4.6.3 Is Quicksort Really Quick?

There is a clear, asymptotic difference between an $\Theta(n \log n)$ algorithm and one that runs in $\Theta(n^2)$. Thus, only the most obstinate reader would doubt my claim that mergesort, heapsort, and quicksort should all outperform insertion sort or selection sort on large enough instances.

But how can we compare two $\Theta(n \log n)$ algorithms to decide which is faster? How can we prove that quicksort is really quick? Unfortunately, the RAM model and Big Oh analysis provide too coarse a set of tools to make that type of distinction. When faced with algorithms of the same asymptotic complexity, implementation details and system quirks such as cache performance and memory size may well prove to be the decisive factor.

What we can say is that experiments show that where a properly implemented quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort. The primary reason is that the operations in the innermost loop are simpler. But I can't argue with you if you don't believe me when I say quicksort is faster. It is a question whose solution lies outside the analytical tools we are using. The best way to tell is to implement both algorithms and experiment.

4.7 Distribution Sort: Sorting via Bucketing

We could sort sorting names for the telephone book by partitioning them according to the first letter of the last name. This will create 26 different piles, or buckets, of names. Observe that any name in the J pile must occur after every name in the I pile, but before any name in the K pile. Therefore, we can proceed to sort each pile individually and just concatenate the bunch of sorted piles together at the end.

If the names are distributed evenly among the buckets, the resulting 26 sorting problems should each be substantially smaller than the original problem. Further, by now partitioning each pile based on the second letter of each name, we generate smaller and smaller piles. The names will be sorted as soon as each bucket contains only a single name. The resulting algorithm is commonly called *bucketsort* or *distribution sort*.

Bucketing is a very effective idea whenever we are confident that the distribution of data will be roughly uniform. It is the idea that underlies hash tables, *kd*-trees, and a variety of other practical data structures. The downside of such techniques is that the performance can be terrible when the data distribution is not what we expected. Although data structures such as balanced binary trees offer guaranteed worst-case behavior for any input distribution, no such promise exists for heuristic data structures on unexpected input distributions.

Nonuniform distributions do occur in real life. Consider Americans with the uncommon last name of Shifflett. When last I looked, the Manhattan telephone directory (with over one million names) contained exactly five Shiffletts. So how many Shiffletts should there be in a small city of 50,000 people? Figure 4.8 shows

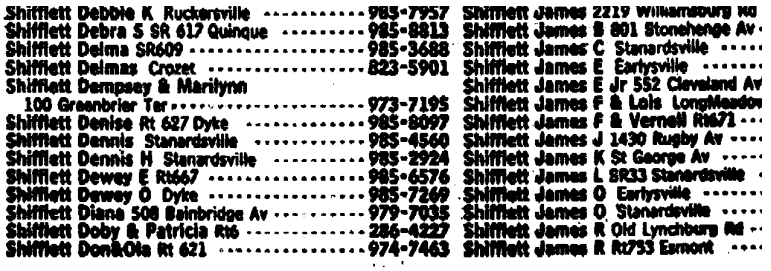


Figure 4.8: A small subset of Charlottesville Shiffletts

a small portion of the *two and a half pages* of Shiffletts in the Charlottesville, Virginia telephone book. The Shifflett clan is a fixture of the region, but it would play havoc with any distribution sort program, as refining buckets from *S* to *Sh* to *Shi* to *Shif* to ... to *Shifflett* results in no significant partitioning.

Take-Home Lesson: Sorting can be used to illustrate most algorithm design paradigms. Data structure techniques, divide-and-conquer, randomization, and incremental construction all lead to efficient sorting algorithms.

4.7.1 Lower Bounds for Sorting

One last issue on the complexity of sorting. We have seen several sorting algorithms that run in worst-case $O(n \log n)$ time, but none of which is linear. To sort n items certainly requires looking at all of them, so any sorting algorithm must be $\Omega(n)$ in the worst case. Can we close this remaining $\Theta(\log n)$ gap?

The answer is no. An $\Omega(n \log n)$ lower bound can be shown by observing that any sorting algorithm must behave differently during execution on each of the distinct $n!$ permutations of n keys. The outcome of each pairwise comparison governs the run-time behavior of any comparison-based sorting algorithm. We can think of the set of all possible executions of such an algorithm as a tree with $n!$ leaves. The minimum height tree corresponds to the fastest possible algorithm, and it happens that $\lg(n!) = \Theta(n \log n)$.

This lower bound is important for several reasons. First, the idea can be extended to give lower bounds for many applications of sorting, including element uniqueness, finding the mode, and constructing convex hulls. Sorting has one of the few nontrivial lower bounds among algorithmic problems. We will present an alternate approach to arguing that fast algorithms are unlikely to exist in Chapter 9.

4.8 War Story: Skiena for the Defense

I lead a quiet, reasonably honest life. One reward for this is that I don't often find myself on the business end of surprise calls from lawyers. Thus I was astonished to get a call from a lawyer who not only wanted to talk with me, but wanted to talk to me about sorting algorithms.

It turned out that her firm was working on a case involving high-performance programs for sorting, and needed an expert witness who could explain technical issues to the jury. From the first edition of this book, they could see I knew something about algorithms, but before taking me on they demanded to see my teaching evaluations to prove that I could explain things to people.[2] It proved to be a fascinating opportunity to learn about how *really* fast sorting programs work. I figured I could finally answer the question of which in-place sorting algorithm was fastest in practice. Was it heapsort or quicksort? What subtle, secret algorithmics made the difference to minimize the number of comparisons in practice?

The answer was quite humbling. *Nobody cared about in-place sorting.* The name of the game was sorting *huge* files, much bigger than could fit in main memory. All the important action was in getting the the data on and off a disk. Cute algorithms for doing internal (in-memory) sorting were not particularly important because the real problem lies in sorting gigabytes at a time.

Recall that disks have relatively long seek times, reflecting how long it takes the desired part of the disk to rotate under the read/write head. Once the head is in the right place, the data moves relatively quickly, and it costs about the same to read a large data block as it does to read a single byte. Thus, the goal is minimizing the number of blocks read/written, and coordinating these operations so the sorting algorithm is never waiting to get the data it needs.

The disk-intensive nature of sorting is best revealed by the annual *Minutesort* competition. The goal is to sort as much data in one minute as possible. The current champion is Jim Wyllie of IBM Research, who managed to sort 116 gigabytes of data in 58.7 seconds on his little old 40-node 80-Itanium cluster with a SAN array of 2,520 disks. Slightly more down-to-earth is the *Pennysort* division, where the goal is the maximized sorting performance per penny of hardware. The current champ here (*BSIS* from China) sorted 32 gigabytes in 1,679 seconds on a $760 PC containing four SATA drives. You can check out the current records at *http://research.microsoft.com/barc/SortBenchmark/*.

That said, which algorithm is best for external sorting? It basically turns out to be a multiway mergesort, employing a lot of engineering and special tricks. You build a heap with members of the top block from each of k sorted lists. By repeatedly plucking the top element off this heap, you build a sorted list merging these k lists. Because this heap is sitting in main memory, these operations are fast. When you have a large enough sorted run, you write it to disk and free up

[2] One of my more cynical faculty colleagues said this was the first time anyone, anywhere, had ever looked at university teaching evaluations.

memory for more data. Once you start to run out of elements from the top block of one of the k sorted lists you are merging, load the next block.

It proves very hard to benchmark sorting programs/algorithms at this level and decide which is *really* fastest. Is it fair to compare a commercial program designed to handle general files with a stripped-down code optimized for integers? The *Minutesort* competition employs randomly-generated 100-byte records. This is a different world than sorting names or integers. For example, one widely employed trick is to strip off a relatively short prefix of the key and initially sort just on that, to avoid lugging around all those extra bytes.

What lessons can be learned from this? The most important, by far, is to do everything you can to avoid being involved in a lawsuit as either a plaintiff or defendant.[3] Courts are not instruments for resolving disputes quickly. Legal battles have a lot in common with military battles: they escalate very quickly, become very expensive in time, money, and soul, and usually end only when both sides are exhausted and compromise. Wise are the parties who can work out their problems without going to court. Properly absorbing this lesson now could save you thousands of times the cost of this book.

On technical matters, it is important to worry about external memory performance whenever you combine very large datasets with low-complexity algorithms (say linear or $n \log n$). Constant factors of even 5 or 10 can make a big difference then between what is feasible and what is hopeless. Of course, quadratic-time algorithms are doomed to fail on large datasets regardless of data access times.

4.9 Binary Search and Related Algorithms

Binary search is a fast algorithm for searching in a sorted array of keys S. To search for key q, we compare q to the middle key $S[n/2]$. If q appears before $S[n/2]$, it must reside in the top half of S; if not, it must reside in the bottom half of S. By repeating this process recursively on the correct half, we locate the key in a total of $\lceil \lg n \rceil$ comparisons—a big win over the $n/2$ comparisons expect using sequential search:

```
int binary_search(item_type s[], item_type key, int low, int high)
{
      int middle;                    /* index of middle element */

      if (low > high) return (-1);  /* key not found */

      middle = (low+high)/2;
```

[3]It is actually quite interesting serving as an expert witness.

```
        if (s[middle] == key) return(middle);

        if (s[middle] > key)
                return( binary_search(s,key,low,middle-1) );
        else
                return(binary_search(s,key,middle+1,high) );
}
```

This much you probably know. What is important is to have a sense of just how fast binary search is. *Twenty questions* is a popular children's game where one player selects a word and the other repeatedly asks true/false questions in an attempt to guess it. If the word remains unidentified after 20 questions, the first party wins; otherwise, the second player takes the honors. In fact, the second player always has a winning strategy, based on binary search. Given a printed dictionary, the player opens it in the middle, selects a word (say "move"), and asks whether the unknown word is before "move" in alphabetical order. Since standard dictionaries contain 50,000 to 200,000 words, we can be certain that the process will terminate within twenty questions.

4.9.1 Counting Occurrences

Several interesting algorithms follow from simple variants of binary search. Suppose that we want to count the number of times a given key k (say "Skiena") occurs in a given sorted array. Because sorting groups all the copies of k into a contiguous block, the problem reduces to finding the right block and then measures its size.

The binary search routine presented above enables us to find the index of an element of the correct block (x) in $O(\lg n)$ time. The natural way to identify the boundaries of the block is to sequentially test elements to the left of x until we find the first one that differs from the search key, and then repeat this search to the right of x. The difference between the indices of the left and right boundaries (plus one) gives the count of the number of occurrences of k.

This algorithm runs in $O(\lg n + s)$, where s is the number of occurrences of the key. This can be as bad as linear if the entire array consists of identical keys. A faster algorithm results by modifying binary search to search for the *boundary* of the block containing k, instead of k itself. Suppose we delete the equality test

```
if (s[middle] == key) return(middle);
```

from the implementation above and return the index `low` instead of -1 on each unsuccessful search. *All* searches will now be unsuccessful, since there is no equality test. The search will proceed to the right half whenever the key is compared to an identical array element, eventually terminating at the right boundary. Repeating the search after reversing the direction of the binary comparison will lead us to the left boundary. Each search takes $O(\lg n)$ time, so we can count the occurrences in logarithmic time regardless of the size of the block.

4.9.2 One-Sided Binary Search

Now suppose we have an array A consisting of a run of 0's, followed by an unbounded run of 1's, and would like to identify the exact point of transition between them. Binary search on the array would provide the transition point in $\lceil \lg n \rceil$ tests, if we had a bound n on the number of elements in the array. In the absence of such a bound, we can test repeatedly at larger intervals ($A[1]$, $A[2]$, $A[4]$, $A[8]$, $A[16]$, ...) until we find a first nonzero value. Now we have a window containing the target and can proceed with binary search. This *one-sided binary search* finds the transition point p using at most $2\lceil \lg p \rceil$ comparisons, regardless of how large the array actually is. One-sided binary search is most useful whenever we are looking for a key that lies close to our current position.

4.9.3 Square and Other Roots

The square root of n is the number r such that $r^2 = n$. Square root computations are performed inside every pocket calculator, but it is instructive to develop an efficient algorithm to compute them.

First, observe that the square root of $n \geq 1$ must be at least 1 and at most n. Let $l = 1$ and $r = n$. Consider the midpoint of this interval, $m = (l + r)/2$. How does m^2 compare to n? If $n \geq m^2$, then the square root must be greater than m, so the algorithm repeats with $l = m$. If $n < m^2$, then the square root must be less than m, so the algorithm repeats with $r = m$. Either way, we have halved the interval using only one comparison. Therefore, after $\lg n$ rounds we will have identified the square root to within ± 1.

This bisection method, as it is called in numerical analysis, can also be applied to the more general problem of finding the roots of an equation. We say that x is a *root* of the function f if $f(x) = 0$. Suppose that we start with values l and r such that $f(l) > 0$ and $f(r) < 0$. If f is a continuous function, there must exist a root between l and r. Depending upon the sign of $f(m)$, where $m = (l + r)/2$, we can cut this window containing the root in half with each test and stop soon as our estimate becomes sufficiently accurate.

Root-finding algorithms that converge faster than binary search are known for both of these problems. Instead of always testing the midpoint of the interval, these algorithms interpolate to find a test point closer to the actual root. Still, binary search is simple, robust, and works as well as possible without additional information on the nature of the function to be computed.

Take-Home Lesson: Binary search and its variants are the quintessential divide-and-conquer algorithms.

4.10 Divide-and-Conquer

One of the most powerful techniques for solving problems is to break them down into smaller, more easily solved pieces. Smaller problems are less overwhelming, and they permit us to focus on details that are lost when we are studying the entire problem. A recursive algorithm starts to become apparent when we can break the problem into smaller instances of the same type of problem. Effective parallel processing requires decomposing jobs into at least as many tasks as processors, and is becoming more important with the advent of cluster computing and multicore processors.

Two important algorithm design paradigms are based on breaking problems down into smaller problems. In Chapter 8, we will see dynamic programming, which typically removes one element from the problem, solves the smaller problem, and then uses the solution to this smaller problem to add back the element in the proper way. *Divide-and-conquer* instead splits the problem in (say) halves, solves each half, then stitches the pieces back together to form a full solution.

To use divide-and-conquer as an algorithm design technique, we must divide the problem into two smaller subproblems, solve each of them recursively, and then meld the two partial solutions into one solution to the full problem. Whenever the merging takes less time than solving the two subproblems, we get an efficient algorithm. Mergesort, discussed in Section 4.5 (page 120), is the classic example of a divide-and-conquer algorithm. It takes only linear time to merge two sorted lists of $n/2$ elements, each of which was obtained in $O(n \lg n)$ time.

Divide-and-conquer is a design technique with many important algorithms to its credit, including mergesort, the fast Fourier transform, and Strassen's matrix multiplication algorithm. Beyond binary search and its many variants, however, I find it to be a difficult design technique to apply in practice. Our ability to analyze divide-and-conquer algorithms rests on our strength to solve the asymptotics of recurrence relations governing the cost of such recursive algorithms.

4.10.1 Recurrence Relations

Many divide-and-conquer algorithms have time complexities that are naturally modeled by recurrence relations. Evaluating such recurrences is important to understanding when divide-and-conquer algorithms perform well, and provide an important tool for analysis in general. The reader who balks at the very idea of analysis is free to skip this section, but there are important insights into design that come from an understanding of the behavior of recurrence relations.

What is a recurrence relation? It is an equation that is defined in terms of itself. The Fibonacci numbers are described by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ and discussed in Section 8.1.1. Many other natural functions are easily expressed as recurrences. Any polynomial can be represented by a recurrence, such as the linear function:

$$a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$$

Any exponential can be represented by a recurrence:

$$a_n = 2a_{n-1}, a_1 = 1 \longrightarrow a_n = 2^{n-1}$$

Finally, lots of weird functions that cannot be described easily with conventional notation can be represented by a recurrence:

$$a_n = na_{n-1}, a_1 = 1 \longrightarrow a_n = n!$$

This means that recurrence relations are a very versatile way to represent functions.

The self-reference property of recurrence relations is shared with recursive programs or algorithms, as the shared roots of both terms reflect. Essentially, recurrence relations provide a way to analyze recursive structures, such as algorithms.

4.10.2 Divide-and-Conquer Recurrences

Divide-and-conquer algorithms tend to break a given problem into some number of smaller pieces (say a), each of which is of size n/b. Further, they spend $f(n)$ time to combine these subproblem solutions into a complete result. Let $T(n)$ denote the worst-case time the algorithm takes to solve a problem of size n. Then $T(n)$ is given by the following recurrence relation:

$$T(n) = aT(n/b) + f(n)$$

Consider the following examples:

- *Sorting* – The running time behavior of mergesort is governed by the recurrence $T(n) = 2T(n/2) + O(n)$, since the algorithm divides the data into equal-sized halves and then spends linear time merging the halves after they are sorted. In fact, this recurrence evaluates to $T(n) = O(n \lg n)$, just as we got by our previous analysis.

