.

Programming Challenges

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

4-1. "Vito's Family" – Programming Challenges 110401, UVA Judge 10041.

4-2. "Stacks of Flapjacks" – Programming Challenges 110402, UVA Judge 120.

4-3. "Bridge" – Programming Challenges 110403, UVA Judge 10037.

4-4. "ShoeMaker's Problem" – Programming Challenges 110405, UVA Judge 10026.

4-5. "ShellSort" – Programming Challenges 110407, UVA Judge 10152.

5

Graph Traversal

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.

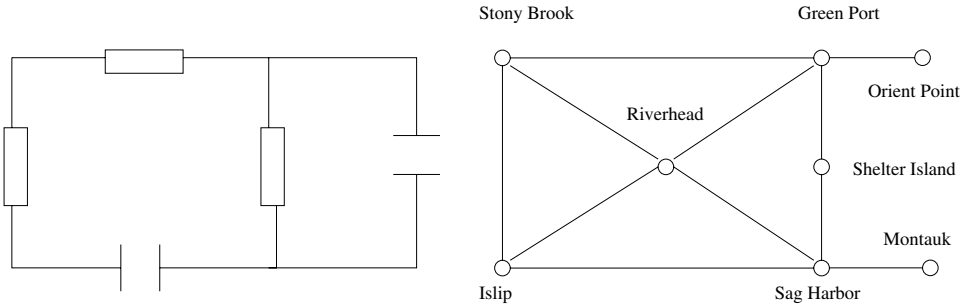

Figure 5.1: Modeling road networks and electronic circuits as graphs

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_5,

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

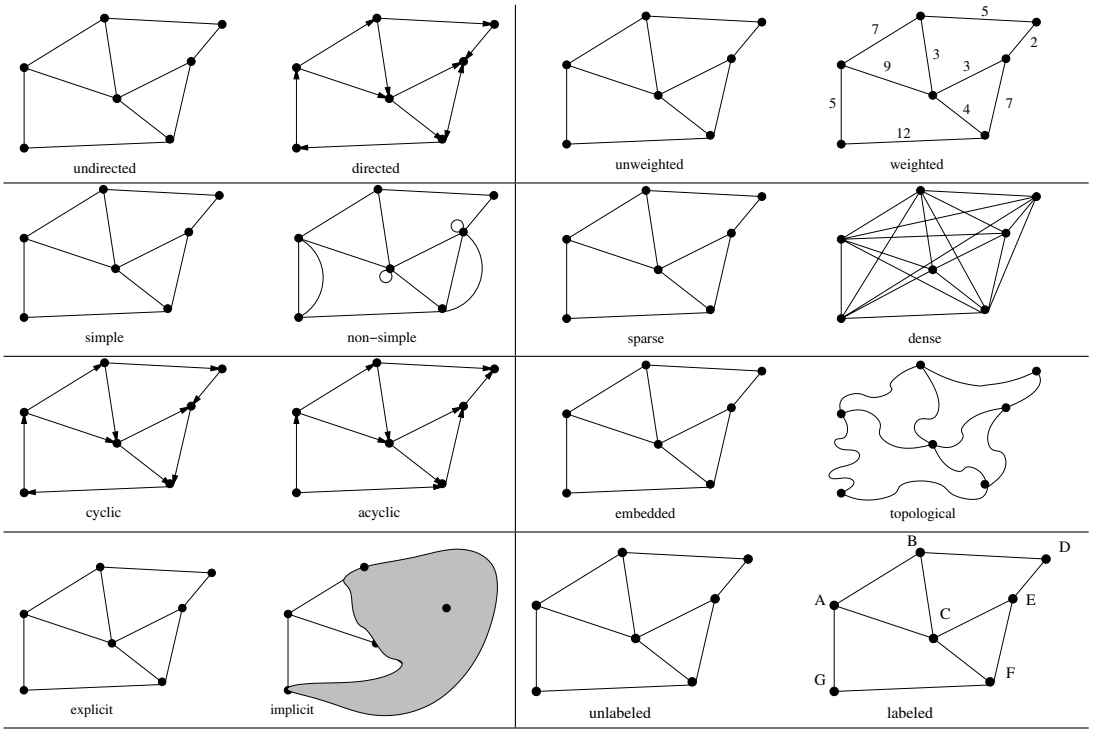

Figure 5.2: Important properties / flavors of graphs

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*.