- *Binary Search* – The running time behavior of binary search is governed by the recurrence $T(n) = T(n/2) + O(1)$, since at each step we spend constant time to reduce the problem to an instance half its size. In fact, this recurrence evaluates to $T(n) = O(\lg n)$, just as we got by our previous analysis.

- *Fast Heap Construction* – The `bubble_down` method of heap construction (described in Section 4.3.4) built an n-element heap by constructing two $n/2$ element heaps and then merging them with the root in logarithmic time. This argument reduces to the recurrence relation $T(n) = 2T(n/2) + O(\lg n)$. In fact, this recurrence evaluates to $T(n) = O(n)$, just as we got by our previous analysis.

- *Matrix Multiplication* – As discussed in Section 2.5.4, the standard matrix multiplication algorithm for two $n \times n$ matrices takes $O(n^3)$, because we compute the dot product of n terms for each of the n^2 elements in the product matrix.

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 $O(n^{\log_b a} \lg n)$.

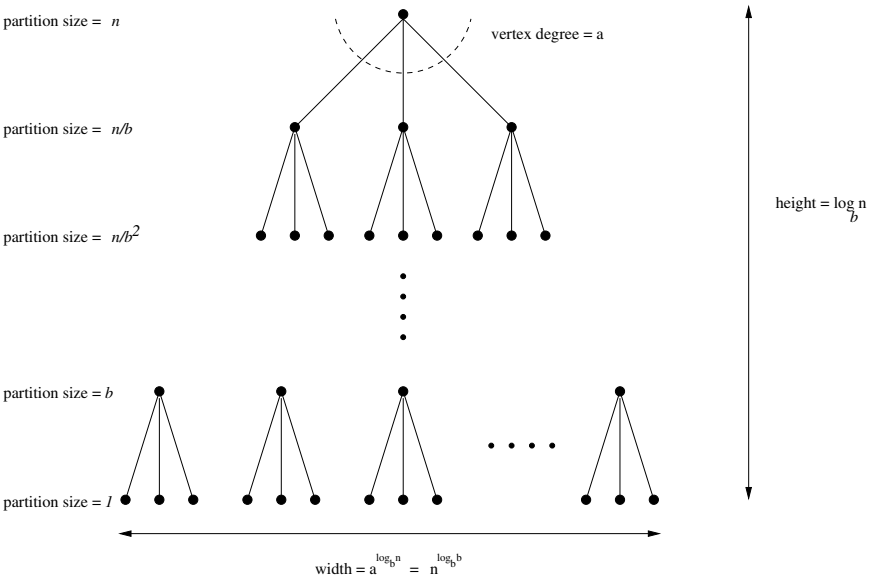


Figure 4.9: The recursion tree resulting from decomposing each problem of size n into a problems of size n/b

- *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.

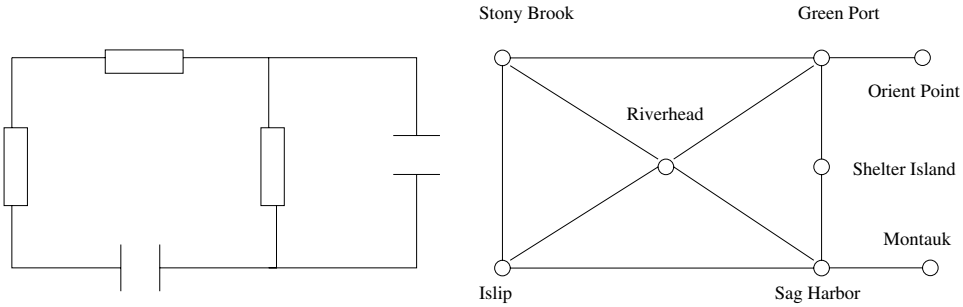


Figure 5.1: Modeling road networks and electronic circuits as graphs

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

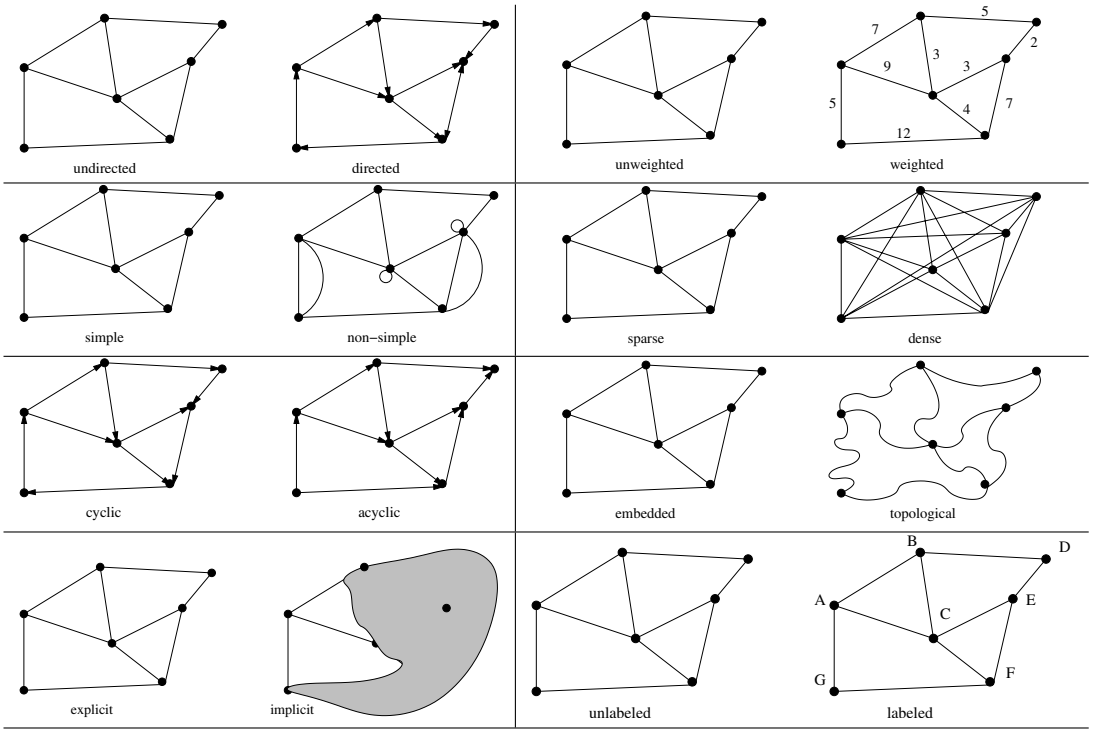


Figure 5.2: Important properties / flavors of graphs

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.

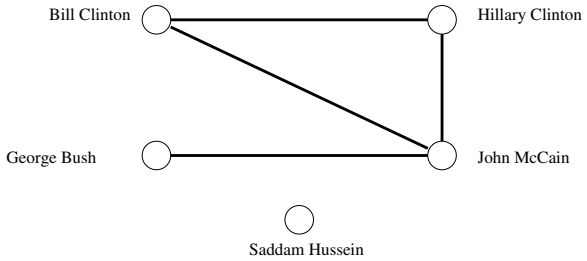


Figure 5.3: A portion of the friendship graph

- *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

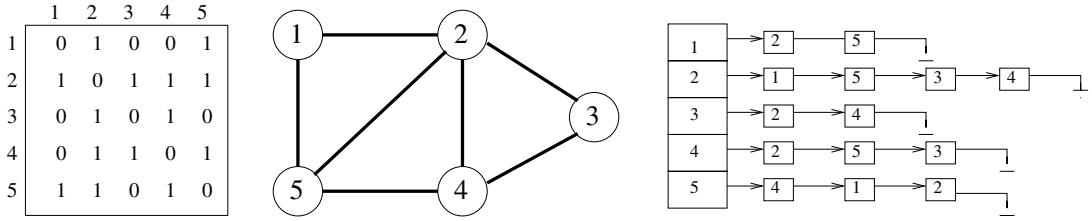


Figure 5.4: The adjacency matrix and adjacency list of a given graph

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

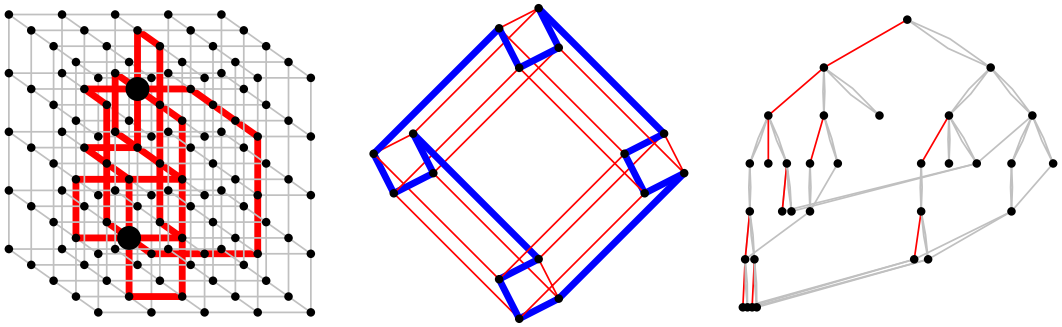


Figure 5.6: Representative *Combinatorica* graphs: edge-disjoint paths (left), Hamiltonian cycle in a hypercube (center), animated depth-first search tree traversal (right)

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.

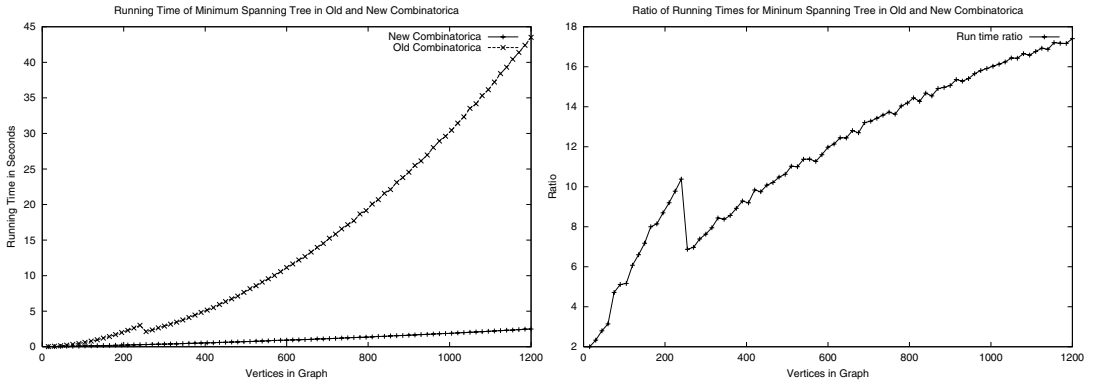


Figure 5.7: Performance comparison between old and new Combinatorica: absolute running times (l), and the ratio of these times (r).

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.

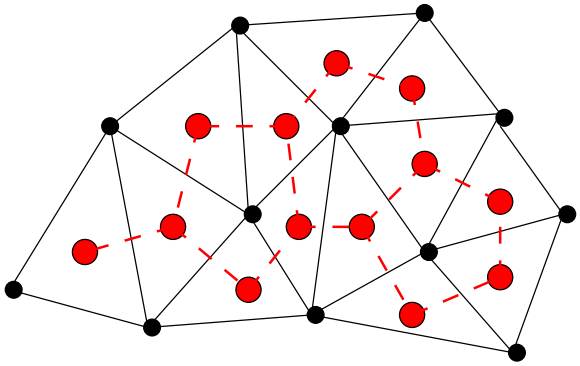


Figure 5.8: The dual graph (dashed lines) of a triangulation

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.

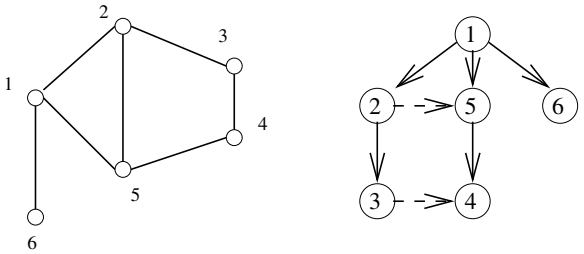


Figure 5.9: An undirected graph and its breadth-first search tree

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.

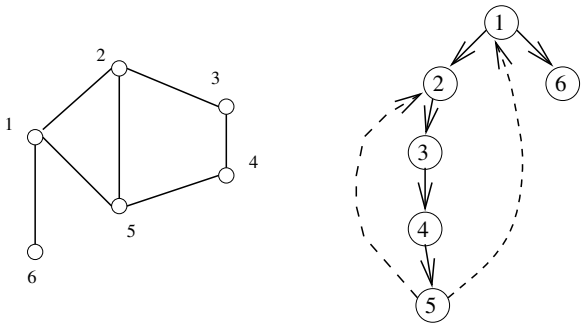


Figure 5.10: An undirected graph and its depth-first search tree

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)

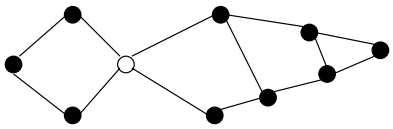


Figure 5.11: An articulation vertex is the weakest point in the graph

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

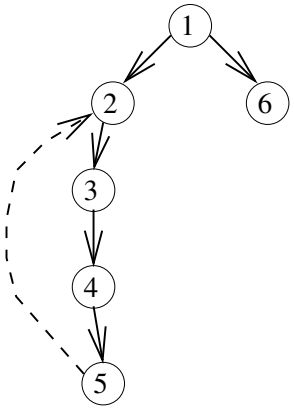


Figure 5.12: DFS tree of a graph containing two articulation vertices (namely 1 and 2). Back edge $(5, 2)$ keeps vertices 3 and 4 from being cut-nodes. Vertices 5 and 6 escape as leaves of the DFS tree

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:

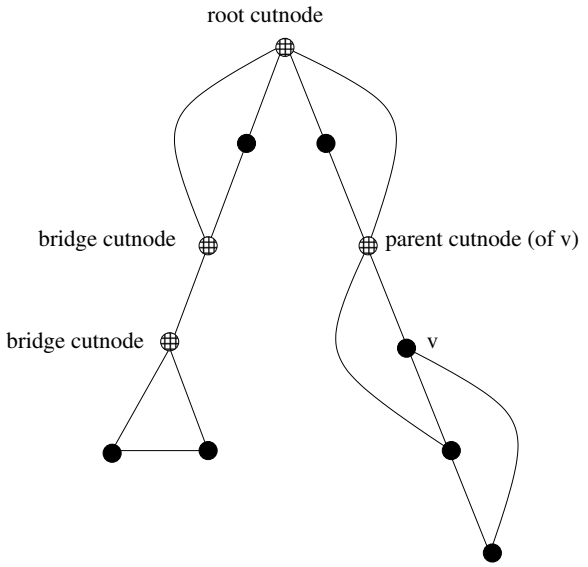


Figure 5.13: The three cases of articulation vertices: root, bridge, and parent cut-nodes

- *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:

```
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.

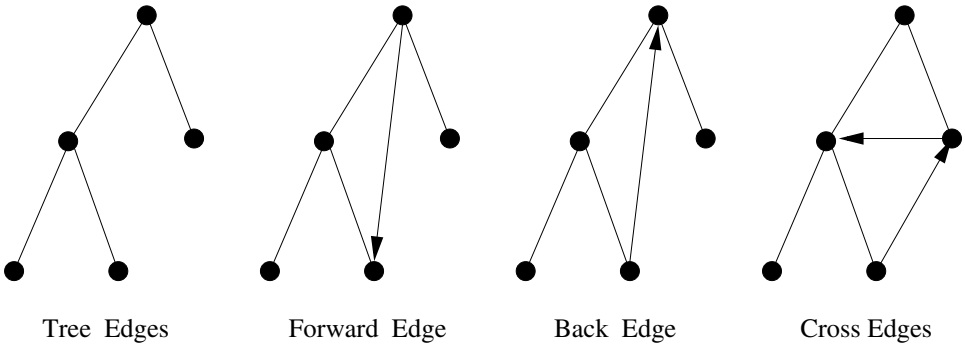


Figure 5.14: Possible edge cases for BFS/DFS traversal

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:

```
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);
}
```

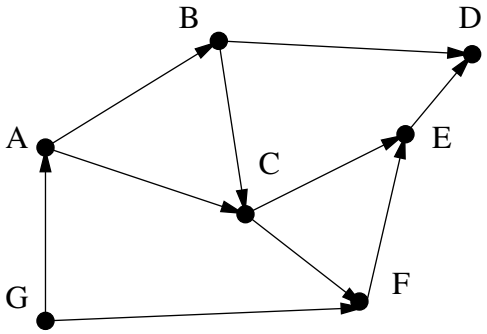


Figure 5.15: A DAG with only one topological sort (G, A, B, C, F, E, D)

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:

```
DFS-graph(G)
    for each vertex u ∈ V[G] do
        state[u] = "undiscovered"
            for each vertex u ∈ V[G] do
        if state[u] = "undiscovered" then
            initialize new component, if desired
            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:

```
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.

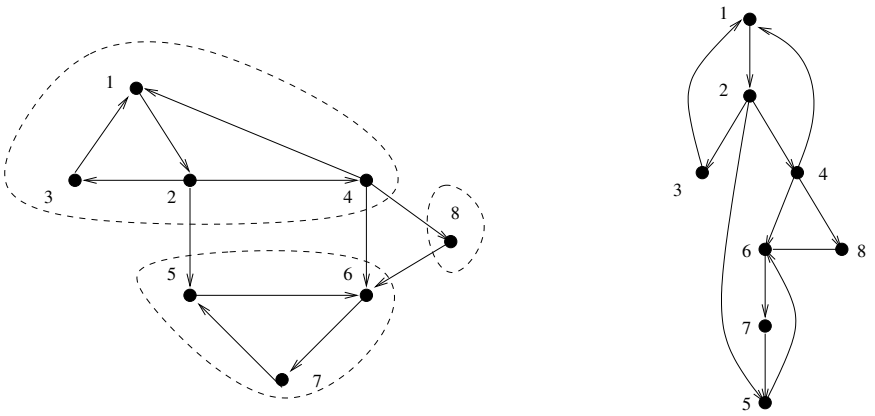


Figure 5.16: The strongly-connected components of a graph, with the associated DFS tree

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 an nonancestor but previously discovered vertex). Our algorithm will peal one strong component off the tree at a time, and assign each of its vertices the number of the component it is in:

```
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);
                }
}
```

Define `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:

```
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:

```
process_vertex_early(int v)
{
        push(&active,v);
}
```

```
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;
      }
}
```

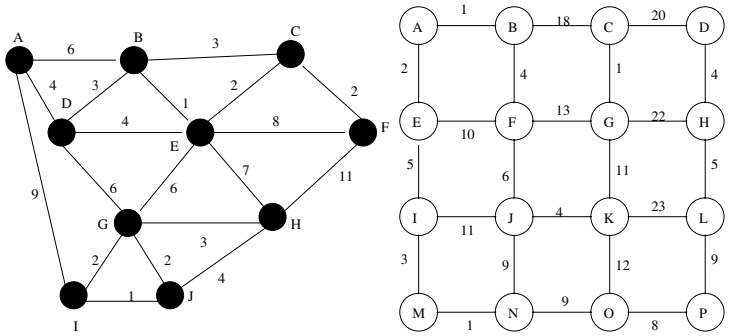
Chapter Notes

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

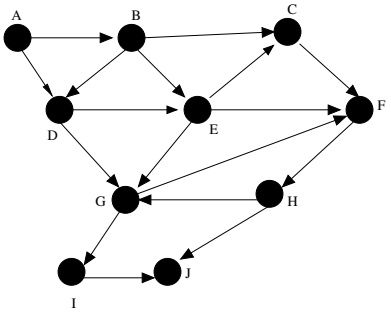
5-1. *[3]* For the following graphs G_1 (left) and G_2 (right):



(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

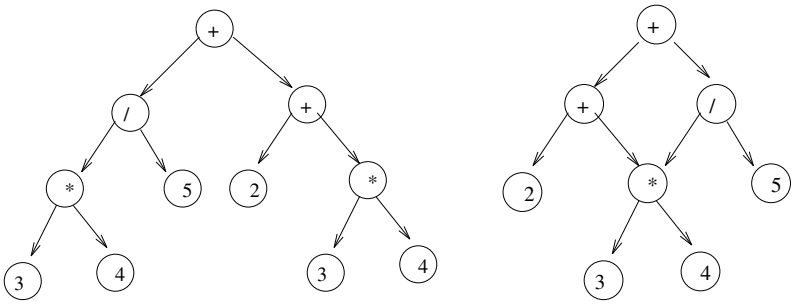


Figure 5.17: Expression $2 + 3 * 4 + (3 * 4)/5$ as a tree and a DAG

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 $2 + 3 * 4 + (3 * 4)/5$ 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 $2 + 3 * 4 + (3 * 4)/5$ 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.

5-11. *[8]* The war story of Section 5.4 (page 158) describes an algorithm for constructing the dual graph of the triangulation efficiently, although it does not guarantee linear time. Give a worst-case linear algorithm for the problem.

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 $V' \in V$ 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 $V' \in V$ 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 $V' \in V$ 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 ith 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.

5-2. "Playing with Wheels" – Programming Challenges 110902, UVA Judge 10067.

5-3. "The Tourist Guide" – Programming Challenges 110903, UVA Judge 10099.

5-4. "Edit Step Ladders" – Programming Challenges 110905, UVA Judge 10029.

5-5. "Tower of Cubes" – Programming Challenges 110906, UVA Judge 10051.

6

Weighted Graph Algorithms

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 `edges[x]`:

```
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 `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 (`weight`), and the third pointing to the next edge in the list (`next`):

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_6,

```
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

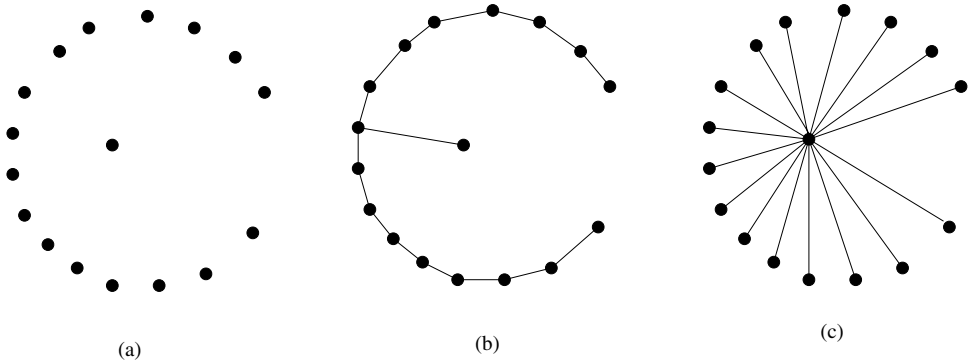


Figure 6.1: (a) Two spanning trees of point set; (b) the minimum spanning tree, and (c) the shortest path from center tree

---

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_{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_{prim} consisted of a set of edges that was a subtree of some minimum spanning tree T_{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_{min}, as shown in Figure 6.2(b). This path must use an edge (v_1, v_2), where v_1 is in T_{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_{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.

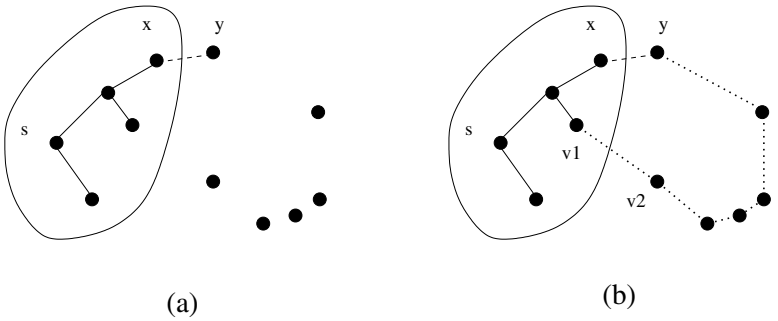


Figure 6.2: Where Prim's algorithm goes bad? No, because $d(v_1, v_2) \geq d(x, y)$

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:

```
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;
```

```
            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

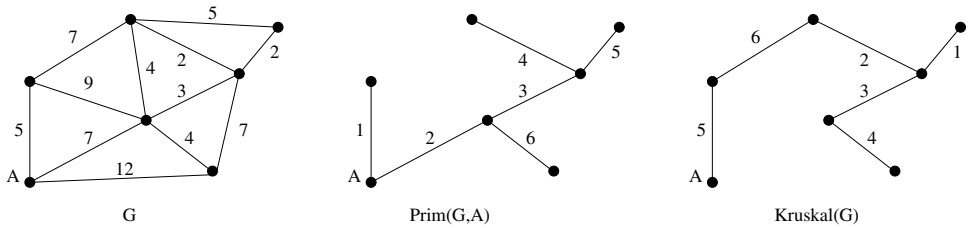


Figure 6.3: A graph G (l) with minimum spanning trees produced by Prim's (m) and Kruskal's (r) algorithms. The numbers on the trees denote the order of insertion; ties are broken arbitrarily

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) ≠ component(w))
            add to T_kruskal
            merge component(v) and component(w)
```

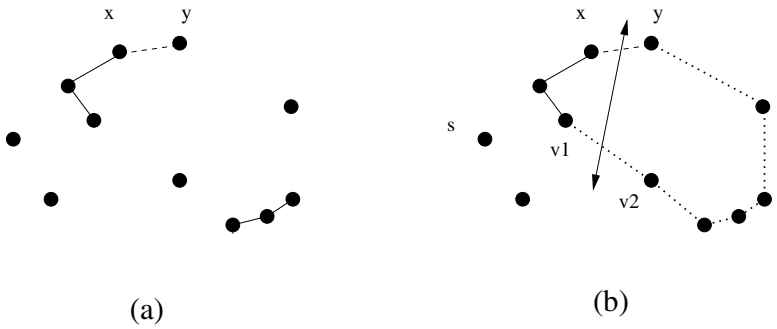


Figure 6.4: Where Kruskal's algorithm goes bad? No, because $d(v_1, v_2) \geq d(x, y)$

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 a single edge (x, y) whose insertion first prevented the tree $T_{kruskal}$ from being a minimum spanning tree T_{min}. Inserting this edge (x, y) into T_{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_{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 $O(m \lg m)$ 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 $O(\lg n)$ time. Union-find is discussed in the next section. With this data structure, Kruskal's algorithm runs in $O(m \lg m)$ 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 psuedocode:

```
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

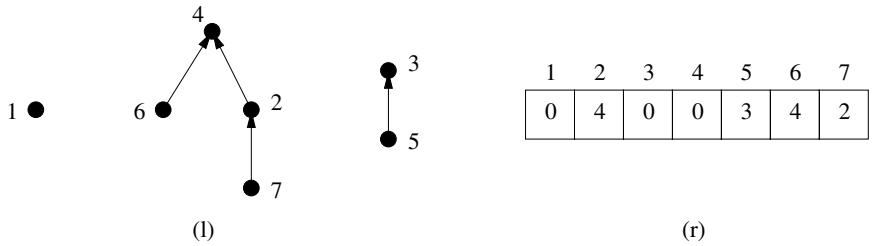


Figure 6.5: Union-find example: structure represented as trees (l) and array (r)

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:

```
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:

```
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, $\lg_2 n$ 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 $\lg(a \cdot b) = \lg(a) + \lg(b)$, 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?"

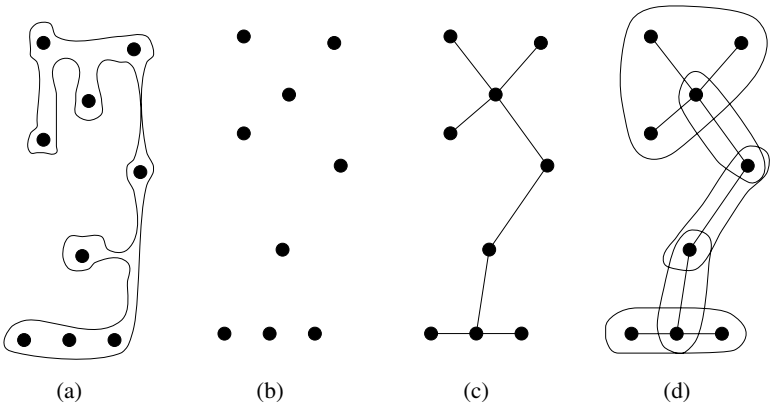


Figure 6.6: An example net showing (a) the metal connection layer, (b) the contact points, (c) their minimum spanning tree, and (d) the points partitioned into clusters

"Circuit boards are certain sets of points that are all connected together with a metal layer. This is what we mean by a net. Sometimes a net consists of two points—i.e. , an isolated wire. Sometimes a net can have 100 to 200 points, like all the connections to power or ground."

"I see. So you have a list of all the connections between pairs of points on the circuit board, and you want to trace out these wires."

He shook his head. "Not quite. The input for our testing program consists only of the net contact points, as shown in Figure 6.6(b). We don't know where the actual wires are, but we don't have to. All we must do is verify that all the points in a net are connected together. We do this by putting the left robot arm on the leftmost point in the net, and then have the right arm move around to all the other points in the net to test if they are all connected to the left point. So they must all be connected to each other."

I thought for a moment about what this meant. "OK. So your right arm has to visit all the other points in the net. How do you choose the order to visit them?"

The technical guy spoke up. "Well, we sort the points from left to right and then go in that order. Is that a good thing to do?"

"Have you ever heard of the traveling salesman problem?" I asked.

He was an electrical engineer, not a computer scientist. "No, what's that?"

"Traveling salesman is the name of the problem that you are trying to solve. Given a set of points to visit, how do you order them to minimize the travel time. Algorithms for the traveling salesman problem have been extensively studied. For small nets, you will be able to find the optimal tour by doing an exhaustive search. For big nets, there are heuristics that will get you very close to the optimal tour." I would have pointed them to Section 16.4 (page 533) if I had had this book handy.

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 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_∞ 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

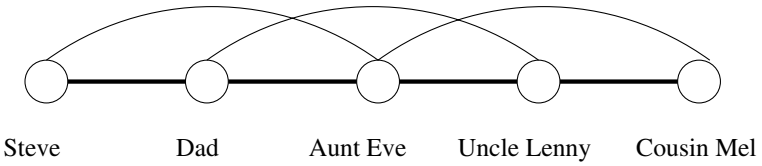


Figure 6.7: Mel Brooks is my father's sister's husband's cousin

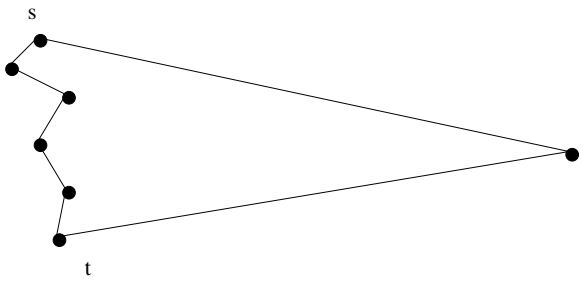


Figure 6.8: The shortest path from s to t may pass through many intermediate vertices

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 $dist(s, v_i) + w(v_i, x)$ over all unfinished $1 \leq i \leq n$, where $w(i, j)$ is the length of the edge from i to j, and $dist(i, j)$ 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 $dist(s, s) = 0$. If (s, y) is the lightest edge incident to s, then this implies that $dist(s, y) = w(s, y)$. 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.

```
ShortestPath-Dijkstra(G, s, t)
    known = {s}
    for i = 1 to n, dist[i] = ∞
    for each edge (s, v), dist[v] = w(s, v)
    last = s
    while (last ≠ t)
        select v_next, the unknown vertex minimizing dist[v]
        for each edge (v_next, x), dist[x] = min[dist[x], dist[v_next] + w(v_next, x)]
        last = v_next
        known = known ∪ {v_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:

```
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 `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 `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 kth 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 kth 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:

```
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 yeasr ain our ectherr arotght eosti on ugis aootinent a oey oation aoncdivee in licesty ane eedicatee un uhe rrorosition uiat all oen are arectee e ual
>
> ony ye are enichde in a irect aitil yar uestini yhethes uiat oatioo or aoy oation ro aoncdivee ane ro eedicatee aan loni eneure ye are oet on a irect aattlediele oe uiat yar ye iate aone un eedicate a rostion oe uiat eiele ar a einal restini rlace eor uiore yin iere iate uhdis lives uiat uhe oation ogght live it is aluniethes eittini ane rrores uiat ye rioule en ugir
>
> att in a laries reore ye aan oou eedicate ye aan oou aoorearate ye aan oou ialloy ugis iroune the arate oen litini ane eeae yin rustgilee iere iate aoorearatee it ear aante our roor rowes un ade or eeuraat the yople yill little oote oor loni renences yiat ye ray iere att it aan oetes eosiet yiat uhfy eie iere it is eor ur uhe litini rathes un ae eedicatee iere un uhe undiniside yopl yhici uhfy yin entght iere iate uiur ear ro onaky aetancde it is rathes eor ur un ae iere eedicatee un uhe irect uarl rencinini adeore ur uiat eron uhere ioooree eeae ye uale inarearee eeuotion uo tiat aaure eor yhici uhfy iere iate uhe lart eull oearure oe eeuotioo tiat ye iere iggily rerolue uiat uhere eeae 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 eatherr brought forth on this continent azoey nation conceivee 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 conceivee and ro dedicatee aan 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 fittinizane proper that ye should en this
>
> aut in a larges sense ye aan oou dedicate ye aan oou consecrate ye aan oou 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

> ray here aut it aan meter forget what uhfy die here it is for ur uhe litioi rather uo ae dedicatee here uo uhe toeioisgee york which uhfy yin fought here iate thus ear ro mocky advancee it is rather for ur uo ae here dedicatee uo uhe great task renagogoi adfore ur that from there honoree deae ye uale increasee devotion uo that aause for which uhfy here iate uhe last eull measure oe devotion that ye here highky resolve that there deae shall oou iate fide io vain that this nation under ioe shall iate azoey birth oe freedom and that ioternmenu oe uhe people ay uhe people for uhe people shall oou perish from uhe earth

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 ith 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?"