- *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.

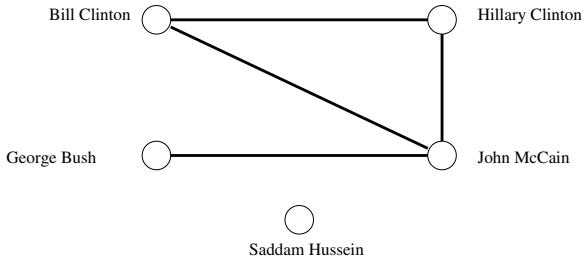

Figure 5.3: A portion of the friendship graph

- *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 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 → Bob McGrath → John Marberger → George W. Bush), but there could

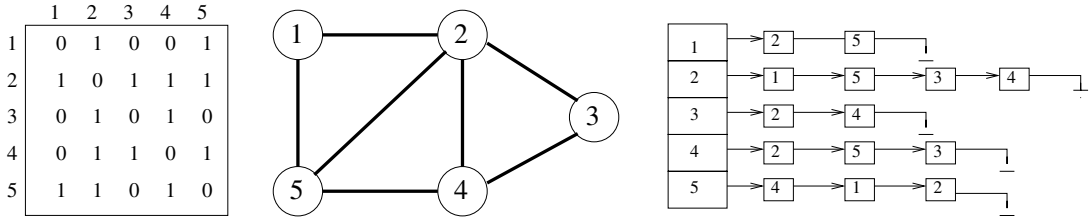

Figure 5.4: The adjacency matrix and adjacency list of a given graph

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 identify, 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 performance. Your two basic choices are adjacency matrices and adjacency lists, illustrated 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.

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

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 `nvertices`. We represent the edges using an array of linked lists:

```
#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;
```

We represent directed edge (x, y) by an `edgenode` y in x's adjacency list. The degree field of the `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 `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.

```
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;
}
```

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:

```
print_graph(graph *g)
{
        int i;                          /* counter */
        edgenode *p;                    /* temporary pointer */

        for (i=1; i<=g->nvertices; i++) {
                printf("%d: ",i);
                p = g->edges[i];
                while (p != NULL) {
                        printf(" %d",p->y);
                        p = p->next;
                }
                printf("\n");
        }
}
```

It is a good idea to use a well-designed graph data type as a model for building your own, or even better as the foundation for your application. We recommend LEDA (see Section 19.1.1 (page 658)) or Boost (see Section 19.1.3 (page 659)) as the best-designed general-purpose graph data structures currently available. They may be more powerful (and hence somewhat slower/larger) than you need, but they do so many things right that you are likely to lose most of the potential do-it-yourself benefits through clumsiness.

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 slow downs 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

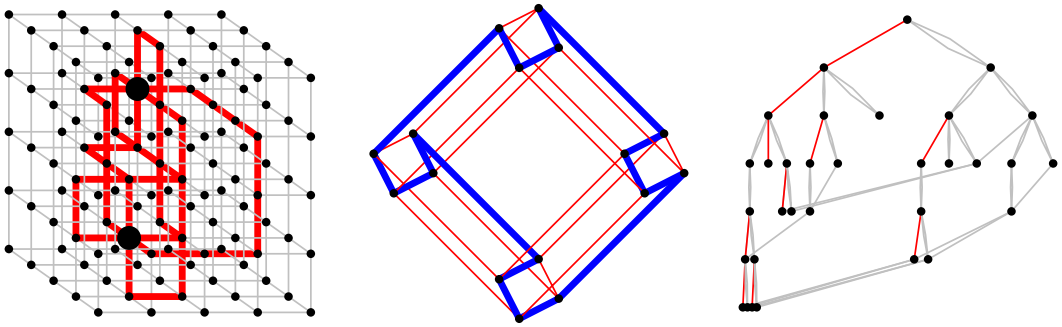

Figure 5.6: Representative *Combinatorica* graphs: edge-disjoint paths (left), Hamiltonian cycle in a hypercube (center), animated depth-first search tree traversal (right)

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 9×9 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?