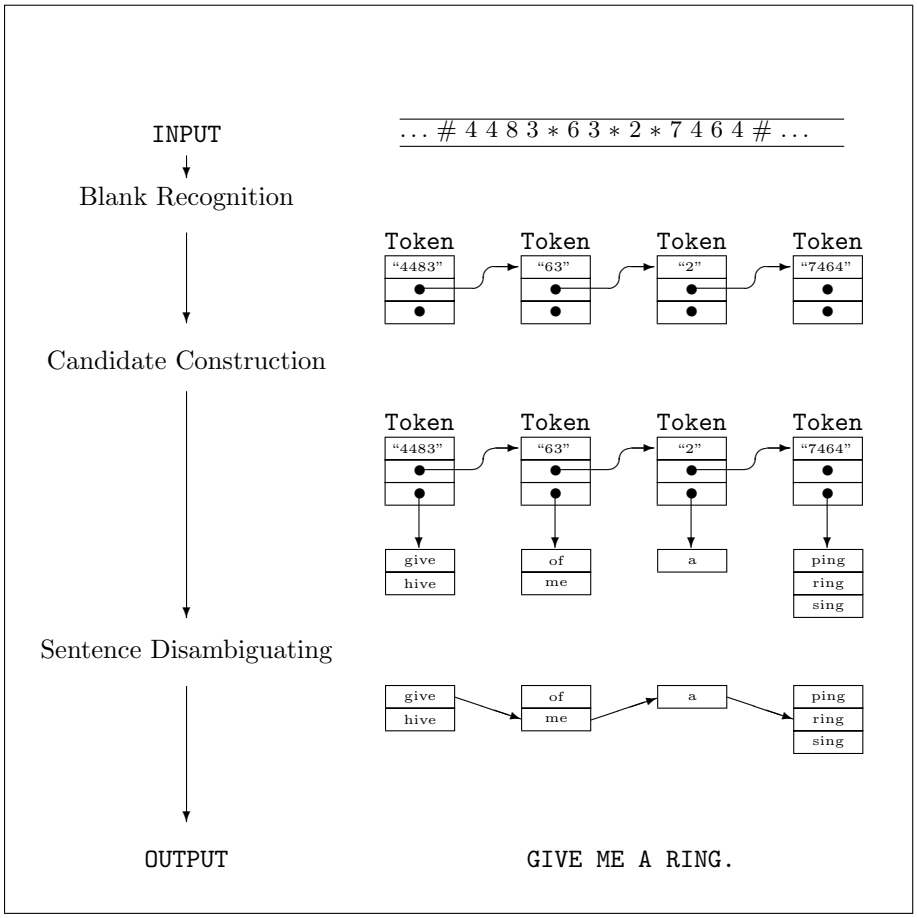


Figure 6.9: The phases of the telephone code reconstruction process

"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:

FOURSCORE 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

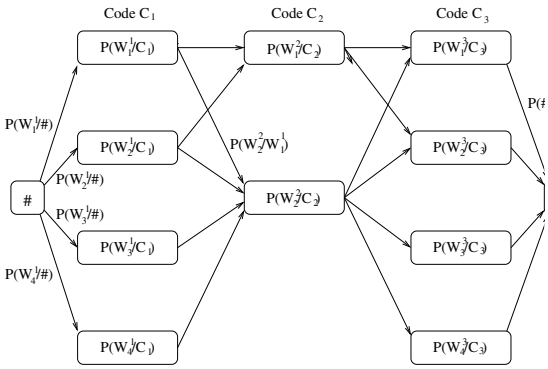


Figure 6.10: The minimum-cost path defines the best interpretation for a sentence

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

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** 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.

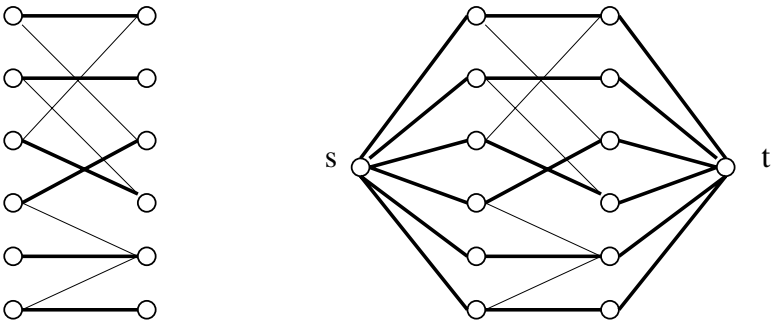


Figure 6.12: Bipartite graph with a maximum matching highlighted (on left). The corresponding network flow instance highlighting the maximum $s-t$ flow (on right).

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, where as 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 to 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.

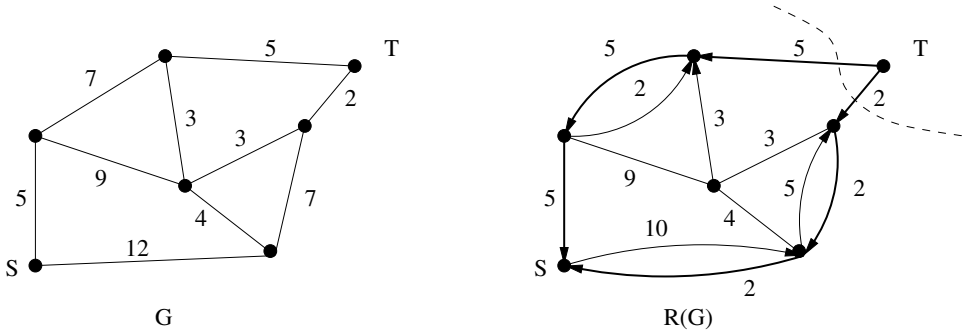


Figure 6.13: Maximum $s-t$ flow in a graph G (on left) showing the associated residual graph $R(G)$ and minimum $s-t$ cut (dotted line near t)

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:

```
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.

```
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:

```
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 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 $(\overrightarrow{l, f})$ from any fragment l that is forced to be to the left

of f, and a directed edge $(\overrightarrow{f, r})$ 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.

6-4. *[3]* Can Prim's and Kruskal's algorithm yield different minimum spanning trees? Explain why or why not.

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 ∞). 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 $O(n^2 m)$ 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 $c_1, ..., c_n$ 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

$$R[c_1, c_{i1}] \cdot R[c_{i1}, c_{i2}] \cdots R[c_{ik-1}, c_{ik}] \cdot R[c_{ik}, c_1]$$

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.

6-4. "Tourist Guide" – Programming Challenges 111006, UVA Judge 10199.

6-5. "The Grand Dinner" – Programming Challenges 111007, UVA Judge 10249.

7

Combinatorial Search and Heuristic Methods

We can solve many problems to optimality using exhaustive search techniques, although the time complexity can be enormous. For certain applications, it may pay to spend extra time to be certain of the optimal solution. A good example occurs in testing a circuit or a program on all possible inputs. You can prove the correctness of the device by trying all possible inputs and verifying that they give the correct answer. Verifying correctness is a property to be proud of. However, claiming that it works correctly on all the inputs you tried is worth much less.

Modern computers have clock rates of a few *gigahertz*, meaning billions of operations per second. Since doing something interesting takes a few hundred instructions, you can hope to search millions of items per second on contemporary machines.

It is important to realize how big (or how small) one million is. One million permutations means all arrangements of roughly 10 or 11 objects, but not more. One million subsets means all combinations of roughly 20 items, but not more. Solving significantly larger problems requires carefully pruning the search space to ensure we look at only the elements that really matter.

In this section, we introduce backtracking as a technique for listing all possible solutions for a combinatorial algorithm problem. We illustrate the power of clever pruning techniques to speed up real search applications. For problems that are too large to contemplate using brute-force combinatorial search, we introduce heuristic methods such as simulated annealing. Such heuristic methods are important weapons in any practical algorist's arsenal.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_7,

7.1 Backtracking

Backtracking is a systematic way to iterate through all the possible configurations of a search space. These configurations may represent all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). Other situations may demand enumerating all spanning trees of a graph, all paths between two vertices, or all possible ways to partition vertices into color classes.

What these problems have in common is that we must generate each one possible configuration exactly once. Avoiding both repetitions and missing configurations means that we must define a systematic generation order. We will model our combinatorial search solution as a vector $a = (a_1, a_2, ..., a_n)$, where each element a_i is selected from a finite ordered set S_i. Such a vector might represent an arrangement where a_i contains the ith element of the permutation. Or, the vector might represent a given subset S, where a_i is true if and only if the ith element of the universe is in S. The vector can even represent a sequence of moves in a game or a path in a graph, where a_i contains the ith event in the sequence.

At each step in the backtracking algorithm, we try to extend a given partial solution $a = (a_1, a_2, ..., a_k)$ by adding another element at the end. After extending it, we must test whether what we now have is a solution: if so, we should print it or count it. If not, we must check whether the partial solution is still potentially extendible to some complete solution.

Backtracking constructs a tree of partial solutions, where each vertex represents a partial solution. There is an edge from x to y if node y was created by advancing from x. This tree of partial solutions provides an alternative way to think about backtracking, for the process of constructing the solutions corresponds exactly to doing a depth-first traversal of the backtrack tree. Viewing backtracking as a depth-first search on an implicit graph yields a natural recursive implementation of the basic algorithm.

```
Backtrack-DFS(A, k)
    if A = (a_1, a_2, ..., a_k) is a solution, report it.
    else
        k = k + 1
        compute S_k
        while S_k ≠ ∅ do
            a_k = an element in S_k
            S_k = S_k − a_k
            Backtrack-DFS(A, k)
```

Although a breadth-first search could also be used to enumerate solutions, a depth-first search is greatly preferred because it uses much less space. The current state of a search is completely represented by the path from the root to the current search depth-first node. This requires space proportional to the *height* of the tree. In breadth-first search, the queue stores all the nodes at the current level, which

is proportional to the *width* of the search tree. For most interesting problems, the width of the tree grows exponentially in its height.

Implementation

The honest working `backtrack` code is given below:

```
bool finished = FALSE;            /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
     int c[MAXCANDIDATES];     /* candidates for next position */
     int ncandidates;          /* next position candidate count */
     int i;                    /* counter */

     if (is_a_solution(a,k,input))
             process_solution(a,k,input);
     else {
             k = k+1;
             construct_candidates(a,k,input,c,&ncandidates);
             for (i=0; i<ncandidates; i++) {
                     a[k] = c[i];
                     make_move(a,k,input);
                     backtrack(a,k,input);
                     unmake_move(a,k,input);
                     if (finished) return;   /* terminate early */
             }
     }
}
```

Backtracking ensures correctness by enumerating all possibilities. It ensures efficiency by never visiting a state more than once.

Study how recursion yields an elegant and easy implementation of the backtracking algorithm. Because a new candidates array `c` is allocated with each recursive procedure call, the subsets of not-yet-considered extension candidates at each position will not interfere with each other.

The application-specific parts of this algorithm consists of five subroutines:

- `is_a_solution(a,k,input)` – This Boolean function tests whether the first k elements of vector a from a complete solution for the given problem. The last argument, `input`, allows us to pass general information into the routine. We can use it to specify n—the size of a target solution. This makes sense when constructing permutations or subsets of n elements, but other data may be relevant when constructing variable-sized objects such as sequences of moves in a game.

- `construct_candidates(a,k,input,c,ncandidates)` – This routine fills an array c with the complete set of possible candidates for the kth position of a, given the contents of the first $k-1$ positions. The number of candidates returned in this array is denoted by `ncandidates`. Again, `input` may be used to pass auxiliary information.

- `process_solution(a,k,input)` – This routine prints, counts, or however processes a complete solution once it is constructed.

- `make_move(a,k,input)` *and* `unmake_move(a,k,input)` – These routines enable us to modify a data structure in response to the latest move, as well as clean up this data structure if we decide to take back the move. Such a data structure could be rebuilt from scratch from the solution vector a as needed, but this is inefficient when each move involves incremental changes that can easily be undone.

  These calls function as null stubs in all of this section's examples, but will be employed in the Sudoku program of Section 7.3 (page 239).

We include a global `finished` flag to allow for premature termination, which could be set in any application-specific routine.

To really understand how backtracking works, you must see how such objects as permutations and subsets can be constructed by defining the right state spaces. Examples of several state spaces are described in subsections below.

7.1.1 Constructing All Subsets

A critical issue when designing state spaces to represent combinatorial objects is how many objects need representing. How many subsets are there of an n-element set, say the integers $\{1, \ldots, n\}$? There are exactly two subsets for $n = 1$, namely $\{\}$ and $\{1\}$. There are four subsets for $n = 2$, and eight subsets for $n = 3$. Each new element doubles the number of possibilities, so there are 2^n subsets of n elements.

Each subset is described by elements that are in it. To construct all 2^n subsets, we set up an array/vector of n cells, where the value of a_i (true or false) signifies whether the ith item is in the given subset. In the scheme of our general backtrack algorithm, $S_k = (true, false)$ and a is a solution whenever $k = n$. We can now construct all subsets with simple implementations of `is_a_solution()`, `construct_candidates()`, and `process_solution()`.

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);                /* is k == n? */
}
```

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
        c[0] = TRUE;
        c[1] = FALSE;
        *ncandidates = 2;
}


process_solution(int a[], int k)
{
        int i;                          /* counter */

        printf("{");
        for (i=1; i<=k; i++)
                if (a[i] == TRUE) printf(" %d",i);

        printf(" }\n");
}
```

Printing each out subset after constructing it proves to be the most complicated of the three routines!

Finally, we must instantiate the call to `backtrack` with the right arguments. Specifically, this means giving a pointer to the empty solution vector, setting $k = 0$ to denote that it is empty, and specifying the number of elements in the universal set:

```
generate_subsets(int n)
{
        int a[NMAX];                    /* solution vector */

        backtrack(a,0,n);
}
```

In what order will the subsets of $\{1, 2, 3\}$ be generated? It depends on the order of moves `construct_candidates`. Since *true* always appears before *false*, the subset of all trues is generated first, and the all-false empty set is generated last: {123}, {12}, {13}, {1}, {23}, {2}, {3}, {}

Trace through this example carefully to make sure you understand the backtracking procedure. The problem of generating subsets is more thoroughly discussed in Section 14.5 (page 452).

7.1.2 Constructing All Permutations

Counting permutations of $\{1, \ldots, n\}$ is a necessary prerequisite to generating them. There are n distinct choices for the value of the first element of a permutation. Once

we have fixed a_1, there are $n-1$ candidates remaining for the second position, since we can have any value except a_1 (repetitions are forbidden in permutation). Repeating this argument yields a total of $n! = \prod_{i=1}^{n} i$ distinct permutations.

This counting argument suggests a suitable representation. Set up an array/vector a of n cells. The set of candidates for the ith position will be the set of elements that have not appeared in the $(i-1)$ elements of the partial solution, corresponding to the first $i-1$ elements of the permutation.

In the scheme of the general backtrack algorithm, $S_k = \{1, \ldots, n\} - a$, and a is a solution whenever $k = n$:

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
        int i;                          /* counter */
        bool in_perm[NMAX];             /* who is in the permutation? */

        for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
        for (i=0; i<k; i++) in_perm[ a[i] ] = TRUE;

        *ncandidates = 0;
        for (i=1; i<=n; i++)
                if (in_perm[i] == FALSE) {
                        c[ *ncandidates] = i;
                        *ncandidates = *ncandidates + 1;
                }
}
```

Testing whether i is a candidate for the kth slot in the permutation can be done by iterating through all $k-1$ elements of a and verifying that none of them matched. However, we prefer to set up a bit-vector data structure (see Section 12.5 (page 385)) to maintain which elements are in the partial solution. This gives a constant-time legality check.

Completing the job requires specifying `process_solution` and `is_a_solution`, as well as setting the appropriate arguments to `backtrack`. All are essentially the same as for subsets:

```
process_solution(int a[], int k)
{
        int i;                          /* counter */

        for (i=1; i<=k; i++) printf(" %d",a[i]);

        printf("\n");
}
```

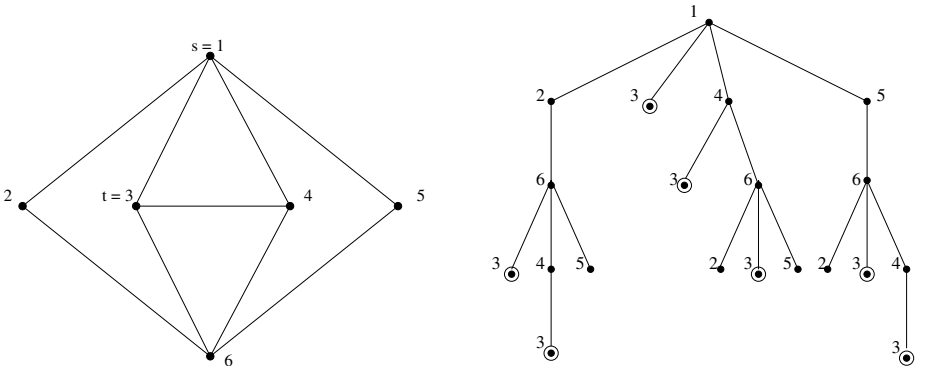


Figure 7.1: Search tree enumerating all simple s-t paths in the given graph (left).

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);
}


generate_permutations(int n)
{
        int a[NMAX];                    /* solution vector */

        backtrack(a,0,n);
}
```

As a consequence of the candidate order, these routines generate permutations in *lexicographic*, or sorted order—i.e., 123, 132, 213, 231, 312, and 321. The problem of generating permutations is more thoroughly discussed in Section 14.4 (page 448).

7.1.3 Constructing All Paths in a Graph

Enumerating all the simple s to t paths through a given graph is a more complicated problem than listing permutations or subsets. There is no explicit formula that counts the number of solutions as a function of the number of edges or vertices, because the number of paths depends upon the structure of the graph.

The starting point of any path from s to t is always s. Thus, s is the only candidate for the first position and $S_1 = \{s\}$. The possible candidates for the second position are the vertices v such that (s, v) is an edge of the graph, for the path wanders from vertex to vertex using edges to define the legal steps. In general,

S_{k+1} consists of the set of vertices adjacent to a_k that have not been used elsewhere in the partial solution A.

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
     int i;                    /* counters */
     bool in_sol[NMAX];        /* what's already in the solution? */
     edgenode *p;              /* temporary pointer */
     int last;                 /* last vertex on current path */

     for (i=1; i<NMAX; i++) in_sol[i] = FALSE;
     for (i=1; i<k; i++) in_sol[ a[i] ] = TRUE;

     if (k==1) {               /* always start from vertex 1 */
             c[0] = 1;
             *ncandidates = 1;
     }
     else    {
             *ncandidates = 0;
             last = a[k-1];
             p = g.edges[last];
             while (p != NULL) {
                     if (!in_sol[ p->y ]) {
                             c[*ncandidates] = p->y;
                             *ncandidates = *ncandidates + 1;
                     }
                     p = p->next;
             }
     }
}
```

We report a successful path whenever $a_k = t$.

```
is_a_solution(int a[], int k, int t)
{
        return (a[k] == t);
}

process_solution(int a[], int k)
{
        solution_count ++;        /* count all s to t paths */
}
```

The solution vector A must have room for all n vertices, although most paths are likely shorter than this. Figure 7.1 shows the search tree giving all paths from a particular vertex in an example graph.

7.2 Search Pruning

Backtracking ensures correctness by enumerating all possibilities. Enumerating all $n!$ permutations of n vertices of the graph and selecting the best one yields the correct algorithm to find the optimal traveling salesman tour. For each permutation, we could see whether all edges implied by the tour really exists in the graph G, and if so, add the weights of these edges together.

However, it would be wasteful to construct all the permutations first and then analyze them later. Suppose our search started from vertex v_1, and it happened that edge (v_1, v_2) was not in G. The next $(n-2)!$ permutations enumerated starting with (v_1, v_2) would be a complete waste of effort. Much better would be to prune the search after v_1, v_2 and continue next with v_1, v_3. By restricting the set of next elements to reflect only moves that are legal from the current partial configuration, we significantly reduce the search complexity.

Pruning is the technique of cutting off the search the instant we have established that a partial solution cannot be extended into a full solution. For traveling salesman, we seek the cheapest tour that visits all vertices. Suppose that in the course of our search we find a tour t whose cost is C_t. Later, we may have a partial solution a whose edge sum $C_A > C_t$. Is there any reason to continue exploring this node? No, because any tour with this prefix $a_1, \ldots, a_k$ will have cost greater than tour t, and hence is doomed to be nonoptimal. Cutting away such failed partial tours as soon as possible can have an enormous impact on running time.

Exploiting symmetry is another avenue for reducing combinatorial searches. Pruning away partial solutions identical to those previously considered requires recognizing underlying symmetries in the search space. For example, consider the state of our TSP search after we have tried all partial positions beginning with v_1. Does it pay to continue the search with partial solutions beginning with v_2? No. Any tour starting and ending at v_2 can be viewed as a rotation of one starting and ending at v_1, for these tours are cycles. There are thus only $(n-1)!$ distinct tours on n vertices, not $n!$. By restricting the first element of the tour to v_1, we save a factor of n in time without missing any interesting solutions. Detecting such symmetries can be subtle, but once identified they can usually be easily exploited.

Take-Home Lesson: Combinatorial searches, when augmented with tree pruning techniques, can be used to find the optimal solution of small optimization problems. How small depends upon the specific problem, but typical size limits are somewhere between $15 \leq n \leq 50$ items.

					1			2
			3		5			
		6					7	
7					3			
			4		8			
1								
			1	2				
		8					4	
		5				6		

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

Figure 7.2: Challenging Sudoku puzzle (l) with solution (r)

7.3 Sudoku

A Sudoku craze has swept the world. Many newspapers now publish daily Sudoku puzzles, and millions of books about Sudoku have been sold. British Airways sent a formal memo forbidding its cabin crews from doing Sudoku during takeoffs and landings. Indeed, I have noticed plenty of Sudoku going on in the back of my algorithms classes during lecture.

What is Sudoku? In its most common form, it consists of a 9×9 grid filled with blanks and the digits 1 to 9. The puzzle is completed when every row, column, and sector (3×3 subproblems corresponding to the nine sectors of a tic-tac-toe puzzle) contain the digits 1 through 9 with no deletions or repetition. Figure 7.2 presents a challenging Sudoku puzzle and its solution.

Backtracking lends itself nicely to the problem of solving Sudoku puzzles. We will use the puzzle here to better illustrate the algorithmic technique. Our state space will be the sequence of open squares, each of which must ultimately be filled in with a number. The candidates for open squares (i, j) are exactly the integers from 1 to 9 that have not yet appeared in row i, column j, or the 3×3 sector containing (i, j). We backtrack as soon as we are out of candidates for a square.

The solution vector `a` supported by `backtrack` only accepts a single integer per position. This is enough to store contents of a board square (1-9) but not the coordinates of the board square. Thus, we keep a separate array of `move` positions as part of our `board` data type provided below. The basic data structures we need to support our solution are:

```
#define DIMENSION 9                  /* 9*9 board */
#define NCELLS  DIMENSION*DIMENSION /* 81 cells in a 9*9 problem */

typedef struct {
    int x, y;                        /* x and y coordinates of point */
} point;
```

```
typedef struct {
    int m[DIMENSION+1][DIMENSION+1]; /* matrix of board contents */
    int freecount;                   /* how many open squares remain? */
    point move[NCELLS+1];            /* how did we fill the squares? */
} boardtype;
```

Constructing the candidates for the next solution position involves first picking the open square we want to fill next (`next_square`), and then identifying which numbers are candidates to fill that square (`possible_values`). These routines are basically bookkeeping, although the subtle details of how they work can have an enormous impact on performance.

```
construct_candidates(int a[], int k, boardtype *board, int c[],
                     int *ncandidates)
{
   int x,y;                     /* position of next move */
   int i;                       /* counter */
   bool possible[DIMENSION+1]; /* what is possible for the square */

   next_square(&x,&y,board);   /* which square should we fill next? */

   board->move[k].x = x;       /* store our choice of next position */
   board->move[k].y = y;

   *ncandidates = 0;

   if ((x<0) && (y<0)) return; /* error condition, no moves possible */

   possible_values(x,y,board,possible);
   for (i=0; i<=DIMENSION; i++)
           if (possible[i] == TRUE) {
                   c[*ncandidates] = i;
                   *ncandidates = *ncandidates + 1;
           }
}
```

We must update our `board` data structure to reflect the effect of filling a candidate value into a square, as well as remove these changes should we backtrack away from this position. These updates are handled by `make_move` and `unmake_move`, both of which are called directly from `backtrack`:

```
make_move(int a[], int k, boardtype *board)
{
        fill_square(board->move[k].x,board->move[k].y,a[k],board);
}


unmake_move(int a[], int k, boardtype *board)
{
        free_square(board->move[k].x,board->move[k].y,board);
}
```

One important job for these board update routines is maintaining how many free squares remain on the board. A solution is found when there are no more free squares remaining to be filled:

```
is_a_solution(int a[], int k, boardtype *board)
{
        if (board->freecount == 0)
                return (TRUE);
        else
                return(FALSE);
}
```

We print the configuration and turn off the backtrack search by setting off the global `finished` flag on finding a solution. This can be done without consequence because "official" Sudoku puzzles are only allowed to have one solution. There can be an enormous number of solutions to nonofficial Sudoku puzzles. Indeed, the empty puzzle (where no number is initially specified) can be filled in exactly 6,670,903,752,021,072,936,960 ways. We can ensure we don't see all of them by turning off the search:

```
process_solution(int a[], int k, boardtype *board)
{
        print_board(board);
        finished = TRUE;
}
```

This completes the program modulo details of identifying the next open square to fill (`next_square`) and identifying the candidates to fill that square (`possible_values`). Two reasonable ways to select the next square are:

- *Arbitrary Square Selection* – Pick the first open square we encounter, possibly picking the first, the last, or a random open square. All are equivalent in that there seems to be no reason to believe that one heuristic will perform any better than the other.

- *Most Constrained Square Selection* – Here, we check each of the open squares (i, j) to see how many number candidates remain for each—i.e. , have not already been used in either row i, column j, or the sector containing (i, j). We pick the square with the fewest number of candidates.

Although both possibilities work correctly, the second option is much, much better. Often there will be open squares with only one remaining candidate. For these, the choice is forced. We might as well make it now, especially since pinning this value down will help trim the possibilities for other open squares. Of course, we will spend more time selecting each candidate square, but if the puzzle is easy enough we may never have to backtrack at all.

If the most constrained square has two possibilities, we have a 1/2 probability of guessing right the first time, as opposed to a $(1/9)^{th}$ probability for a completely unconstrained square. Reducing our average number of choices from (say) 3 per square to 2 per square is an enormous win, since it multiplies for each position. If we have (say) 20 positions to fill, we must enumerate only $2^{20} = 1,048,576$ solutions. A branching factor of 3 at each of the 20 positions will result in over 3,000 times as much work!

Our final decision concerns the `possible_values` we allow for each square. We have two possibilities:

- *Local Count* – Our backtrack search works correctly if the routine generating candidates for board position (i, j) (`possible_values`) does the obvious thing and allows all numbers 1 to 9 that have not appeared in the given row, column, or sector.

- *Look ahead* – But what if our current partial solution has some *other* open square where there are no candidates remaining under the local count criteria? There is no possible way to complete this partial solution into a full Sudoku grid. Thus there *really* are zero possible moves to consider for (i, j) because of what is happening elsewhere on the board!

  We will discover this obstruction eventually, when we pick this square for expansion, discover it has no moves, and then have to backtrack. But why wait, since all our efforts until then will be wasted? We are *much* better off backtracking immediately and moving on.[1]

Successful pruning requires looking ahead to see when a solution is doomed to go nowhere, and backing off as soon as possible.

Table 7.1 presents the number of calls to `is_a_solution` for all four backtracking variants on three Sudoku instances of varying complexity:

[1]This look-ahead condition might have naturally followed from the most-constrained square selection, had it been permitted to select squares with no moves. However, my implementation credited squares that already contained numbers as having no moves, thus limiting the next square choices to squares with at least one move.

Pruning Condition		Puzzle Complexity		
`next_square`	`possible_values`	Easy	Medium	Hard
arbitrary	local count	1,904,832	863,305	never finished
arbitrary	look ahead	127	142	12,507,212
most constrained	local count	48	84	1,243,838
most constrained	look ahead	48	65	10,374

Table 7.1: Sudoku run times (in number of steps) under different pruning strategies

- The *Easy* board was intended to be easy for a human player. Indeed, my program solved it without any backtracking steps when the most constrained square was selected as the next position.
- The *Medium* board stumped all the contestants at the finals of the World Sudoku Championship in March 2006. The decent search variants still required only a few backtrack steps to dispatch this problem.
- The *Hard* problem is the board displayed in Figure 7.2, which contains only 17 fixed numbers. This is the fewest specified known number of positions in any problem instance that has only one complete solution.

What is considered to be a "hard" problem instance depends upon the given heuristic. Certainly you know people who find math/theory harder than programming and others who think differently. Heuristic A may well think instance I_1 is easier than I_2, while heuristic B ranks them in the other order.

What can we learn from these experiments? Looking ahead to eliminate dead positions as soon as possible is the best way to prune a search. Without this operation, we never finished the hardest puzzle and took thousands of times longer than we should have on the easier ones.

Smart square selection had a similar impact, even though it nominally just rearranges the order in which we do the work. However, doing more constrained positions first is tantamount to reducing the outdegree of each node in the tree, and each additional position we fix adds constraints that help lower the degree for future selections.

It took the better part of an hour (48:44) to solve the puzzle in Figure 7.2 when I selected an arbitrary square for my next move. Sure, my program was faster in most instances, but Sudoku puzzles are designed to be solved by people using pencils in much less time than this. Making the next move in the most constricted square reduced search time by a factor of over 1,200. Each puzzle we tried can now be solved in seconds—the time it takes to reach for the pencil if you want to do it by hand.

This is the power of a pruning search. Even simple pruning strategies can suffice to reduce running time from impossible to instantaneous.

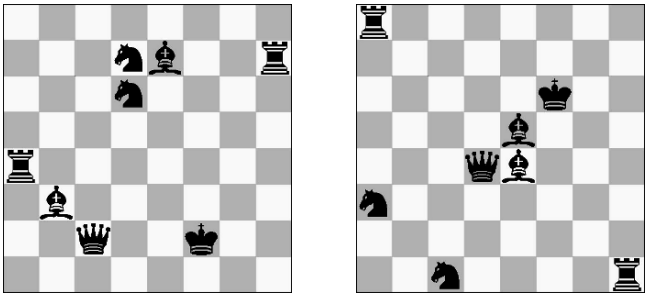


Figure 7.3: Configurations covering 63 but not 64 squares

7.4 War Story: Covering Chessboards

Every researcher dreams of solving a classical problem—one that has remained open and unsolved for over a century. There is something romantic about communicating across the generations, being part of the evolution of science, and helping to climb another rung up the ladder of human progress. There is also a pleasant sense of smugness that comes from figuring out how to do something that nobody could do before you.

There are several possible reasons why a problem might stay open for such a long period of time. Perhaps it is so difficult and profound that it requires a uniquely powerful intellect to solve. A second reason is technological—the ideas or techniques required to solve the problem may not have existed when it was first posed. A final possibility is that no one may have cared enough about the problem in the interim to seriously bother with it. Once, I helped solve a problem that had been open for over a hundred years. Decide for yourself which reason best explains why.

Chess is a game that has fascinated mankind for thousands of years. In addition, it has inspired many combinatorial problems of independent interest. The combinatorial explosion was first recognized with the legend that the inventor of chess demanded as payment one grain of rice for the first square of the board, and twice as much for the $(i+1)$st square than the ith square. The king was astonished to learn he had to cough up $\sum_{i=1}^{64} 2^i = 2^{65} - 1 = 36{,}893{,}488{,}147{,}419{,}103{,}231$ grains of rice. In beheading the inventor, the wise king first established pruning as a technique for dealing with the combinatorial explosion.

In 1849, Kling posed the question of whether all 64 squares on the board can be simultaneously threatened by an arrangement of the eight main pieces on the chess board—the king, queen, two knights, two rooks, and two oppositely colored bishops. Pieces do not threaten the square they sit on. Configurations that simultaneously threaten 63 squares, such as those in Figure 7.3, have been known for a long time, but whether this was the best possible remained an open problem. This problem

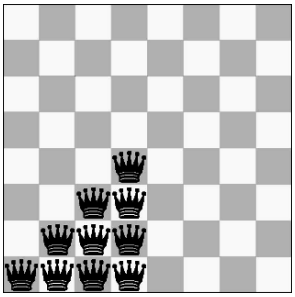


Figure 7.4: The ten unique positions for the queen, with respect to symmetry

seemed ripe for solution by exhaustive combinatorial searching, although whether it was solvable depended upon the size of the search space.

Consider the eight main pieces in chess (king, queen, two rooks, two bishops, and two knights). How many ways can they be positioned on a chessboard? The trivial bound is $64!/(64-8)! = 178{,}462{,}987{,}637{,}760 \approx 10^{15}$ positions. Anything much larger than about 10^9 positions would be unreasonable to search on a modest computer in a modest amount of time.

Getting the job done would require significant pruning. Our first idea was to remove symmetries. Accounting for orthogonal and diagonal symmetries left only ten distinct positions for the queen, shown in Figure 7.4.

Once the queen is placed, there are $64 \cdot 63/2 = 2{,}016$ distinct ways to position a pair of rooks or knights, 64 places to locate the king, and 32 spots for each of the white and black bishops. Such an exhaustive search would test $2{,}663{,}550{,}812{,}160 \approx 10^{13}$ distinct positions—still much too large to try.

We could use backtracking to construct all of the positions, but we had to find a way to prune the search space significantly. Pruning the search meant that we needed a quick way to prove that there was no way to complete a partially filled-in position to cover all 64 squares. Suppose we had already placed seven pieces on the board, and together they covered all but 10 squares of the board. Say the remaining piece was the king. Can there be a position to place the king so that all squares are threatened? The answer must be no, because the king can threaten at most eight squares according to the rules of chess. There can be no reason to test any king position. We might win big pruning such configurations.

This pruning strategy required carefully ordering the evaluation of the pieces. Each piece can threaten a certain maximum number of squares: the queen 27, the king/knight 8, the rook 14, and the bishop 13. We would want to insert the pieces in decreasing order of mobility. We can prune when the number of unthreatened squares exceeds the sum of the maximum coverage of the unplaced pieces. This sum is minimized by using the decreasing order of mobility.

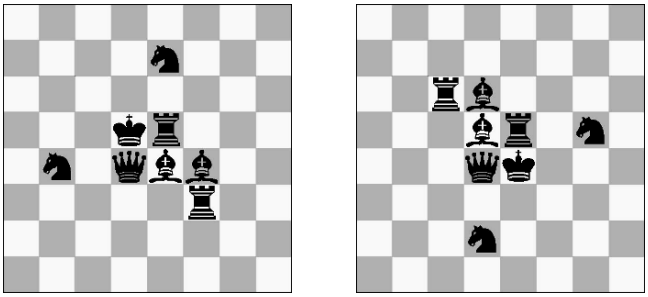


Figure 7.5: Weakly covering 64 squares

When we implemented a backtrack search using this pruning strategy, we found that it eliminated over 95% of the search space. After optimizing our move generation, our program could search over 1,000 positions per second. But this was still too slow, for $10^{12}/10^3 = 10^9$ seconds meant 1,000 days! Although we might further tweak the program to speed it up by an order of magnitude or so, what we really needed was to find a way to prune more nodes.

Effective pruning meant eliminating large numbers of positions at a single stroke. Our previous attempt was too weak. What if instead of placing up to eight pieces on the board simultaneously, we placed *more* than eight pieces. Obviously, the more pieces we placed simultaneously, the more likely they would threaten all 64 squares. But *if* they didn't cover, all subsets of eight distinct pieces from the set couldn't possibly threaten all squares. The potential existed to eliminate a vast number of positions by pruning a single node.

So in our final version, the nodes of our search tree corresponded to chessboards that could have any number of pieces, and more than one piece on a square. For a given board, we distinguished *strong* and *weak* attacks on a square. A strong attack corresponds to the usual notion of a threat in chess. A square is *weakly attacked* if the square is strongly attacked by some subset of the board—that is, a weak attack ignores any possible blocking effects of intervening pieces. All 64 squares can be weakly attacked with eight pieces, as shown in Figure 7.5.

Our algorithm consisted of two passes. The first pass listed boards where every square was weakly attacked. The second pass filtered the list by considering blocking pieces. A weak attack is much faster to compute (no blocking to worry about), and any strong attack set is always a subset of a weak attack set. The position could be pruned whenever there was a non-weakly threatened square.

This program was efficient enough to complete the search on a slow 1988-era IBM PC-RT in under one day. It did not find a single position covering all 64 squares with the bishops on opposite colored squares. However, our program showed that it is possible to cover the board with *seven* pieces provided a queen and a knight can occupy the same square, as shown in Figure 7.6.

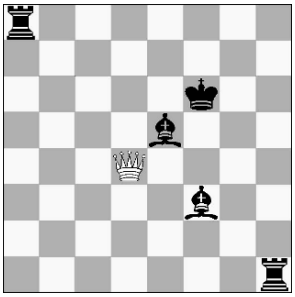


Figure 7.6: Seven pieces suffice when superimposing queen and knight (shown as a white queen)

Take-Home Lesson: Clever pruning can make short work of surprisingly hard combinatorial search problems. Proper pruning will have a greater impact on search time than any other factor.

7.5 Heuristic Search Methods

Heuristic methods provide an alternate way to approach difficult combinatorial optimization problems. Backtracking gave us a method to find the best of all possible solutions, as scored by a given objective function. However, any algorithm searching all configurations is doomed to be impossible on large instances.

In this section, we discuss such heuristic search methods. We devote the bulk of our attention to simulated annealing, which I find to be the most reliable method to apply in practice. Heuristic search algorithms have an air of voodoo about them, but how they work and why one method might work better than another follows logically enough if you think them through.

In particular, we will look at three different heuristic search methods: random sampling, gradient-descent search, and simulated annealing. The traveling salesman problem will be our ongoing example for comparing heuristics. All three methods have two common components:

- *Solution space representation* – This is a complete yet concise description of the set of possible solutions for the problem. For traveling salesman, the solution space consists of $(n-1)!$ elements—namely all possible circular permutations of the vertices. We need a data structure to represent each element of the solution space. For TSP, the candidate solutions can naturally be represented using an array S of $n-1$ vertices, where S_i defines the $(i+1)$st vertex on the tour starting from v_1.
- *Cost function* – Search methods need a *cost* or *evaluation* function to access the quality of each element of the solution space. Our search heuristic

identifies the element with the best possible score—either highest or lowest depending upon the nature of the problem. For TSP, the cost function for evaluating a given candidate solution S should just sum up the costs involved, namely the weight of all edges (S_i, S_{i+1}), where S_{n+1} denotes v_1.

7.5.1 Random Sampling

The simplest method to search in a solution space uses random sampling. It is also called the *Monte Carlo method*. We repeatedly construct random solutions and evaluate them, stopping as soon as we get a good enough solution, or (more likely) when we are tired of waiting. We report the best solution found over the course of our sampling.

True random sampling requires that we are able to select elements from the solution space *uniformly at random*. This means that each of the elements of the solution space must have an equal probability of being the next candidate selected. Such sampling can be a subtle problem. Algorithms for generating random permutations, subsets, partitions, and graphs are discussed in Sections 14.4-14.7.

```
random_sampling(tsp_instance *t, int nsamples, tsp_solution *bestsol)
{
        tsp_solution s;                 /* current tsp solution */
        double best_cost;               /* best cost so far */
        double cost_now;                /* current cost */
        int i;                          /* counter */

        initialize_solution(t->n,&s);
        best_cost = solution_cost(&s,t);
        copy_solution(&s,bestsol);

        for (i=1; i<=nsamples; i++) {
                random_solution(&s);
                cost_now = solution_cost(&s,t);
                if (cost_now < best_cost) {
                        best_cost = cost_now;
                        copy_solution(&s,bestsol);
                }
        }
}
```

When might random sampling do well?

- *When there are a high proportion of acceptable solutions* – Finding a piece of hay in a haystack is easy, since almost anything you grab is a straw. When solutions are plentiful, a random search should find one quickly.

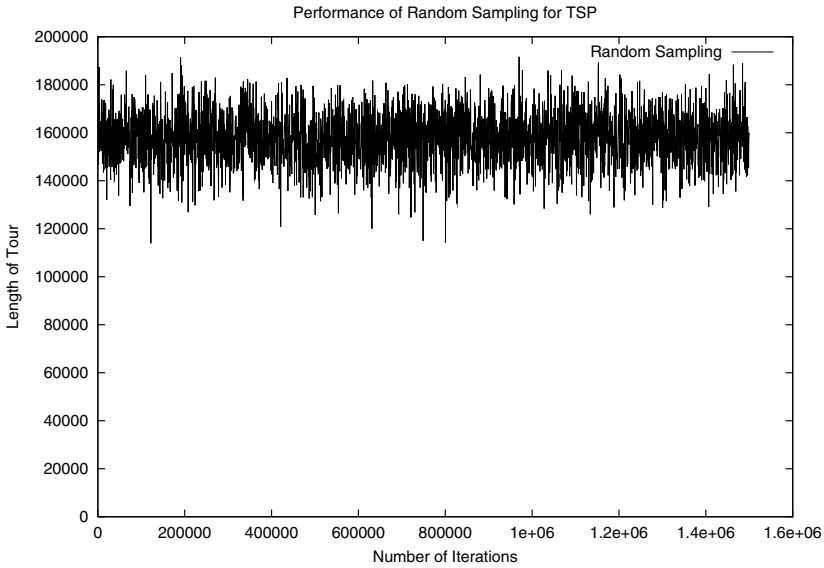


Figure 7.7: Search time/quality tradeoffs for TSP using random sampling.

Finding prime numbers is domain where a random search proves successful. Generating large random prime numbers for keys is an important aspect of cryptographic systems such as RSA. Roughly one out of every $\ln n$ integers are prime, so only a modest number of samples need to be taken to discover primes that are several hundred digits long.

- *When there is no coherence in the solution space* – Random sampling is the right thing to do when there is no sense of when we are getting *closer* to a solution. Suppose you wanted to find one of your friends who has a social security number that ends in 00. There is not much you can hope to do but tap an arbitrary fellow on their shoulder and ask. No cleverer method will be better than random sampling.

  Consider again the problem of hunting for a large prime number. Primes are scattered quite arbitrarily among the integers. Random sampling is as good as anything else.

How does random sampling do on TSP? Pretty lousy. The best solution I found after testing 1.5 million random permutations of a classic TSP instance (the capital cities of the 48 continental United States) was 101,712.8. This is more than three times the cost of the optimal tour! The solution space consists almost entirely

of mediocre to bad solutions, so quality grows very slowly with the amount of sampling / running time we invest. Figure 7.7 presents the arbitrary up-and-down movements of random sampling and generally poor quality solutions encountered on the journey, so you can get a sense of how the score varied over each iteration.

Most problems we encounter, like TSP, have relatively few good solutions but a highly coherent solution space. More powerful heuristic search algorithms are required to deal effectively with such problems.

Stop and Think: Picking the Pair

Problem: We need an efficient and unbiased way to generate random pairs of vertices to perform random vertex swaps. Propose an efficient algorithm to generate elements from the $\binom{n}{2}$ *unordered* pairs on $\{1, \ldots, n\}$ uniformly at random.

Solution: Uniformly generating random structures is a surprisingly subtle problem. Consider the following procedure to generate random unordered pairs:

```
i = random_int(1,n-1);
j = random_int(i+1,n);
```

It is clear that this indeed generates unordered pairs, since $i < j$. Further, it is clear that all $\binom{n}{2}$ unordered pairs can indeed be generated, assuming that `random_int` generates integers uniformly between its two arguments.

But are they uniform? The answer is no. What is the probability that pair $(1, 2)$ is generated? There is a $1/(n-1)$ chance of getting the 1, and then a $1/(n-1)$ chance of getting the 2, which yields $p(1, 2) = 1/(n-1)^2$. But what is the probability of getting $(n-1, n)$? Again, there is a $1/n$ chance of getting the first number, but now there is only one possible choice for the second candidate! This pair will occur n times more often than the first!

The problem is that fewer pairs start with big numbers than little numbers. We could solve this problem by calculating exactly how unordered pairs start with i (exactly $(n-i)$) and appropriately bias the probability. The second value could then be selected uniformly at random from $i+1$ to n.

But instead of working through the math, let's exploit the fact that randomly generating the n^2 *ordered* pairs uniformly is easy. Just pick two integers independently of each other. Ignoring the ordering (i.e. , permuting the ordered pair to unordered pair (x, y) so that $x < y$) gives us a $2/n^2$ probability of generating each unordered pair of distinct elements. If we happen to generate a pair (x, x), we discard it and try again. We will get unordered pairs uniformly at random in constant expected time using the following algorithm:

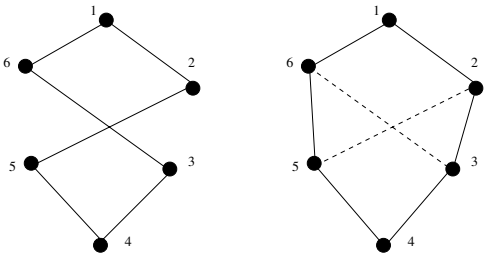


Figure 7.8: Improving a TSP tour by swapping vertices 2 and 6

```
do {
     i = random_int(1,n);
     j = random_int(1,n);
     if (i > j) swap(&i,&j);
} while (i==j);
```

■

7.5.2 Local Search

Now suppose you want to hire an algorithms expert as a consultant to solve your problem. You *could* dial a phone number at random, ask if they are an algorithms expert, and hang up the phone if they say no. After many repetitions you will probably find one, but it would probably be more efficient to ask the fellow on the phone for someone more likely to know an algorithms expert, and call *them* up instead.

A local search employs *local neighborhood* around every element in the solution space. Think of each element x in the solution space as a vertex, with a directed edge (x, y) to every candidate solution y that is a neighbor of x. Our search proceeds from x to the most promising candidate in x's neighborhood.

We certainly do *not* want to explicitly construct this neighborhood graph for any sizable solution space. Think about TSP, which will have $(n-1)!$ vertices in this graph. We are conducting a heuristic search precisely because we cannot hope to do this many operations in a reasonable amount of time.

Instead, we want a general transition mechanism that takes us to the next solution by slightly modifying the current one. Typical transition mechanisms include swapping a random pair of items or changing (inserting or deleting) a single item in the solution.

The most obvious transition mechanism for TSP would be to swap the current tour positions of a random pair of vertices S_i and S_j, as shown in Figure 7.8.

This changes up to eight edges on the tour, deleting the edges currently adjacent to both S_i and S_j, and adding their replacements. Ideally, the effect that these incremental changes have on measuring the quality of the solution can be computed incrementally, so cost function evaluation takes time proportional to the size of the change (typically constant) instead of linear to the size of the solution.

A local search heuristic starts from an arbitrary element of the solution space, and then scans the neighborhood looking for a favorable transition to take. For TSP, this would be *transition*, which lowers the cost of the tour. In a *hill-climbing* procedure, we try to find the top of a mountain (or alternately, the lowest point in a ditch) by starting at some arbitrary point and taking any step that leads in the direction we want to travel. We repeat until we have reached a point where all our neighbors lead us in the wrong direction. We are now *King of the Hill* (or *Dean of the Ditch*).

We are probably not *King of the Mountain*, however. Suppose you wake up in a ski lodge, eager to reach the top of the neighboring peak. Your first transition to gain altitude might be to go upstairs to the top of the building. And then you are trapped. To reach the top of the mountain, you must go downstairs and walk outside, but this violates the requirement that each step has to increase your score. Hill-climbing and closely related heuristics such as greedy search or gradient descent search are great at finding local optima quickly, but often fail to find the globally best solution.

```
hill_climbing(tsp_instance *t, tsp_solution *s)
{
     double cost;                /* best cost so far */
     double delta;               /* swap cost */
     int i,j;                    /* counters */
     bool stuck;                 /* did I get a better solution? */
     double transition();

     initialize_solution(t->n,s);
     random_solution(s);
     cost = solution_cost(s,t);

     do {
             stuck = TRUE;
             for (i=1; i<t->n; i++)
                     for (j=i+1; j<=t->n; j++) {
                             delta = transition(s,t,i,j);
                             if (delta < 0) {
                                     stuck = FALSE;
                                     cost = cost + delta;
                             }
```

```
                        else
                                transition(s,t,j,i);
                    }
    } while (!stuck);
}
```

When does local search do well?

- *When there is great coherence in the solution space* – Hill climbing is at its best when the solution space is *convex*. In other words, it consists of exactly one hill. No matter where you start on the hill, there is always a direction to walk up until you are at the absolute global maximum.

  Many natural problems do have this property. We can think of a binary search as starting in the middle of a search space, where exactly one of the two possible directions we can walk will get us closer to the target key. The simplex algorithm for linear programming (see Section 13.6 (page 411)) is nothing more than hill-climbing over the right solution space, yet guarantees us the optimal solution to any linear programming problem.

- *Whenever the cost of incremental evaluation is much cheaper than global evaluation* – It costs $\Theta(n)$ to evaluate the cost of an arbitrary n-vertex candidate TSP solution, because we must total the cost of each edge in the circular permutation describing the tour. Once that is found, however, the cost of the tour after swapping a given pair of vertices can be determined in constant time.

  If we are given a very large value of n and a very small budget of how much time we can spend searching, we are better off using it to do several incremental evaluations than a few random samples, even if we are looking for a needle in a haystack.

The primary drawback of a local search is that soon there isn't anything left for us to do as we find the local optimum. Sure, if we have more time we could start from different random points, but in landscapes of many low hills we are unlikely to stumble on the optimum.

How does local search do on TSP? Much better than random sampling for a similar amount of time. With over a total of 1.5 million tour evaluations in our 48-city TSP instance, our best local search tour had a length of 40,121.2—only 19.6% more than the optimal tour of 33,523.7.

This is good, but not great. You would not be happy to learn you are paying 19.6% more taxes than you should. Figure 7.9 illustrates the trajectory of a local search: repeated streaks from random tours down to decent solutions of fairly similar quality. We need more powerful methods to get closer to the optimal solution.

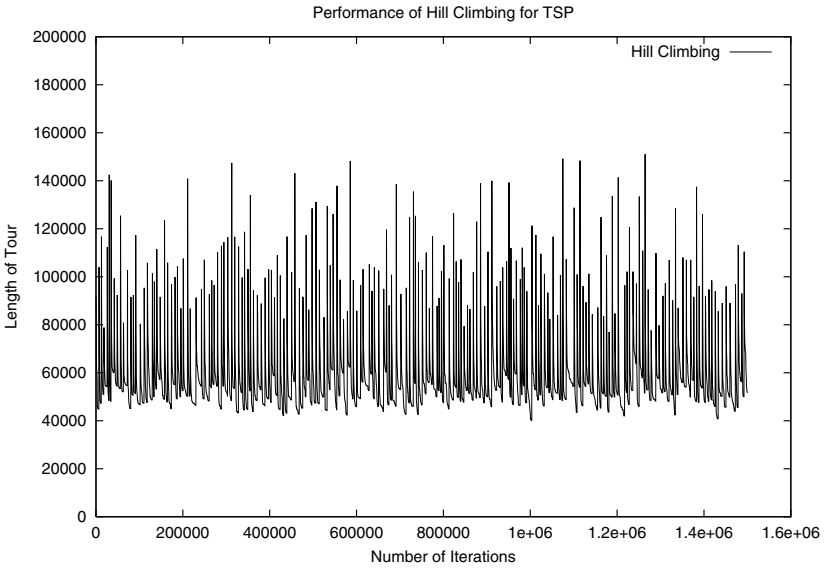


Figure 7.9: Search time/quality tradeoffs for TSP using hill climbing.

7.5.3 Simulated Annealing

Simulated annealing is a heuristic search procedure that allows occasional transitions leading to more expensive (and hence inferior) solutions. This may not sound like progress, but it helps keep our search from getting stuck in local optima. That poor fellow trapped on the top floor of a ski lodge would do better to break the glass and jump out the window if he really wanted to reach the top of the mountain.

The inspiration for simulated annealing comes from the physical process of cooling molten materials down to the solid state. In thermodynamic theory, the energy state of a system is described by the energy state of each particle constituting it. A particle's energy state jumps about randomly, with such transitions governed by the temperature of the system. In particular, the transition probability $P(e_i, e_j, T)$ from energy e_i to e_j at temperature T is given by

$$P(e_i, e_j, T) = e^{(e_i - e_j)/(k_B T)}$$

where k_B is a constant—called Boltzmann's constant.

What does this formula mean? Consider the value of the exponent under different conditions. The probability of moving from a high-energy state to a lower-energy state is very high. But, there is still a nonzero probability of accepting a

transition into a high-energy state, with such small jumps much more likely than big ones. The higher the temperature, the more likely energy jumps will occur.

```
Simulated-Annealing()
    Create initial solution S
    Initialize temperature t
    repeat
        for i = 1 to iteration-length do
            Generate a random transition from S to S_i
            If (C(S) ≥ C(S_i)) then S = S_i
            else if (e^((C(S)−C(S_i))/(k·t)) > random[0,1)) then S = S_i
        Reduce temperature t
    until (no change in C(S))
    Return S
```

What relevance does this have for combinatorial optimization? A physical system, as it cools, seeks to reach a minimum-energy state. Minimizing the total energy is a combinatorial optimization problem for any set of discrete particles. Through random transitions generated according to the given probability distribution, we can mimic the physics to solve arbitrary combinatorial optimization problems.

Take-Home Lesson: Forget about this molten metal business. Simulated annealing is effective because it spends much more of its time working on good elements of the solution space than on bad ones, and because it avoids getting trapped repeatedly in the same local optima.

As with a local search, the problem representation includes both a representation of the solution space and an easily computable cost function $C(s)$ measuring the quality of a given solution. The new component is the *cooling schedule*, whose parameters govern how likely we are to accept a bad transition as a function of time.

At the beginning of the search, we are eager to use randomness to explore the search space widely, so the probability of accepting a negative transition should be high. As the search progresses, we seek to limit transitions to local improvements and optimizations. The cooling schedule can be regulated by the following parameters:

- *Initial system temperature* – Typically $t_1 = 1$.
- *Temperature decrement function* – Typically $t_k = \alpha \cdot t_{k-1}$, where $0.8 \leq \alpha \leq 0.99$. This implies an exponential decay in the temperature, as opposed to a linear decay.
- *Number of iterations between temperature change* – Typically, 100 to 1,000 iterations might be permitted before lowering the temperature.

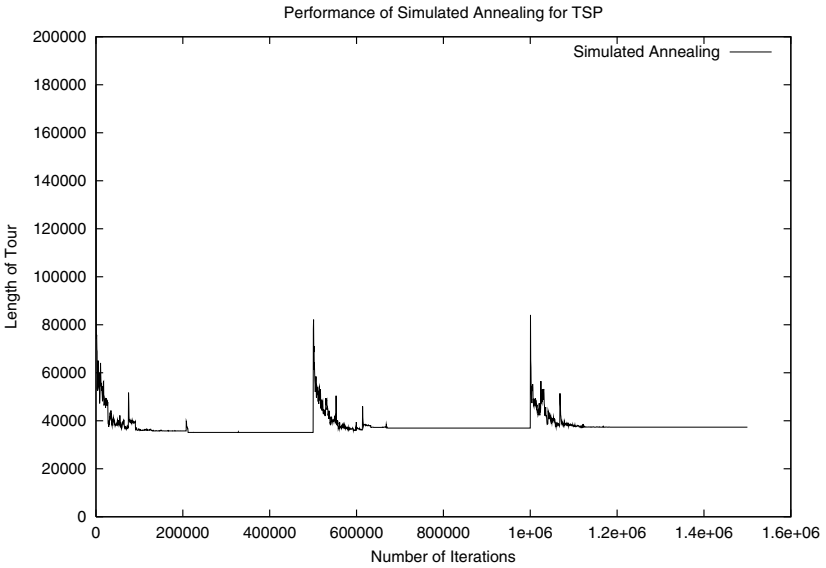


Figure 7.10: Search time/quality tradeoffs for TSP using simulated annealing

- *Acceptance criteria* – A typical criterion is to accept any transition from s_i to s_{i+1} when $C(s_{i+1}) < C(s_i)$, and also accept a negative transition whenever

  $$e^{-\frac{(C(s_i)-C(s_{i+1}))}{k \cdot t_i}} \geq r,$$

  where r is a random number $0 \leq r < 1$. The constant k normalizes this cost function so that almost all transitions are accepted at the starting temperature.

- *Stop criteria* – Typically, when the value of the current solution has not changed or improved within the last iteration or so, the search is terminated and the current solution reported.

Creating the proper cooling schedule is somewhat of a trial-and-error process of mucking with constants and seeing what happens. It probably pays to start from an existing implementation of simulated annealing, so check out my full implementation (at *http://www.algorist.com*) as well as others provided in Section 13.5 (page 407).

Compare the search/time execution profiles of our three heuristics. The cloud of points corresponding to random sampling is significantly worse than the solutions

encountered by the other heuristics. The scores of the short streaks corresponding to runs of hill-climbing solutions are clearly much better.

But best of all are the profiles of the three simulated annealing runs in Figure 7.10. All three runs lead to much better solutions than the best hill-climbing result. Further, it takes relatively few iterations to score most of the improvement, as shown by the three rapid plunges toward optimum we see with simulated annealing.

Using the same 1,500,000 iterations as the other methods, simulated annealing gave us a solution of cost 36,617.4—only 9.2% over the optimum. Even better solutions are available to those willing to wait a few minutes. Letting it run for 5,000,000 iterations got the score down to 34,254.9, or 2.2% over the optimum. There were no further improvements after I cranked it up to 10,000,000 iterations.

In expert hands, the best problem-specific heuristics for TSP can slightly outperform simulated annealing. But the simulated annealing solution works admirably. It is my heuristic method of choice.

Implementation

The implementation follows the pseudocode quite closely:

```
anneal(tsp_instance *t, tsp_solution *s)
{
  int i1, i2;                  /* pair of items to swap */
  int i,j;                     /* counters */
  double temperature;          /* the current system temp */
  double current_value;        /* value of current state */
  double start_value;          /* value at start of loop */
  double delta;                /* value after swap */
  double merit, flip;          /* hold swap accept conditions*/
  double exponent;             /* exponent for energy funct*/
  double random_float();
  double solution_cost(), transition();

  temperature = INITIAL_TEMPERATURE;

  initialize_solution(t->n,s);
  current_value = solution_cost(s,t);

  for (i=1; i<=COOLING_STEPS; i++) {
          temperature *= COOLING_FRACTION;

          start_value = current_value;

          for (j=1; j<=STEPS_PER_TEMP; j++) {
```

```
            /* pick indices of elements to swap */
            i1 = random_int(1,t->n);
            i2 = random_int(1,t->n);

            flip = random_float(0,1);

            delta = transition(s,t,i1,i2);
            exponent = (-delta/current_value)/(K*temperature);
            merit = pow(E,exponent);

            if (delta < 0)                              /*ACCEPT-WIN*/
                 current_value = current_value+delta;
            else { if (merit > flip)                    /*ACCEPT-LOSS*/
                 current_value = current_value+delta;
            else                                        /* REJECT */
                 transition(s,t,i1,i2);
            }
          }

          /* restore temperature if progress has been made */
          if ((current_value-start_value) < 0.0)
                 temperature = temperature/COOLING_FRACTION;
  }
}
```

7.5.4 Applications of Simulated Annealing

We provide several examples to demonstrate how these components can lead to elegant simulated annealing solutions for real combinatorial search problems.

Maximum Cut

The "maximum cut" problem seeks to partition the vertices of a weighted graph G into sets V_1 and V_2 to maximize the weight (or number) of edges with one vertex in each set. For graphs that specify an electronic circuit, the maximum cut in the graph defines the largest amount of data communication that can take place in the circuit simultaneously. As discussed in Section 16.6 (page 541), maximum cut is NP-complete.

How can we formulate maximum cut for simulated annealing? The solution space consists of all 2^{n-1} possible vertex partitions. We save a factor of two over all vertex subsets because vertex v_1 can be assumed to be fixed on the left side of the partition. The subset of vertices accompanying it can be represented using

a bit vector. The cost of a solution is the sum of the weights cut in the current configuration. A natural transition mechanism selects one vertex at random and moves it across the partition simply by flipping the corresponding bit in the bit vector. The change in the cost function will be the weight of its old neighbors minus the weight of its new neighbors. This can be computed in time proportional to the degree of the vertex.

This kind of simple, natural modeling is the right type of heuristic to seek in practice.

Independent Set

An "independent set" of a graph G is a subset of vertices S such that there is no edge with both endpoints in S. The maximum independent set of a graph is the largest such empty induced subgraph. Finding large independent sets arises in dispersion problems associated with facility location and coding theory, as discussed in Section 16.2 (page 528).

The natural state space for a simulated annealing solution would be all 2^n subsets of the vertices, represented as a bit vector. As with maximum cut, a simple transition mechanism would add or delete one vertex from S.

One natural cost function for subset S might be 0 if S contains an edge, and $|S|$ if it is indeed an independent set. This function ensures that we work towards an independent set at all times. However, this condition is strict enough that we are liable to move only in a narrow portion of the possible search space. More flexibility in the search space and quicker cost function computations can result from allowing nonempty graphs at the early stages of cooling. Better in practice would be a cost function like $C(S) = |S| - \lambda \cdot m_S/T$, where λ is a constant, T is the temperature, and m_S is the number of edges in the subgraph induced by S. The dependence of $C(S)$ on T ensures that the search will drive the edges out faster as the system cools.

Circuit Board Placement

In designing printed circuit boards, we are faced with the problem of positioning modules (typically, integrated circuits) on the board. Desired criteria in a layout may include (1) minimizing the area or aspect ratio of the board so that it properly fits within the allotted space, and (2) minimizing the total or longest wire length in connecting the components. Circuit board placement is representative of the kind of messy, multicriterion optimization problems for which simulated annealing is ideally suited.

Formally, we are given a collection of rectangular modules $r_1, \ldots, r_n$, each with associated dimensions $h_i \times l_i$. Further, for each pair of modules r_i, r_j, we are given the number of wires w_{ij} that must connect the two modules. We seek a placement of the rectangles that minimizes area and wire length, subject to the constraint that no two rectangles overlap each other.

The state space for this problem must describe the positions of each rectangle. To make this discrete, the rectangles can be restricted to lie on vertices of an integer grid. Reasonable transition mechanisms including moving one rectangle to a different location, or swapping the position of two rectangles. A natural cost function would be

$$C(S) = \lambda_{area}(S_{height} \cdot S_{width}) + \sum_{i=1}^{n}\sum_{j=1}^{n}(\lambda_{wire} \cdot w_{ij} \cdot d_{ij} + \lambda_{overlap}(r_i \cap r_j))$$

where λ_{area}, λ_{wire}, and $\lambda_{overlap}$ are constants governing the impact of these components on the cost function. Presumably, $\lambda_{overlap}$ should be an inverse function of temperature, so after gross placement it adjusts the rectangle positions to be disjointed.

Take-Home Lesson: Simulated annealing is a simple but effective technique for efficiently obtaining good but not optimal solutions to combinatorial search problems.

7.6 War Story: Only it is Not a Radio

"Think of it as a radio," he chuckled. "Only it is not a radio."

I'd been whisked by corporate jet to the research center of a large but very secretive company located somewhere east of California. They were so paranoid that I never got to see the object we were working on, but the people who brought me in did a great job of abstracting the problem.

The application concerned a manufacturing technique known as *selective assembly*. Eli Whitney started the Industrial Revolution through his system of *interchangeable parts*. He carefully specified the manufacturing tolerances on each part in his machine so that the parts were *interchangeable*, meaning that any legal cog-widget could be used to replace any other legal cog-widget. This greatly sped up the process of manufacturing, because the workers could just put parts together instead of having to stop to file down rough edges and the like. It made replacing broken parts a snap. This was a very good thing.

Unfortunately, it also resulted in large piles of cog-widgets that were slightly outside the manufacturing tolerance and thus had to be discarded. Another clever fellow then observed that maybe one of these defective cog-widgets could be used when all the *other* parts in the given assembly *exceeded* their required manufacturing tolerances. Good plus bad could well equal good enough. This is the idea of *selective assembly*.

"Each not-radio is made up of n different types of not-radio parts," he told me. For the ith part type (say the right flange gasket), we have a pile of s_i instances of this part type. Each part (flange gasket) comes with a measure of the degree to which it deviates from perfection. We need to match up the parts so as to create the greatest number of working not-radios as possible."

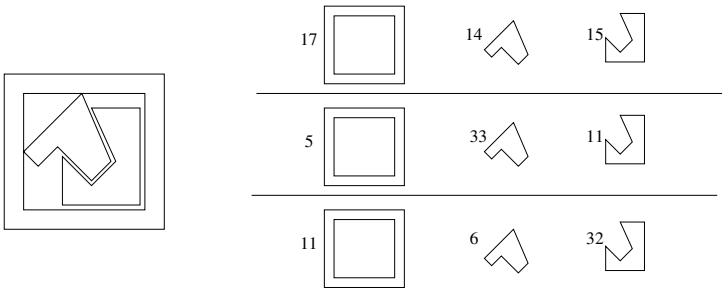


Figure 7.11: Part assignments for three not-radios, such that each had at most 50 points total defect

The situation is illustrated in Figure 7.11. Each not-radio consists of three parts, and the sum of the defects in any functional not-radio must total at most 50. By cleverly balancing the good and bad parts in each machine, we can use all the parts and make three working not-radios.

I thought about the problem. The simplest procedure would take the best part for each part type, make a not-radio out of them, and repeat until the not-radio didn't play (or do whatever a not-radio does). But this would create a small number of not-radios drastically varying in quality, whereas they wanted as many decent not-radios as possible.

The goal was to match up good parts and bad parts so the total amount of badness wasn't so bad. Indeed, the problem sounded related to *matching* in graphs (see Section 15.6 (page 498)). Suppose we built a graph where the vertices were the part instances, and add an edge for all two part instances that were within the total error tolerance. In graph matching, we seek the largest number of edges such that no vertex appears more than once in the matching. This is analogous to the largest number of two-part assemblies we can form from the given set of parts.

"I can solve your problem using matching," I announced, "Provided not-radios are each made of only two parts."

There was silence. Then they all started laughing at me. "*Everyone* knows not-radios have more than two parts," they said, shaking their heads.

That spelled the end of this algorithmic approach. Extending to more than two parts turned the problem into matching on hypergraphs,[2]—a problem which is NP-complete. Further, it might take exponential time in the number of part types just to build the graph, since we had to explicitly construct each possible hyperedge/assembly.

[2]A *hypergraph* is made up of edges that can contain more than two vertices each. They can be thought of as general collections of subsets of vertices/elements.

I went back to the drawing board. They wanted to put parts into assemblies so that no assembly would have more total defects than allowed. Described that way, it sounded like a packing problem. In the *bin packing* problem (see Section 17.9 (page 595)), we are given a collection of items of different sizes and asked to store them using the smallest possible number of bins, each of which has a fixed capacity of size k. Here, the assemblies represented the bins, each of which could absorb total defect $\leq k$. The items to pack represented the individual parts, whose size would reflect its quality of manufacture.

It wasn't pure bin packing, however, since parts came in different types. The application imposed constraints on the allowable contents of each bin. Creating the maximum number of not-radios meant that we sought a packing that maximized the number of bins which contained exactly one part for each of the m different parts types.

Bin packing is NP-complete, but is a natural candidate for a heuristic search approach. The solution space consists of assignments of parts to bins. We initially pick a random part of each type for each bin to give us a starting configuration for the search.

The local neighborhood operation involves moving parts around from one bin to another. We could move one part at a time, but more effective was *swapping* parts of a particular type between two randomly chosen bins. In such a swap, both bins remain complete not-radios, hopefully with better error tolerance than before. Thus, our swap operator required three random integers—one to select the appropriate parts type (from 1 to m) and two more to select the assembly bins involved (say from 1 to b).

The key decision was the cost function to use. They supplied the hard limit k on the total defect level for each *individual* assembly. But what was the best way to score a *set* of assemblies? We could just return the number of acceptable complete assemblies as our score—an integer from 1 to b. Although this was indeed what we wanted to optimize, it would not be sensitive to detect when we were making partial progress towards a solution. Suppose one of our swaps succeeded in bringing one of the nonfunctional assemblies much closer to the not-radio limit k. That would be a better starting point for further progress than the original, and should be favored.

My final cost function was as follows. I gave one point for every working assembly, and a significantly smaller total for each nonworking assembly based on how close it was to the threshold k. The score for a nonworking assembly decreased exponentially based on how much it was over k. Thus the optimizer would seek to maximize the number of working assemblies, and then try to drive another assembly close to the limit.

I implemented this algorithm, and then ran the search on the test case they provided. It was an instance taken directly from the factory floor. Not-radios turn out to contain $m = 8$ important parts types. Some parts types are more expensive than others, and so they have fewer available candidates to consider. The most

prefix \ suffix	AA	AG	GA	GG
AA	$AAAA$	$AAAG$	$AAGA$	$AAGG$
AG	$AGAA$	$AGAG$	$AGGA$	$AGGG$
GA	$GAAG$	$GAAG$	$GAGA$	$GAGG$
GG	$GGAA$	$GGAG$	$GGGA$	$GGGG$

Figure 7.12: A prefix-suffix array of all purine 4-mers

constrained parts type had only eight representatives, so there could be at most eight possible assemblies from this given mix.

I watched as simulated annealing chugged and bubbled on the problem instance. The number of completed assemblies instantly climbed (1, 2, 3, 4) before progress started to slow a bit. Then came 5 and 6 in a hiccup, with a pause before assembly 7 came triumphantly together. But tried as it might, the program could not put together eight not-radios before I lost interest in watching.

I called and tried to admit defeat, but they wouldn't hear it. It turns out that the best the factory had managed after extensive efforts was only *six* working not-radios, so my result represented a significant improvement!

7.7 War Story: Annealing Arrays

The war story of Section 3.9 (page 94) reported how we used advanced data structures to simulate a new method for sequencing DNA. Our method, interactive sequencing by hybridization (SBH), required building arrays of specific oligonucleotides on demand.

A biochemist at Oxford University got interested in our technique, and moreover he had in his laboratory the equipment we needed to test it out. The Southern Array Maker, manufactured by Beckman Instruments, prepared discrete oligonucleotide sequences in 64 parallel rows across a polypropylene substrate. The device constructs arrays by appending single characters to each cell along specific rows and columns of arrays. Figure 7.12 shows how to construct an array of all $2^4 = 16$ purine (A or G) 4-mers by building the prefixes along rows and the suffixes along columns. This technology provided an ideal environment for testing the feasibility of interactive SBH in a laboratory, because with proper programming it gave a way to fabricate a wide variety of oligonucleotide arrays on demand.

However, we had to provide the proper programming. Fabricating complicated arrays required solving a difficult combinatorial problem. We were given as input a set of n strings (representing oligonucleotides) to fabricate in an $m \times m$ array (where $m = 64$ on the Southern apparatus). We had to produce a schedule of row and column commands to realize the set of strings S. We proved that the

problem of designing dense arrays was NP-complete, but that didn't really matter. My student Ricky Bradley and I had to solve it anyway.

"We are going to have to use a heuristic," I told him. "So how can we model this problem?"

"Well, each string can be partitioned into prefix and suffix pairs that realize it. For example, the string ACC can be realized in four different ways: prefix '' and suffix ACC, prefix A and suffix CC, prefix AC and suffix C, or prefix ACC and suffix ''. We seek the smallest set of prefixes and suffixes that together realize all the given strings," Ricky said.

"Good. This gives us a natural representation for simulated annealing. The state space will consist of all possible subsets of prefixes and suffixes. The natural transitions between states might include inserting or deleting strings from our subsets, or swapping a pair in or out."

"What's a good cost function?" he asked.

"Well, we need as small an array as possible that covers all the strings. How about taking the maximum of number of rows (prefixes) or columns (suffixes) used in our array, plus the number of strings from S that are not yet covered. Try it and let's see what happens."

Ricky went off and implemented a simulated annealing program along these lines. It printed out the state of the solution each time a transition was accepted and was fun to watch. The program quickly kicked out unnecessary prefixes and suffixes, and the array began shrinking rapidly in size. But after several hundred iterations, progress started to slow. A transition would knock out an unnecessary suffix, wait a while, then add a different suffix back again. After a few thousand iterations, no real improvement was happening.

"The program doesn't seem to recognize when it is making progress. The evaluation function only gives credit for minimizing the larger of the two dimensions. Why not add a term to give some credit to the other dimension."

Ricky changed the evaluation function, and we tried again. This time, the program did not hesitate to improve the shorter dimension. Indeed, our arrays started to be skinny rectangles instead of squares.

"OK. Let's add another term to the evaluation function to give it points for being roughly square."

Ricky tried again. Now the arrays were the right shape, and progress was in the right direction. But the progress was still slow.

"Too many of the insertion moves don't affect many strings. Maybe we should skew the random selections so that the important prefix/suffixes get picked more often."

Ricky tried again. Now it converged faster, but sometimes it still got stuck. We changed the cooling schedule. It did better, but was it doing well? Without a lower bound knowing how close we were to optimal, it couldn't really tell how good our solution was. We tweaked and tweaked until our program stopped improving.

Our final solution refined the initial array by applying the following random moves:

Figure 7.13: Compression of the HIV array by simulated annealing – after 0, 500, 1,000, and 5,750 iterations

---

- *swap* – swap a prefix/suffix on the array with one that isn't.
- *add* – add a random prefix/suffix to the array.
- *delete* – delete a random prefix/suffix from the array.
- *useful add* – add the prefix/suffix with the highest usefulness to the array.
- *useful delete* – delete the prefix/suffix with the lowest usefulness from the array.
- *string add* – randomly select a string not on the array, and add the most useful prefix and/or suffix to cover this string.

A standard cooling schedule was used, with an exponentially decreasing temperature (dependent upon the problem size) and a temperature-dependent Boltzmann criterion for accepting states that have higher costs. Our final cost function was defined as

$$cost = 2 \times max + min + \frac{(max - min)^2}{4} + 4(str_{total} - str_{in})$$

where max is the size of the maximum chip dimension, min is the size of the minimum chip dimension, $str_{total} = |S|$, and str_{in} is the number of strings from S currently on the chip.

How well did we do? Figure 7.13 shows the convergence of a custom array consisting of the 5,716 unique 7-mers of the HIV virus. Figure 7.13 shows snapshots of the state of the chip at four points during the annealing process (0, 500, 1,000, and the final chip at 5,750 iterations). Black pixels represent the first occurrence of an HIV 7-mer. The final chip size here is 130×132—quite an improvement over

the initial size of 192×192. It took about fifteen minutes' worth of computation to complete the optimization, which was perfectly acceptable for the application.

But how well did we do? Since simulated annealing is only a heuristic, we really don't know how close to optimal our solution is. I think we did pretty well, but can't really be sure. Simulated annealing is a good way to handle complex optimization problems. However, to get the best results, expect to spend more time tweaking and refining your program than you did in writing it in the first place. This is dirty work, but sometimes you have to do it.

7.8 Other Heuristic Search Methods

Several heuristic search methods have been proposed to search for good solutions for combinatorial optimization problems. Like simulated annealing, many techniques relies on analogies to real-world physical processes. Popular methods include *genetic algorithms*, *neural networks*, and *ant colony optimization*.

The intuition behind these methods is highly appealing, but skeptics decry them as voodoo optimization techniques that rely more on nice analogies to nature than demonstrated computational results on problems that have been studied using other methods.

The question isn't whether you can get decent answers for many problems given enough effort using these techniques. Clearly you can. The real question is whether they lead to *better* solutions with *less implementation complexity* than the other methods we have discussed.

In general, I don't believe that they do. But in the spirit of free inquiry, I introduce genetic algorithms, which is the most popular of these methods. See the chapter notes for more detailed readings.

Genetic Algorithms

Genetic algorithms draw their inspiration from evolution and natural selection. Through the process of natural selection, organisms adapt to optimize their chances for survival in a given environment. Random mutations occur in an organism's genetic description, which then get passed on to its children. Should a mutation prove helpful, these children are more likely to survive and reproduce. Should it be harmful, these children won't, and so the bad trait will die with them.

Genetic algorithms maintain a "population" of solution candidates for the given problem. Elements are drawn at random from this population and allowed to "reproduce" by combining aspects of the two-parent solutions. The probability that an element is chosen to reproduce is based on its "fitness,"—essentially the cost of the solution it represents. Unfit elements die from the population, to be replaced by a successful-solution offspring.

The idea behind genetic algorithms is extremely appealing. However, they don't seem to work as well on practical combinatorial optimization problems as simulated

annealing does. There are two primary reasons for this. First, it is quite unnatural to model applications in terms of genetic operators like mutation and crossover on bit strings. The pseudobiology adds another level of complexity between you and your problem. Second, genetic algorithms take a very long time on nontrivial problems. The crossover and mutation operations typically make no use of problem-specific structure, so most transitions lead to inferior solutions, and convergence is slow. Indeed, the analogy with evolution—where significant progress require millions of years—can be quite appropriate.

We will not discuss genetic algorithms further, to discourage you from considering them for your applications. However, pointers to implementations of genetic algorithms are provided in Section 13.5 (page 407) if you really insist on playing with them.

> *Take-Home Lesson:* I have *never* encountered any problem where genetic algorithms seemed to me the right way to attack it. Further, I have *never* seen any computational results reported using genetic algorithms that have favorably impressed me. Stick to simulated annealing for your heuristic search voodoo needs.

7.9 Parallel Algorithms

Two heads are better than one, and more generally, n heads are better than $n-1$. Parallel processing is becoming more important with the advent of cluster computing and multicore processors. It seems like the easy way out of hard problems. Indeed, sometimes, for some problems, parallel algorithms are the most effective solution. High-resolution, real-time graphics applications must render thirty frames per second for realistic animation. Assigning each frame to a distinct processor, or dividing each image into regions assigned to different processors, might be the only way to get the job done in time. Large systems of linear equations for scientific applications are routinely solved in parallel.

However, there are several pitfalls associated with parallel algorithms that you should be aware of:

- *There is often a small upper bound on the potential win* – Suppose that you have access to twenty processors that can be devoted exclusively to your job. Potentially, these could be used to speed up the fastest sequential program by up to a factor of twenty. That is nice, but greater performance gains maybe possible by finding a better sequential algorithm. Your time spent parallelizing a code might well be better spent enhancing the sequential version. Performance-tuning tools such as profilers are better developed for sequential machines than for parallel models.

- *Speedup means nothing* – Suppose my parallel program runs 20 times faster on a 20-processor machine than it does on one processor. That's great, isn't

it? If you always get linear speedup and have an arbitrary number of processors, you will eventually beat any sequential algorithm. However, a carefully designed sequential algorithm can often beat an easily-parallelized code running on a typical parallel machine. The one-processor parallel version of your code is likely to be a crummy sequential algorithm, so measuring speedup typically provides an unfair test of the benefits of parallelism.

The classic example of this occurs in the minimax game-tree search algorithm used in computer chess programs. A brute-force tree search is embarrassingly easy to parallelize: just put each subtree on a different processor. However, a lot of work gets wasted because the same positions get considered on different machines. Moving from a brute-force search to the more clever alpha-beta pruning algorithm can easily save 99.99% of the work, thus dwarfing any benefits of a parallel brute-force search. Alpha-beta can be parallelized, but not easily, and the speedups grow surprisingly slowly as a function of the number of processors you have.

- *Parallel algorithms are tough to debug* – Unless your problem can be decomposed into several independent jobs, the different processors must communicate with each other to end up with the correct final result. Unfortunately, the nondeterministic nature of this communication makes parallel programs notoriously difficult to debug. Perhaps the best example is *Deep Blue*—the world-champion chess computer. Although it eventually beat Kasparov, over the years it lost several games in embarrassing fashion due to bugs, mostly associated with its extensive parallelism.

I recommend considering parallel processing only after attempts at solving a problem sequentially prove too slow. Even then, I would restrict attention to algorithms that parallelize the problem by partitioning the input into distinct tasks where no communication is needed between the processors, except to collect the final results. Such large-grain, naive parallelism can be simple enough to be both implementable and debuggable, because it really reduces to producing a good sequential implementation. There can be pitfalls even in this approach, however, as shown in the war story below.

7.10 War Story: Going Nowhere Fast

In Section 2.8 (page 51), I related our efforts to build a fast program to test Waring's conjecture for pyramidal numbers. At that point, my code was fast enough that it could complete the job in a few weeks running in the background of a desktop workstation. This option did not appeal to my supercomputing colleague, however.

"Why don't we do it in parallel?" he suggested. "After all, you have an outer loop doing the same calculation on each integer from 1 to 1,000,000,000. I can split this range of numbers into different intervals and run each range on a different processor. Watch, it will be easy."