Approximate year command/machine	1990 Sun-3	1991 Sun-4	1998 Sun-5	2000 Ultra 5	2004 SunBlade
PlanarQ[GridGraph[4,4]]	234.10	69.65	27.50	3.60	0.40
Length[Partitions[30]]	289.85	73.20	24.40	3.44	1.58
VertexConnectivity[GridGraph[3,3]]	239.67	47.76	14.70	2.00	0.91
RandomPartition[1000]	831.68	267.5	22.05	3.12	0.87

Table 5.1: Old *Combinatorica* benchmarks on low-end Sun workstations, from 1990 to today, (running time in seconds)

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.

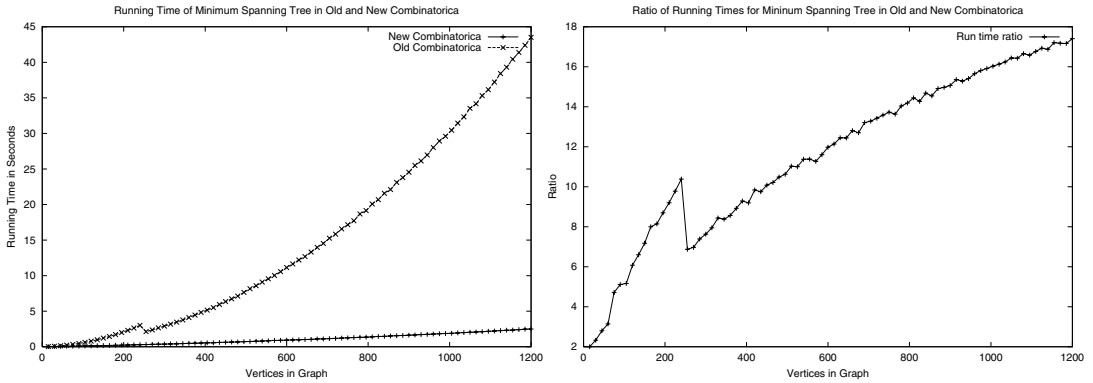

Figure 5.7: Performance comparison between old and new Combinatorica: absolute running times (l), and the ratio of these times (r).

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.

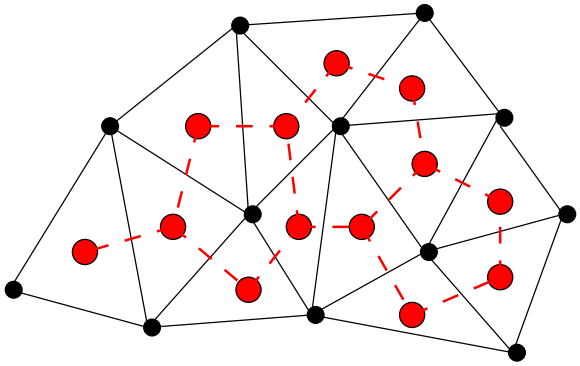

Figure 5.8: The dual graph (dashed lines) of a triangulation

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 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.

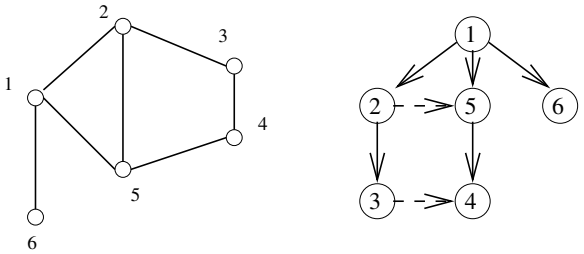

Figure 5.9: An undirected graph and its breadth-first search tree

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.

```
BFS(G, s)
    for each vertex u ∈ V[G] − {s} do
        state[u] = "undiscovered"
        p[u] = nil, i.e. no parent is in the BFS tree
    state[s] = "discovered"
    p[s] = nil
    Q = {s}
    while Q ≠ ∅ do
        u = dequeue[Q]
        process vertex u as desired
        for each v ∈ Adj[u] do
            process edge (u, v) as desired
            if state[v] = "undiscovered" then
                state[v] = "discovered"
                p[v] = u
                enqueue[Q, v]
        state[u] = "processed"
```

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.

```
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:

```
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:

```
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

```
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:

```
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:

vertex	1	2	3	4	5	6
parent	-1	1	2	5	1	1

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).

5.7 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 *read* any n-vertex, m-edge graph.

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

5.7.1 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 *connected* if there is a path between any two vertices. If the theory is true, it means the friendship graph must be connected.

A *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 `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 `process_vertex`.

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 *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:

DFS(G, u)
    $state[u]$ = "discovered"
    process vertex u if desired
    $entry[u] = time$

```
time = time + 1
for each v ∈ Adj[u] do
        process edge (u, v) if desired
        if state[v] = "undiscovered" then
                p[v] = u
                DFS(G, v)
state[u] = "processed"
exit[u] = time
time = time + 1
```

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.

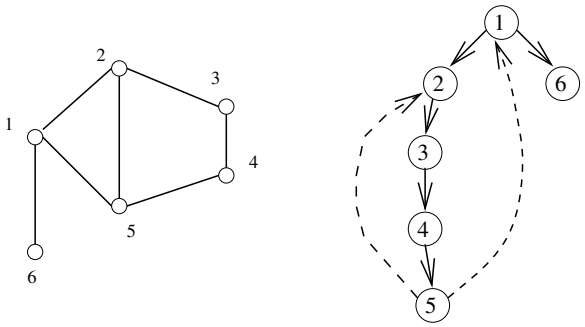

Figure 5.10: An undirected graph and its depth-first search tree

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:

```
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 (`process_vertex_early()`) or after we have finished processing all of them (`process_vertex_late()`). Sometimes we will take special actions at both times, say `process_vertex_early()` to initialize a vertex-specific data structure, which will be modified on edge-processing operations and then analyzed afterwards using `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)

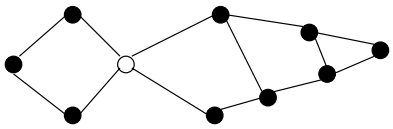

Figure 5.11: An articulation vertex is the weakest point in the graph

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 == 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 `dfs`:

```
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 `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

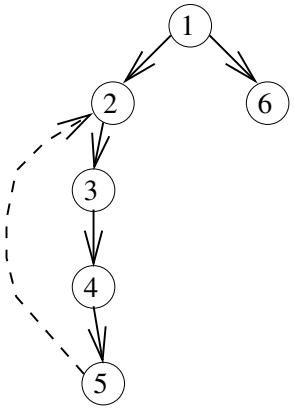

Figure 5.12: DFS tree of a graph containing two articulation vertices (namely 1 and 2). Back edge $(5, 2)$ keeps vertices 3 and 4 from being cut-nodes. Vertices 5 and 6 escape as leaves of the DFS tree

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.

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 `reachable_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, `reachable_ancestor[v] = v`:

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

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

We update `reachable_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:

```
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:

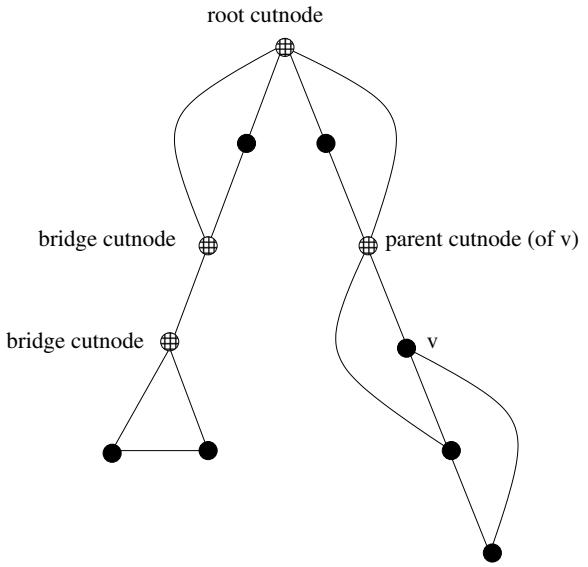

Figure 5.13: The three cases of articulation vertices: root, bridge, and parent cut-nodes

- *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: