He set to work trying to do our computations on an Intel IPSC-860 hypercube using 32 nodes with 16 megabytes of memory per node—very big iron for the time. However, instead of getting answers, I was treated to a regular stream of e-mail about system reliability over the next few weeks:

- "Our code is running fine, except one processor died last night. I will rerun."
- "This time the machine was rebooted by accident, so our long-standing job was killed."
- "We have another problem. The policy on using our machine is that nobody can command the entire machine for more than thirteen hours, under any condition."

Still, eventually, he rose to the challenge. Waiting until the machine was stable, he locked out 16 processors (half the computer), divided the integers from 1 to 1,000,000,000 into 16 equal-sized intervals, and ran each interval on its own processor. He spent the next day fending off angry users who couldn't get their work done because of our rogue job. The instant the first processor completed analyzing the numbers from 1 to 62,500,000, he announced to all the people yelling at him that the rest of the processors would soon follow.

But they didn't. He failed to realize that the time to test each integer increased as the numbers got larger. After all, it would take longer to test whether 1,000,000,000 could be expressed as the sum of three pyramidal numbers than it would for 100. Thus, at slower and slower intervals each new processor would announce its completion. Because of the architecture of the hypercube, he couldn't return any of the processors until our entire job was completed. Eventually, half the machine and most of its users were held hostage by one, final interval.

What conclusions can be drawn from this? If you are going to parallelize a problem, be sure to balance the load carefully among the processors. Proper load balancing, using either back-of-the-envelope calculations or the partition algorithm we will develop in Section 8.5 (page 294), would have significantly reduced the time we needed the machine, and his exposure to the wrath of his colleagues.

Chapter Notes

The treatment of backtracking here is partially based on my book *Programming Challenges* [SR03]. In particular, the `backtrack` routine presented here is a generalization of the version in Chapter 8 of [SR03]. Look there for my solution to the famous *eight queens problem*, which seeks all chessboard configurations of eight mutually nonattacking queens on an 8×8 board.

The original paper on simulated annealing [KGV83] included an application to VLSI module placement problems. The applications from Section 7.5.4 (page 258) are based on material from [AK89].

The heuristic TSP solutions presented here employ vertex-swap as the local neighborhood operation. In fact, edge-swap is a more powerful operation. Each edge-swap changes two edges in the tour at most, as opposed to at most four edges with a vertex-swap. This improves the possibility of a local improvement. However, more sophisticated data structures are necessary to efficiently maintain the order of the resulting tour [FJMO93].

The different heuristic search techniques are ably presented in Aarts and Lenstra [AL97], which I strongly recommend for those interested in learning more about heuristic searches. Their coverage includes *tabu search*, a variant of simulated annealing that uses extra data structures to avoid transitions to recently visited states. Ant colony optimization is discussed in [DT04]. See [MF00] for a more favorable view of genetic algorithms and the like.

More details on our combinatorial search for optimal chessboard-covering positions appear in our paper [RHS89]. Our work using simulated annealing to compress DNA arrays was reported in [BS97]. See Pugh [Pug86] and Coullard et al. [CGJ98] for more on selective assembly. Our parallel computations on pyramidal numbers were reported in [DY94].

7.11 Exercises

Backtracking

7-1. *[3]* A *derangement* is a permutation p of $\{1, \ldots, n\}$ such that no item is in its proper position, i.e. $p_i \neq i$ for all $1 \leq i \leq n$. Write an efficient backtracking program with pruning that constructs all the derangements of n items.

7-2. *[4] Multisets* are allowed to have repeated elements. A multiset of n items may thus have fewer than $n!$ distinct permutations. For example, $\{1, 1, 2, 2\}$ has only six different permutations: $\{1, 1, 2, 2\}$, $\{1, 2, 1, 2\}$, $\{1, 2, 2, 1\}$, $\{2, 1, 1, 2\}$, $\{2, 1, 2, 1\}$, and $\{2, 2, 1, 1\}$. Design and implement an efficient algorithm for constructing all permutations of a multiset.

7-3. *[5]* Design and implement an algorithm for testing whether two graphs are isomorphic to each other. The graph isomorphism problem is discussed in Section 16.9 (page 550). With proper pruning, graphs on hundreds of vertices can be tested reliably.

7-4. *[5]* Anagrams are rearrangements of the letters of a word or phrase into a different word or phrase. Sometimes the results are quite striking. For example, "MANY VOTED BUSH RETIRED" is an anagram of "TUESDAY NOVEMBER THIRD," which correctly predicted the result of the 1992 U.S. presidential election. Design and implement an algorithm for finding anagrams using combinatorial search and a dictionary.

7-5. *[8]* Design and implement an algorithm for solving the subgraph isomorphism problem. Given graphs G and H, does there exist a subgraph H' of H such that G is isomorphic to H'? How does your program perform on such special cases of subgraph isomorphism as Hamiltonian cycle, clique, independent set, and graph isomorphism?

7-6. *[8]* In the turnpike reconstruction problem, you are given $n(n-1)/2$ distances in sorted order. The problem is to find the positions of n points on the line that give rise to these distances. For example, the distances $\{1, 2, 3, 4, 5, 6\}$ can be determined by placing the second point 1 unit from the first, the third point 3 from the second, and the fourth point 2 from the third. Design and implement an efficient algorithm to report all solutions to the turnpike reconstruction problem. Exploit additive constraints when possible to minimize the search. With proper pruning, problems with hundreds of points can be solved reliably.

Combinatorial Optimization

For each of the problems below, either (1) implement a combinatorial search program to solve it optimally for small instance, and/or (2) design and implement a simulated annealing heuristic to get reasonable solutions. How well does your program perform in practice?

7-7. *[5]* Design and implement an algorithm for solving the bandwidth minimization problem discussed in Section 13.2 (page 398).

7-8. *[5]* Design and implement an algorithm for solving the maximum satisfiability problem discussed in Section 14.10 (page 472).

7-9. *[5]* Design and implement an algorithm for solving the maximum clique problem discussed in Section 16.1 (page 525).

7-10. *[5]* Design and implement an algorithm for solving the minimum vertex coloring problem discussed in Section 16.7 (page 544).

7-11. *[5]* Design and implement an algorithm for solving the minimum edge coloring problem discussed in Section 16.8 (page 548).

7-12. *[5]* Design and implement an algorithm for solving the minimum feedback vertex set problem discussed in Section 16.11 (page 559).

7-13. *[5]* Design and implement an algorithm for solving the set cover problem discussed in Section 18.1 (page 621).

Interview Problems

7-14. *[4]* Write a function to find all permutations of the letters in a particular string.

7-15. *[4]* Implement an efficient algorithm for listing all k-element subsets of n items.

7-16. *[5]* An anagram is a rearrangement of the letters in a given string into a sequence of dictionary words, like *Steven Skiena* into *Vainest Knees*. Propose an algorithm to construct all the anagrams of a given string.

7-17. *[5]* Telephone keypads have letters on each numerical key. Write a program that generates all possible words resulting from translating a given digit sequence (e.g., 145345) into letters.

7-18. *[7]* You start with an empty room and a group of n people waiting outside. At each step, you may either admit one person into the room, or let one out. Can you arrange a sequence of 2^n steps, so that every possible combination of people is achieved exactly once?

7-19. *[4]* Use a random number generator (rng04) that generates numbers from $\{0, 1, 2, 3, 4\}$ with equal probability to write a random number generator that generates numbers from 0 to 7 (rng07) with equal probability. What are expected number of calls to rng04 per call of rng07?

Programming Challenges

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

7-1. "Little Bishops" – Programming Challenges 110801, UVA Judge 861.

7-2. "15-Puzzle Problem" – Programming Challenges 110802, UVA Judge 10181.

7-3. "Tug of War" – Programming Challenges 110805, UVA Judge 10032.

7-4. "Color Hash" – Programming Challenges 110807, UVA Judge 704.

8

Dynamic Programming

The most challenging algorithmic problems involve optimization, where we seek to find a solution that maximizes or minimizes some function. Traveling salesman is a classic optimization problem, where we seek the tour visiting all vertices of a graph at minimum total cost. But as shown in Chapter 1, it is easy to propose "algorithms" solving TSP that generate reasonable-looking solutions but did not *always* produce the minimum cost tour.

Algorithms for optimization problems require proof that they always return the best possible solution. Greedy algorithms that make the best local decision at each step are typically efficient but usually do not guarantee global optimality. Exhaustive search algorithms that try all possibilities and select the best always produce the optimum result, but usually at a prohibitive cost in terms of time complexity.

Dynamic programming combines the best of both worlds. It gives us a way to design custom algorithms that systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency). By storing the *consequences* of all possible decisions and using this information in a systematic way, the total amount of work is minimized.

Once you understand it, dynamic programming is probably the easiest algorithm design technique to apply in practice. In fact, I find that dynamic programming algorithms are often easier to reinvent than to try to look up in a book. That said, *until* you understand dynamic programming, it seems like magic. You must figure out the trick before you can use it.

Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results. The trick is seeing whether the naive recursive algorithm computes the same subproblems over and over and over again. If so, storing the answer for each subproblems in a table to look up instead of recompute

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_8,

can lead to an efficient algorithm. Start with a recursive algorithm or definition. Only once we have a correct recursive algorithm do we worry about speeding it up by using a results matrix.

Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent *left to right* order among components. Left-to-right objects includes: character strings, rooted trees, polygons, and integer sequences. Dynamic programming is best learned by carefully studying examples until things start to click. We present three war stories where dynamic programming played the decisive role to demonstrate its utility in practice.

8.1 Caching vs. Computation

Dynamic programming is essentially a tradeoff of space for time. Repeatedly recomputing a given quantity is harmless unless the time spent doing so becomes a drag on performance. Then we are better off storing the results of the initial computation and looking them up instead of recomputing them again.

The tradeoff between space and time exploited in dynamic programming is best illustrated when evaluating recurrence relations such as the Fibonacci numbers. We look at three different programs for computing them below.

8.1.1 Fibonacci Numbers by Recursion

The Fibonacci numbers were originally defined by the Italian mathematician Fibonacci in the thirteenth century to model the growth of rabbit populations. Rabbits breed, well, like rabbits. Fibonacci surmised that the number of pairs of rabbits born in a given year is equal to the number of pairs of rabbits born in each of the two previous years, starting from one pair of rabbits in the first year. To count the number of rabbits born in the nth year, he defined the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

with basis cases $F_0 = 0$ and $F_1 = 1$. Thus, $F_2 = 1$, $F_3 = 2$, and the series continues $\{3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots\}$. As it turns out, Fibonacci's formula didn't do a very good job of counting rabbits, but it does have a host of interesting properties.

Since they are defined by a recursive formula, it is easy to write a recursive program to compute the nth Fibonacci number. A recursive function algorithm written in C looks like this:

```
long fib_r(int n)
{
        if (n == 0) return(0);
        if (n == 1) return(1);

        return(fib_r(n-1) + fib_r(n-2));
}
```

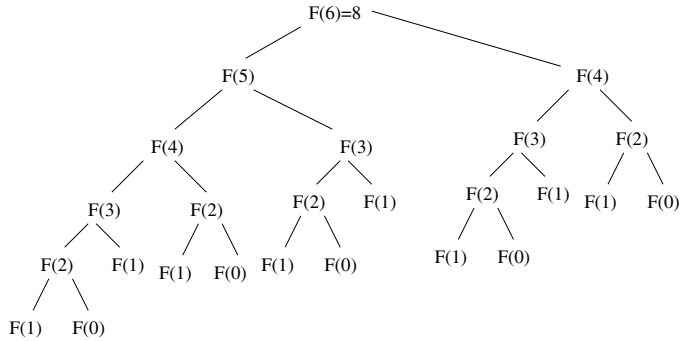


Figure 8.1: The computation tree for computing Fibonacci numbers recursively

The course of execution for this recursive algorithm is illustrated by its *recursion tree*, as illustrated in Figure 8.1. This tree is evaluated in a depth-first fashion, as are all recursive algorithms. I encourage you to trace this example by hand to refresh your knowledge of recursion.

Note that $F(4)$ is computed on both sides of the recursion tree, and $F(2)$ is computed no less than five times in this small example. The weight of all this redundancy becomes clear when you run the program. It took more than 7 minutes for my program to compute the first 45 Fibonacci numbers. You could probably do it faster by hand using the right algorithm.

How much time does this algorithm take to compute $F(n)$? Since $F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$, this means that $F_n > 1.6^n$. Since our recursion tree has only 0 and 1 as leaves, summing up to such a large number means we must have at least 1.6^n leaves or procedure calls! This humble little program takes exponential time to run!

8.1.2 Fibonacci Numbers by Caching

In fact, we can do much better. We can explicitly store (or *cache*) the results of each Fibonacci computation $F(k)$ in a table data structure indexed by the parameter k. The key to avoiding recomputation is to explicitly check for the value before trying to compute it:

```
#define MAXN    45       /* largest interesting n */
#define UNKNOWN   -1     /* contents denote an empty cell */
long f[MAXN+1];          /* array for caching computed fib values */
```

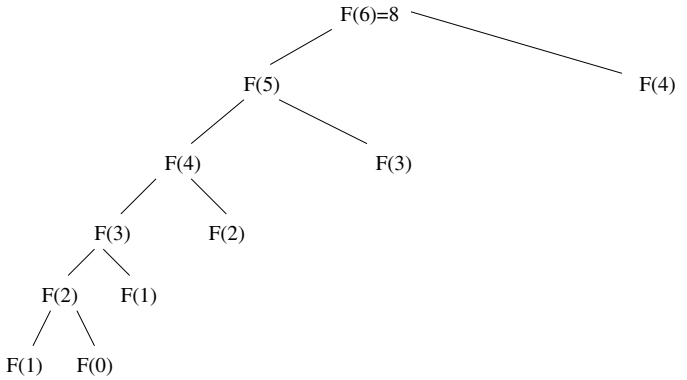


Figure 8.2: The Fibonacci computation tree when caching values

```
long fib_c(int n)
{
        if (f[n] == UNKNOWN)
                f[n] = fib_c(n-1) + fib_c(n-2);

        return(f[n]);
}


long fib_c_driver(int n)
{
        int i;                  /* counter */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = UNKNOWN;

        return(fib_c(n));
}
```

To compute $F(n)$, we call `fib_c_driver(n)`. This initializes our cache to the two values we initially know ($F(0)$ and $F(1)$) as well as the `UNKNOWN` flag for all the rest we don't. It then calls a look-before-crossing-the-street version of the recursive algorithm.

This cached version runs instantly up to the largest value that can fit in a long integer. The new recursion tree (Figure 8.2) explains why. There is no meaningful branching, because only the left-side calls do computation. The right-side calls find what they are looking for in the cache and immediately return.

What is the running time of this algorithm? The recursion tree provides more of a clue than the code. In fact, it computes $F(n)$ in linear time (in other words, $O(n)$ time) because the recursive function `fib_c(k)` is called exactly twice for each value $0 \leq k \leq n$.

This general method of explicitly caching results from recursive calls to avoid recomputation provides a simple way to get *most* of the benefits of full dynamic programming, so it is worth a more careful look. In principle, such caching can be employed on any recursive algorithm. However, storing partial results would have done absolutely no good for such recursive algorithms as *quicksort*, *backtracking*, and *depth-first search* because all the recursive calls made in these algorithms have distinct *parameter values*. It doesn't pay to store something you will never refer to again.

Caching makes sense only when the space of distinct parameter values is modest enough that we can afford the cost of storage. Since the argument to the recursive function `fib_c(k)` is an integer between 0 and n, there are only $O(n)$ values to cache. A linear amount of space for an exponential amount of time is an excellent tradeoff. But as we shall see, we can do even better by eliminating the recursion completely.

> *Take-Home Lesson:* Explicit caching of the results of recursive calls provides *most* of the benefits of dynamic programming, including usually the same running time as the more elegant full solution. If you prefer doing extra programming to more subtle thinking, you can stop here.

8.1.3 Fibonacci Numbers by Dynamic Programming

We can calculate F_n in linear time more easily by explicitly specifying the order of evaluation of the recurrence relation:

```
long fib_dp(int n)
{
        int i;              /* counter */
        long f[MAXN+1];     /* array to cache computed fib values */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = f[i-1]+f[i-2];

        return(f[n]);
}
```

We have removed all recursive calls! We evaluate the Fibonacci numbers from smallest to biggest and store all the results, so we know that we have F_{i-1} and F_{i-2} ready whenever we need to compute F_i. The linearity of this algorithm should

be apparent. Each of the n values is computed as the simple sum of two integers in total $O(n)$ time and space.

More careful study shows that we do not need to store all the intermediate values for the entire period of execution. Because the recurrence depends on two arguments, we only need to retain the last two values we have seen:

```
long fib_ultimate(int n)
{
        int i;                  /* counter */
        long back2=0, back1=1;  /* last two values of f[n] */
        long next;              /* placeholder for sum */

        if (n == 0) return (0);

        for (i=2; i<n; i++) {
                next = back1+back2;
                back2 = back1;
                back1 = next;
        }
        return(back1+back2);
}
```

This analysis reduces the storage demands to constant space with no asymptotic degradation in running time.

8.1.4 Binomial Coefficients

We now show how to compute the *binomial coefficients* as another illustration of how to eliminate recursion by specifying the order of evaluation. The binomial coefficients are the most important class of counting numbers, where $\binom{n}{k}$ counts the number of ways to choose k things out of n possibilities.

How do you compute the binomial coefficients? First, $\binom{n}{k} = n!/((n-k)!k!)$, so in principle you can compute them straight from factorials. However, this method has a serious drawback. Intermediate calculations can easily cause arithmetic overflow, even when the final coefficient fits comfortably within an integer.

A more stable way to compute binomial coefficients is using the recurrence relation implicit in the construction of Pascal's triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

m / n	0	1	2	3	4	5
0	A					
1	B	G				
2	C	1	H			
3	D	2	3	I		
4	E	4	5	6	J	
5	F	7	8	9	10	K

m / n	0	1	2	3	4	5
0	1					
1	1	1				
2	1	1	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	**5**	1

Figure 8.3: Evaluation order for `binomial_coefficient` at $M[5,4]$ (l). Initialization conditions A-K, recurrence evaluations 1-10. Matrix contents after evaluation (r)

Each number is the sum of the two numbers directly above it. The recurrence relation implicit in this is that

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Why does this work? Consider whether the nth element appears in one of the $\binom{n}{k}$ subsets of k elements. If so, we can complete the subset by picking $k-1$ other items from the other $n-1$. If not, we must pick all k items from the remaining $n-1$. There is no overlap between these cases, and all possibilities are included, so the sum counts all k subsets.

No recurrence is complete without basis cases. What binomial coefficient values do we know without computing them? The left term of the sum eventually drives us down to $\binom{n-k}{0}$. How many ways are there to choose 0 things from a set? Exactly one, the empty set. If this is not convincing, then it is equally good to accept that $\binom{m}{1} = m$ as the basis case. The right term of the sum runs us up to $\binom{k}{k}$. How many ways are there to choose k things from a k-element set? Exactly one—the complete set. Together, these basis cases and the recurrence define the binomial coefficients on all interesting values.

The best way to evaluate such a recurrence is to build a table of possible values up to the size that you are interested in:

Figure 8.3 demonstrates a proper evaluation order for the recurrence. The initialized cells are marked from A-K, denoting the order in which they were assigned values. Each remaining cell is assigned the sum of the cell directly above it and the cell immediately above and to the left. The triangle of cells marked 1 to 10 denote the evaluation order in computing $\binom{5}{4} = 5$ using the code below:

```
long binomial_coefficient(n,m)
int n,m;                          /* computer n choose m */
{
        int i,j;                  /* counters */
        long bc[MAXN][MAXN];      /* table of binomial coefficients */

        for (i=0; i<=n; i++) bc[i][0] = 1;

        for (j=0; j<=n; j++) bc[j][j] = 1;

        for (i=1; i<=n; i++)
                for (j=1; j<i; j++)
                        bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

        return( bc[n][m] );
}
```

Study this function carefully to see how we did it. The rest of this chapter will focus more on formulating and analyzing the appropriate recurrence than the mechanics of table manipulation demonstrated here.

8.2 Approximate String Matching

Searching for patterns in text strings is a problem of unquestionable importance. Section 3.7.2 (page 91) presented algorithms for *exact* string matching—finding where the pattern string P occurs as a substring of the text string T. Life is often not that simple. Words in either the text or pattern can be mispelled (sic), robbing us of exact similarity. Evolutionary changes in genomic sequences or language usage imply that we often search with archaic patterns in mind: "Thou shalt not kill" morphs over time into "You should not murder."

How can we search for the substring closest to a given pattern to compensate for spelling errors? To deal with inexact string matching, we must first define a cost function telling us how far apart two strings are—i.e., a distance measure between pairs of strings. A reasonable distance measure reflects the number of *changes* that must be made to convert one string to another. There are three natural types of changes:

- *Substitution* – Replace a single character from pattern P with a different character in text T, such as changing "shot" to "spot."
- *Insertion* – Insert a single character into pattern P to help it match text T, such as changing "ago" to "agog."
- *Deletion* – Delete a single character from pattern P to help it match text T, such as changing "hour" to "our."

Properly posing the question of string similarity requires us to set the cost of each of these string transform operations. Assigning each operation an equal cost of 1 defines the *edit distance* between two strings. Approximate string matching arises in many applications, as discussed in Section 18.4 (page 631).

Approximate string matching seems like a difficult problem, because we must decide exactly where to delete and insert (potentially) many characters in pattern and text. But let us think about the problem in reverse. What information would we like to have to make the final decision? What can happen to the last character in the matching for each string?

8.2.1 Edit Distance by Recursion

We can define a recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted. Chopping off the characters involved in this last edit operation leaves a pair of smaller strings. Let i and j be the last character of the relevant prefix of P and T, respectively. There are three pairs of shorter strings after the last operation, corresponding to the strings after a match/substitution, insertion, or deletion. *If* we knew the cost of editing these three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly. We *can* learn this cost through the magic of recursion.

More precisely, let $D[i, j]$ be the minimum number of differences between $P_1, P_2, \ldots, P_i$ and the segment of T ending at j. $D[i, j]$ is the *minimum* of the three possible ways to extend smaller strings:

- If $(P_i = T_j)$, then $D[i-1, j-1]$, else $D[i-1, j-1]+1$. This means we either match or substitute the ith and jth characters, depending upon whether the tail characters are the same.

- $D[i-1, j]+1$. This means that there is an extra character in the pattern to account for, so we do not advance the text pointer and pay the cost of an insertion.

- $D[i, j-1]+1$. This means that there is an extra character in the text to remove, so we do not advance the pattern pointer and pay the cost of a deletion.

```
#define MATCH       0       /* enumerated type symbol for match */
#define INSERT      1       /* enumerated type symbol for insert */
#define DELETE      2       /* enumerated type symbol for delete */
```

```
int string_compare(char *s, char *t, int i, int j)
{
        int k;                  /* counter */
        int opt[3];             /* cost of the three options */
        int lowest_cost;        /* lowest cost */

        if (i == 0) return(j * indel(' '));
        if (j == 0) return(i * indel(' '));

        opt[MATCH] = string_compare(s,t,i-1,j-1) + match(s[i],t[j]);
        opt[INSERT] = string_compare(s,t,i,j-1) + indel(t[j]);
        opt[DELETE] = string_compare(s,t,i-1,j) + indel(s[i]);

        lowest_cost = opt[MATCH];
        for (k=INSERT; k<=DELETE; k++)
                if (opt[k] < lowest_cost) lowest_cost = opt[k];

        return( lowest_cost );
}
```

This program is absolutely correct—convince yourself. It also turns out to be impossibly slow. Running on my computer, the computation takes several seconds to compare two 11-character strings, and disappears into Never-Never Land on anything longer.

Why is the algorithm so slow? It takes exponential time because it recomputes values again and again and again. At every position in the string, the recursion branches three ways, meaning it grows at a rate of at least 3^n—indeed, even faster since most of the calls reduce only one of the two indices, not both of them.

8.2.2 Edit Distance by Dynamic Programming

So, how can we make this algorithm practical? The important observation is that most of these recursive calls are computing things that have been previously computed. How do we know? There can only be $|P| \cdot |T|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the argument parameters of recursive calls. By storing the values for each of these (i, j) pairs in a table, we just look them up as needed and avoid recomputing them.

A table-based, dynamic programming implementation of this algorithm is given below. The table is a two-dimensional matrix m where each of the $|P| \cdot |T|$ cells contains the cost of the optimal solution to a subproblem, as well as a parent pointer explaining how we got to this location:

```
typedef struct {
        int cost;               /* cost of reaching this cell */
        int parent;             /* parent cell */
} cell;

cell m[MAXLEN+1][MAXLEN+1];     /* dynamic programming table */
```

Our dynamic programming implementation has three differences from the recursive version. First, it gets its intermediate values using table lookup instead of recursive calls. Second, it updates the `parent` field of each cell, which will enable us to reconstruct the edit sequence later. Third, it is implemented using a more general `goal_cell()` function instead of just returning `m[|P|][|T|].cost`. This will enable us to apply this routine to a wider class of problems.

```
int string_compare(char *s, char *t)
{
        int i,j,k;              /* counters */
        int opt[3];             /* cost of the three options */

        for (i=0; i<MAXLEN; i++) {
                row_init(i);
                column_init(i);
        }

        for (i=1; i<strlen(s); i++) {
                for (j=1; j<strlen(t); j++) {
                        opt[MATCH] = m[i-1][j-1].cost + match(s[i],t[j]);
                        opt[INSERT] = m[i][j-1].cost + indel(t[j]);
                        opt[DELETE] = m[i-1][j].cost + indel(s[i]);

                        m[i][j].cost = opt[MATCH];
                        m[i][j].parent = MATCH;
                        for (k=INSERT; k<=DELETE; k++)
                                if (opt[k] < m[i][j].cost) {
                                        m[i][j].cost = opt[k];
                                        m[i][j].parent = k;
                                }
                }
        }
        goal_cell(s,t,&i,&j);
        return( m[i][j].cost );
}
```

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
:		**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	**1**	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	**2**	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	3	**2**	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	4	3	**2**	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	5	4	3	**2**	3	4	5	6	6	7	7	8	9	10
s:	6	6	6	5	4	3	**2**	3	4	5	6	7	8	8	9	10
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6	7	8	9	10
a:	8	8	8	7	6	5	4	3	3	**4**	5	6	7	8	9	10
l:	9	9	9	8	7	6	5	4	4	4	**4**	5	6	7	8	9
t:	10	10	10	9	8	7	6	5	5	5	5	**5**	6	7	8	8
-:	11	11	11	10	9	8	7	6	6	6	6	6	**5**	6	7	8
n:	12	12	12	11	10	9	8	7	7	7	7	7	6	**5**	6	7
o:	13	13	13	12	11	10	9	8	7	8	8	8	7	6	**5**	6
t:	14	14	14	13	12	11	10	9	8	8	9	9	8	7	6	**5**

Figure 8.4: Example of a dynamic programming matrix for editing distance computation, with the optimal alignment path highlighted in bold

Be aware that we adhere to somewhat unusual string and index conventions in the routine above. In particular, we assume that each string has been padded with an initial blank character, so the first real character of string `s` sits in `s[1]`. Why did we do this? It enables us to keep the matrix `m` indices in sync with those of the strings for clarity. Recall that we must dedicate the zeroth row and column of `m` to store the boundary values matching the empty prefix. Alternatively, we could have left the input strings intact and just adjusted the indices accordingly.

To determine the value of cell (i, j), we need three values sitting and waiting for us—namely, the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$. Any evaluation order with this property will do, including the row-major order used in this program.[1]

As an example, we show the cost matrices for turning $p =$ "thou shalt not" into $t =$ "you should not" in five moves in Figure 8.4.

8.2.3 Reconstructing the Path

The string comparison function returns the cost of the optimal alignment, but not the alignment itself. Knowing you can convert "thou shalt not" to "you should not" in only five moves is dandy, but what is the sequence of editing operations that does it?

The possible solutions to a given dynamic programming problem are described by paths through the dynamic programming matrix, starting from the initial configuration (the pair of empty strings $(0, 0)$) down to the final goal state (the pair

[1]Suppose we create a graph with a vertex for every matrix cell, and a directed edge (x, y), when the value of cell x is needed to compute the value of cell y. Any topological sort on the resulting DAG (why must it be a DAG?) defines an acceptable evaluation order.

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	0	**-1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1
t:	1	**2**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h:	2	2	**0**	0	0	0	0	0	1	1	1	1	1	1	1	1
o:	3	2	0	**0**	0	0	0	0	0	1	1	1	1	1	0	1
u:	4	2	0	2	**0**	1	1	1	1	0	1	1	1	1	1	1
-:	5	2	0	2	2	**0**	1	1	1	1	0	0	0	1	1	1
s:	6	2	0	2	2	2	**0**	1	1	1	1	0	0	0	0	0
h:	7	2	0	2	2	2	2	**0**	**1**	1	1	1	1	1	0	0
a:	8	2	0	2	2	2	2	2	0	**0**	0	0	0	0	0	0
l:	9	2	0	2	2	2	2	2	0	0	**0**	1	1	1	1	1
t:	10	2	0	2	2	2	2	2	0	0	0	**0**	0	0	0	0
-:	11	2	0	2	2	0	2	2	0	0	0	0	**0**	1	1	1
n:	12	2	0	2	2	2	2	2	0	0	0	0	2	**0**	1	1
o:	13	2	0	0	2	2	2	2	0	0	0	0	2	2	**0**	1
t:	14	2	0	2	2	2	2	2	2	0	0	0	2	2	2	**0**

Figure 8.5: Parent matrix for edit distance computation, with the optimal alignment path highlighted in bold

of full strings ($|P|, |T|$)). The key to building the solution is to reconstruct the decisions made at every step along the optimal path that leads to the goal state. These decisions have been recorded in the `parent` field of each array cell.

Reconstructing these decisions is done by walking backward from the goal state, following the `parent` pointer back to an earlier cell. We repeat this process until we arrive back at the initial cell. The `parent` field for `m[i,j]` tells us whether the operation at (i, j) was `MATCH`, `INSERT`, or `DELETE`. Tracing back through the parent matrix in Figure 8.5 yields the edit sequence `DSMMMMMISMSMMMM` from "thou-shalt-not" to "you-should-not"—meaning delete the first "t", replace the "h" with "y", match the next five characters before inserting an "o", replace "a" with "u", and finally replace the "t" with a "d."

Walking backward reconstructs the solution in reverse order. However, clever use of recursion can do the reversing for us:

```
reconstruct_path(char *s, char *t, int i, int j)
{
        if (m[i][j].parent == -1) return;

        if (m[i][j].parent == MATCH) {
                reconstruct_path(s,t,i-1,j-1);
                match_out(s, t, i, j);
                return;
        }
        if (m[i][j].parent == INSERT) {
                reconstruct_path(s,t,i,j-1);
                insert_out(t,j);
```

```
                return;
        }
        if (m[i][j].parent == DELETE) {
                reconstruct_path(s,t,i-1,j);
                delete_out(s,i);
                return;
        }
}
```

For many problems, including edit distance, the tour can be reconstructed from the cost matrix without explicitly retaining the last-move pointer array. The trick is working backward from the costs of the three possible ancestor cells and corresponding string characters to reconstruct the move that took you to the current cell at the given cost.

8.2.4 Varieties of Edit Distance

The `string_compare` and path reconstruction routines reference several functions that we have not yet defined. These fall into four categories:

- *Table Initialization* – The functions `row_init` and `column_init` initialize the zeroth row and column of the dynamic programming table, respectively. For the string edit distance problem, cells $(i, 0)$ and $(0, i)$ correspond to matching length-i strings against the empty string. This requires exactly i insertions/deletions, so the definition of these functions is clear:

```
row_init(int i)
{
 m[0][i].cost = i;
 if (i>0)
     m[0][i].parent =  INSERT;
 else
     m[0][i].parent = -1;
}
```

```
column_init(int i)
{
 m[i][0].cost = i;
 if (i>0)
     m[i][0].parent = DELETE;
 else
     m[i][0].parent = -1;
}
```

- *Penalty Costs* – The functions `match(c,d)` and `indel(c)` present the costs for transforming character c to d and inserting/deleting character c. For standard edit distance, `match` should cost nothing if the characters are identical, and 1 otherwise; while `indel` returns 1 regardless of what the argument is. But application-specific cost functions can be employed, perhaps more forgiving of replacements located near each other on standard keyboard layouts or characters that sound or look similar.

```
int match(char c, char d)
{
  if (c == d) return(0);
  else return(1);
}
```

```
int indel(char c)
{
  return(1);
}
```

- *Goal Cell Identification* – The function `goal_cell` returns the indices of the cell marking the endpoint of the solution. For edit distance, this is defined by the length of the two input strings. However, other applications we will soon encounter do not have fixed goal locations.

```
goal_cell(char *s, char *t, int *i, int *j)
{
  *i = strlen(s) - 1;
  *j = strlen(t) - 1;
}
```

- *Traceback Actions* – The functions `match_out`, `insert_out`, and `delete_out` perform the appropriate actions for each edit operation during traceback. For edit distance, this might mean printing out the name of the operation or character involved, as determined by the needs of the application.

```
insert_out(char *t, int j)
{
        printf("I");
}

delete_out(char *s, int i)
{
        printf("D");
}
```

```
match_out(char *s, char *t,
          int i, int j)
{
    if (s[i]==t[j]) printf("M");
    else printf("S");
}
```

All of these functions are quite simple for edit distance computation. However, we must confess it is difficult to get the boundary conditions and index manipulations correctly. Although dynamic programming algorithms are easy to design once you understand the technique, getting the details right requires carefully thinking and thorough testing.

This may seem to be a lot of infrastructure to develop for such a simple algorithm. However, several important problems can now be solved as special cases of edit distance using only minor changes to some of these stub functions:

- *Substring Matching* – Suppose that we want to find where a short pattern P best occurs within a long text T—say, searching for "Skiena" in all its misspellings (Skienna, Skena, Skina, ...) within a long file. Plugging this

search into our original edit distance function will achieve little sensitivity, since the vast majority of any edit cost will consist of deleting that which is not "Skiena" from the body of the text. Indeed, matching any scattered $\cdots S \cdots k \cdots i \cdots e \cdots n \cdots a$ and deleting the rest yields an optimal solution.

We want an edit distance search where the cost of starting the match is independent of the position in the text, so that a match in the middle is not prejudiced against. Now the goal state is not necessarily at the end of both strings, but the cheapest place to match the entire pattern somewhere in the text. Modifying these two functions gives us the correct solution:

```
row_init(int i)
{
        m[0][i].cost = 0;       /* note change */
        m[0][i].parent = -1;    /* note change */
}

goal_cell(char *s, char *t, int *i, int *j)
{
        int k;                  /* counter */

        *i = strlen(s) - 1;
        *j = 0;
        for (k=1; k<strlen(t); k++)
                if (m[*i][k].cost < m[*i][*j].cost) *j = k;
}
```

- *Longest Common Subsequence* – Perhaps we are interested in finding the longest scattered string of characters included within both strings. Indeed, this problem will be discussed in Section 18.8. The *longest common subsequence* (LCS) between "democrat" and "republican" is *eca*.

  A common subsequence is defined by all the identical-character matches in an edit trace. To maximize the number of such matches, we must prevent substitution of nonidentical characters. With substitution forbidden, the only way to get rid of the noncommon subsequence is through insertion and deletion. The minimum cost alignment has the fewest such "in-dels", so it must preserve the longest common substring. We get the alignment we want by changing the match-cost function to make substitutions expensive:

```
int match(char c, char d)
{
        if (c == d) return(0);
        else return(MAXLEN);
}
```

Actually, it suffices to make the substitution penalty greater than that of an insertion plus a deletion for substitution to lose any allure as a possible edit operation.

- *Maximum Monotone Subsequence* – A numerical sequence is *monotonically increasing* if the ith element is at least as big as the $(i-1)$st element. The *maximum monotone subsequence* problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence. A longest increasing subsequence of 243517698 is 23568.

  In fact, this is just a longest common subsequence problem, where the second string is the elements of S sorted in increasing order. Any common sequence of these two must (a) represent characters in proper order in S, and (b) use only characters with increasing position in the collating sequence—so, the longest one does the job. Of course, this approach can be modified to give the longest decreasing sequence by simply reversing the sorted order.

As you can see, our edit distance routine can be made to do many amazing things easily. The trick is observing that your problem is just a special case of approximate string matching.

The alert reader may notice that it is unnecessary to keep all $O(mn)$ cells to compute the cost of an alignment. If we evaluate the recurrence by filling in the columns of the matrix from left to right, we will never need more than two columns of cells to store what is necessary for the computation. Thus, $O(m)$ space is sufficient to evaluate the recurrence without changing the time complexity. Unfortunately, we cannot reconstruct the alignment without the full matrix.

Saving space in dynamic programming is very important. Since memory on any computer is limited, using $O(nm)$ space proves more of a bottleneck than $O(nm)$ time. Fortunately, there is a clever divide-and-conquer algorithm that computes the actual alignment in $O(nm)$ time and $O(m)$ space. It is discussed in Section 18.4 (page 631).

8.3 Longest Increasing Sequence

There are three steps involved in solving a problem by dynamic programming:

1. Formulate the answer as a recurrence relation or recursive algorithm.

2. Show that the number of different parameter values taken on by your recurrence is bounded by a (hopefully small) polynomial.

3. Specify an order of evaluation for the recurrence so the partial results you need are always available when you need them.

To see how this is done, let's see how we would develop an algorithm to find the longest monotonically increasing subsequence within a sequence of n numbers. Truth be told, this was described as a special case of edit distance in the previous section, where it was called *maximum monotone subsequence*. Still, it is instructive to work it out from scratch. Indeed, dynamic programming algorithms are often easier to reinvent than look up.

We distinguish an increasing sequence from a *run*, where the elements must be physical neighbors of each other. The selected elements of both must be sorted in increasing order from left to right. For example, consider the sequence

$$S = \{2, 4, 3, 5, 1, 7, 6, 9, 8\}$$

The longest increasing subsequence of S has length 5, including {2,3,5,6,8}. In fact, there are eight of this length (can you enumerate them?). There are four longest increasing runs of length 2: $(2, 4)$, $(3, 5)$, $(1, 7)$, and $(6, 9)$.

Finding the longest increasing run in a numerical sequence is straightforward. Indeed, you should be able to devise a linear-time algorithm fairly easily. But finding the longest increasing subsequence is considerably trickier. How can we identify which scattered elements to skip? To apply dynamic programming, we need to construct a recurrence that computes the length of the longest sequence. To find the right recurrence, ask what information about the first $n - 1$ elements of S would help you to find the answer for the entire sequence?

- The length of the longest increasing sequence in $s_1, s_2, \ldots, s_{n-1}$ seems a useful thing to know. In fact, this will be the longest increasing sequence in S, unless s_n extends some increasing sequence of the same length.

  Unfortunately, the length of this sequence is not enough information to complete the full solution. Suppose I told you that the longest increasing sequence in $s_1, s_2, \ldots, s_{n-1}$ was of length 5 and that $s_n = 9$. Will the length of the final longest increasing subsequence of S be 5 or 6?

- We need to know the length of the longest sequence that s_n will extend. To be certain we know this, we really need the length of the longest sequence that *any* possible value for s_n can extend.

This provides the idea around which to build a recurrence. Define l_i to be the length of the longest sequence ending with s_i.

The longest increasing sequence containing the nth number will be formed by appending it to the longest increasing sequence to the left of n that ends on a number smaller than s_n. The following recurrence computes l_i:

$$\begin{aligned} l_i &= \max_{0<j<i} l_j + 1, \text{ where } (s_j < s_i), \\ l_0 &= 0 \end{aligned}$$

These values define the length of the longest increasing sequence ending at each number. The length of the longest increasing subsequence of the entire permutation is given by $\max_{1 \leq i \leq n} l_i$, since the winning sequence will have to end somewhere.

Here is the table associated with our previous example:

Sequence s_i	2	4	3	5	1	7	6	9	8
Length l_i	1	2	3	3	1	4	4	5	5
Predecessor p_i	–	1	1	2	–	4	4	6	6

What auxiliary information will we need to store to reconstruct the actual sequence instead of its length? For each element s_i, we will store its *predecessor*—the index p_i of the element that appears immediately before s_i in the longest increasing sequence ending at s_i. Since all of these pointers go towards the left, it is a simple matter to start from the last value of the longest sequence and follow the pointers so as to reconstruct the other items in the sequence.

What is the time complexity of this algorithm? Each one of the n values of l_i is computed by comparing s_i against (up to) $i - 1 \leq n$ values to the left of it, so this analysis gives a total of $O(n^2)$ time. In fact, by using dictionary data structures in a clever way, we can evaluate this recurrence in $O(n \lg n)$ time. However, the simple recurrence would be easy to program and therefore is a good place to start.

Take-Home Lesson: Once you understand dynamic programming, it can be easier to work out such algorithms from scratch than to try to look them up.

8.4 War Story: Evolution of the Lobster

I caught the two graduate students lurking outside my office as I came in to work that morning. There they were, two future PhDs working in the field of high-performance computer graphics. They studied new techniques for rendering pretty computer images, but the picture they painted for me that morning was anything but pretty.

"You see, we want to build a program to morph one image into another," they explained.

"What do you mean by morph?" I asked.

"For special effects in movies, we want to construct the intermediate stages in transforming one image into another. Suppose we want to turn you into Humphrey Bogart. For this to look realistic, we must construct a bunch of in-between frames that start out looking like you and end up looking like him."

"If you can realistically turn me into Bogart, you have something," I agreed.

"But our problem is that it isn't very realistic." They showed me a dismal morph between two images. "The trouble is that we must find the right corre-

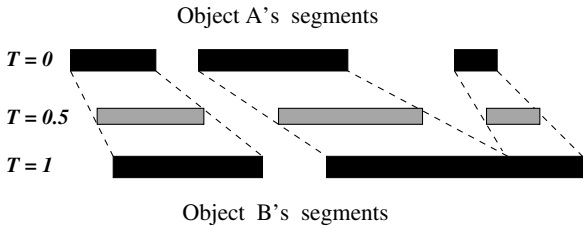


Figure 8.6: A successful alignment of two lines of pixels

spondence between features in the two images. It looks real bad when we get the correspondence wrong and try to morph a lip into an ear."

"I'll bet. So you want me to give you an algorithm for matching up lips?"

"No, even simpler. We morph each row of the initial image into the identical row of the final image. You can assume that we give you two lines of pixels, and you have to find the best possible match between the dark pixels in a row from object A to the dark pixels in the corresponding row of object B. Like this," they said, showing me images of successful matchings like Figure 8.6.

"I see," I said. "You want to match big dark regions to big dark regions and small dark regions to small dark regions."

"Yes, but only if the matching doesn't shift them too much to the left or the right. We might prefer to merge or break up regions rather than shift them too far away, since that might mean matching a chin to an eyebrow. What is the best way to do it?"

"One last question. Will you ever want to match two intervals to each other in such a way that they cross?"

"No, I guess not. Crossing intervals can't match. It would be like switching your left and right eyes."

I scratched my chin and tried to look puzzled, but I'm just not as good an actor as Bogart. I'd had a hunch about what needed to be done the instant they started talking about lines of pixels. They want to transform one array of pixels into another array, using the minimum amount of changes. That sounded like editing one string of pixels into another string, which is a classic application of dynamic programming. See Sections 8.2 and 18.4 for discussions of approximate string matching.

The fact that the intervals couldn't cross settled the issue. It meant that whenever a stretch of dark pixels from A was mapped to a stretch from B, the problem would be split into two smaller subproblems—i.e. , the pixels to the left of the match and the pixels to the right of the match. The cost of the global match would ultimately be the cost of this match plus those of matching all the pixels to the left and of matching all the pixels to the right. Constructing the optimal match on the left side is a smaller problem and hence simpler. Further, there could be only

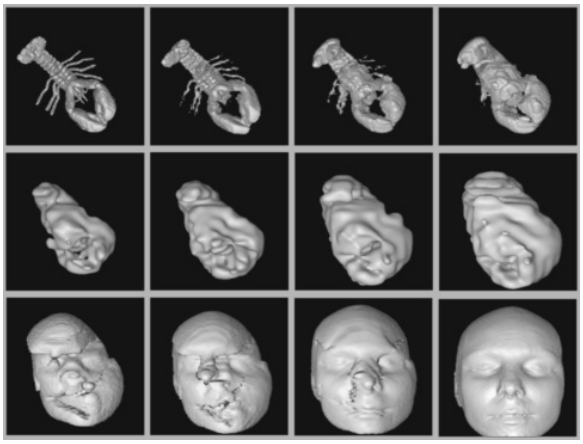


Figure 8.7: Morphing a lobster into a head via dynamic programming

$O(n^2)$ possible left subproblems, since each is completely described by the pair of one of n top pixels and one of n bottom pixels.

"Your algorithm will be based on dynamic programming," I pronounced. "However, there are several possible ways to do things, depending upon whether you want to edit pixels or runs. I would probably convert each row into a list of black pixel runs, with the runs sorted by right endpoint. Label each run with its starting position and length. You will maintain the cost of the cheapest match between the leftmost i runs and the leftmost j runs for all i and j. The possible edit operations are:

- *Full run match* – We may match top run i to run bottom j for a cost that is a function of the difference in the lengths of the two runs and their positions.

- *Merging runs* – We may match a string of consecutive top runs to a bottom run. The cost will be a function of the number of runs, their relative positions, and their lengths.

- *Splitting runs* – We may match a top run to a string of consecutive bottom runs. This is just the converse of the merge. Again, the cost will be a function of the number of runs, their relative positions, and their lengths.

"For each pair of runs (i, j) and all the cases that apply, we compute the cost of the edit operation and add to the (already computed and stored) edit cost to the left of the start of the edit. The cheapest of these cases is what we will take for the cost of $c[i, j]$."

The pair of graduate students scribbled this down, then frowned. "So we will have a cost measure for matching two runs as a function of their lengths and positions. How do we decide what the relative costs should be?"

"That is your business. The dynamic programming serves to optimize the matchings *once* you know the cost functions. It is up to your aesthetic sense to decide the penalties for line length changes and offsets. My recommendation is that you implement the dynamic programming and try different penalty values on each of several different images. Then, pick the setting that seems to do what you want."

They looked at each other and smiled, then ran back into the lab to implement it. Using dynamic programming to do their alignments, they completed their morphing system. It produced the images in Figure 8.7, morphing a lobster into a man. Unfortunately, they never got around to turning me into Humphrey Bogart.

8.5 The Partition Problem

Suppose that three workers are given the task of scanning through a shelf of books in search of a given piece of information. To get the job done fairly and efficiently, the books are to be partitioned among the three workers. To avoid the need to rearrange the books or separate them into piles, it is simplest to divide the shelf into three regions and assign each region to one worker.

But what is the fairest way to divide up the shelf? If all books are the same length, the job is pretty easy. Just partition the books into equal-sized regions,

100 100 100 | 100 100 100 | 100 100 100

so that everyone has 300 pages to deal with.

But what if the books are not the same length? Suppose we used the same partition when the book sizes looked like this:

100 200 300 | 400 500 600 | 700 800 900

I, would volunteer to take the first section, with only 600 pages to scan, instead of the last one, with 2,400 pages. The fairest possible partition for this shelf would be

100 200 300 400 500 | 600 700 | 800 900

where the largest job is only 1,700 pages and the smallest job 1,300.

In general, we have the following problem:

Problem: Integer Partition without Rearrangement
Input: An arrangement S of nonnegative numbers $\{s_1, \ldots, s_n\}$ and an integer k.
Output: Partition S into k or fewer ranges, to minimize the maximum sum over all the ranges, without reordering any of the numbers.

This so-called *linear partition* problem arises often in parallel process. We seek to balance the work done across processors to minimize the total elapsed run time.

The bottleneck in this computation will be the processor assigned the most work. Indeed, the war story of Section 7.10 (page 268) revolves around a botched solution to this problem.

Stop for a few minutes and try to find an algorithm to solve the linear partition problem.

A novice algorist might suggest a heuristic as the most natural approach to solving the partition problem. Perhaps they would compute the average size of a partition, $\sum_{i=1}^{n} s_i/k$, and then try to insert the dividers to come close to this average. However, such heuristic methods are doomed to fail on certain inputs because they do not systematically evaluate all possibilities.

Instead, consider a recursive, exhaustive search approach to solving this problem. Notice that the kth partition starts right after we placed the $(k-1)$st divider. Where can we place this last divider? Between the ith and $(i+1)$st elements for some i, where $1 \leq i \leq n$. What is the cost of this? The total cost will be the larger of two quantities—(1) the cost of the last partition $\sum_{j=i+1}^{n} s_j$, and (2) the cost of the largest partition formed to the left of i. What is the size of this left partition? To minimize our total, we want to use the $k-2$ remaining dividers to partition the elements $\{s_1, \ldots, s_i\}$ as equally as possible. *This is a smaller instance of the same problem, and hence can be solved recursively!*

Therefore, let us define $M[n,k]$ to be the minimum possible cost over all partitionings of $\{s_1, \ldots, s_n\}$ into k ranges, where the cost of a partition is the largest sum of elements in one of its parts. Thus defined, this function can be evaluated:

$$M[n,k] = \min_{i=1}^{n} \max(M[i,k-1], \sum_{j=i+1}^{n} s_j)$$

We must specify the boundary conditions of the recurrence relation. These boundary conditions always settle the smallest possible values for each of the arguments of the recurrence. For this problem, the smallest reasonable value of the first argument is $n = 1$, meaning that the first partition consists of a single element. We can't create a first partition smaller than s_1 regardless of how many dividers are used. The smallest reasonable value of the second argument is $k = 1$, implying that we do not partition S at all. In summary:

$$M[1,k] = s_1, \text{ for all } k > 0 \text{ and,}$$

$$M[n,1] = \sum_{i=1}^{n} s_i$$

By definition, this recurrence must return the size of the optimal partition. How long does it take to compute this when we store the partial results? A total of $k \cdot n$ cells exist in the table. How much time does it take to compute the result

$M[n', k']$? Calculating this quantity involves finding the minimum of n' quantities, each of which is the maximum of the table lookup and a sum of at most n' elements. If filling each of kn boxes takes at most n^2 time per box, the total recurrence can be computed in $O(kn^3)$ time.

The evaluation order computes the smaller values before the bigger values, so that each evaluation has what it needs waiting for it. Full details are provided in the code below:

```
partition(int s[], int n, int k)
{
        int m[MAXN+1][MAXK+1];                  /* DP table for values */
        int d[MAXN+1][MAXK+1];                  /* DP table for dividers */
        int p[MAXN+1];                          /* prefix sums array */
        int cost;                               /* test split cost */
        int i,j,x;                              /* counters */

        p[0] = 0;                               /* construct prefix sums */
        for (i=1; i<=n; i++) p[i]=p[i-1]+s[i];

        for (i=1; i<=n; i++) m[i][1] = p[i];    /* initialize boundaries */
        for (j=1; j<=k; j++) m[1][j] = s[1];

        for (i=2; i<=n; i++)                    /* evaluate main recurrence */
                for (j=2; j<=k; j++) {
                        m[i][j] = MAXINT;
                        for (x=1; x<=(i-1); x++) {
                                cost = max(m[x][j-1], p[i]-p[x]);
                                if (m[i][j] > cost) {
                                        m[i][j] = cost;
                                        d[i][j] = x;
                                }
                        }
                }

        reconstruct_partition(s,d,n,k);         /* print book partition */
}
```

This implementation above, in fact, runs faster than advertised. Our original analysis assumed that it took $O(n^2)$ time to update each cell of the matrix. This is because we selected the best of up to n possible points to place the divider, each of which requires the sum of up to n possible terms. In fact, it is easy to avoid the need to compute these sums by storing the set of n prefix sums $p[i] = \sum_{k=1}^{i} s_k$,

M	k		
n	1	2	3
1	1	1	1
1	2	1	1
1	3	2	1
1	4	2	2
1	5	3	2
1	6	3	2
1	7	4	3
1	8	4	3
1	9	5	3

D	k		
n	1	2	3
1	–	–	–
1	–	1	1
1	–	1	2
1	–	2	2
1	–	2	3
1	–	3	4
1	–	3	4
1	–	4	5
1	–	4	6

M	k		
n	1	2	3
1	1	1	1
2	3	2	2
3	6	3	3
4	10	6	4
5	15	9	6
6	21	11	9
7	28	15	11
8	36	21	15
9	45	24	17

D	k		
n	1	2	3
1	–	–	–
2	–	1	1
3	–	2	2
4	–	3	3
5	–	3	4
6	–	4	5
7	–	5	6
8	–	5	6
9	–	6	7

Figure 8.8: Dynamic programming matrices M and D for two input instances. Partitioning $\{1,1,1,1,1,1,1,1,1\}$ into $\{\{1,1,1\},\{1,1,1\},\{1,1,1\}\}$ (l) . Partitioning $\{1,2,3,4,5,6,7,8,9\}$ into $\{\{1,2,3,4,5\},\{6,7\},\{8,9\}\}$ (r).

since $\sum_{k=i}^{j} s_k = p[j] - p[k]$. This enables us to evaluate the recurrence in linear time per cell, yielding an $O(kn^2)$ algorithm.

By studying the recurrence relation and the dynamic programming matrices of Figure 8.8, you should be able to convince yourself that the final value of $M(n,k)$ will be the cost of the largest range in the optimal partition. For most applications, however, what we need is the actual partition that does the job. Without it, all we are left with is a coupon with a great price on an out-of-stock item.

The second matrix, D, is used to reconstruct the optimal partition. Whenever we update the value of $M[i,j]$, we record which divider position was required to achieve that value. To reconstruct the path used to get to the optimal solution, we work backward from $D[n,k]$ and add a divider at each specified position. This backwards walking is best achieved by a recursive subroutine:

```
reconstruct_partition(int s[],int d[MAXN+1][MAXK+1], int n, int k)
{
        if (k==1)
                print_books(s,1,n);
        else {
                reconstruct_partition(s,d,d[n][k],k-1);
                print_books(s,d[n][k]+1,n);
        }
}


print_books(int s[], int start, int end)
{
        int i;                  /* counter */

        for (i=start; i<=end; i++) printf(" %d ",s[i]);
        printf("\n");
}
```

sentence ::= noun–phrase
verb–phrase
noun–phrase ::= article noun
verb–phrase ::= verb noun–phrase
article ::= *the, a*
noun ::= *cat, milk*
verb ::= *drank*

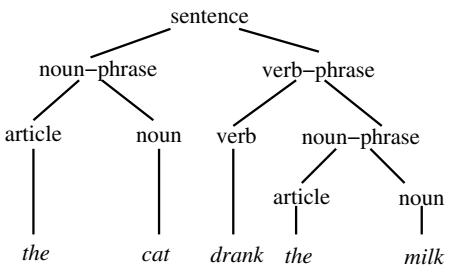


Figure 8.9: A context-free grammar (l) with an associated parse tree (r)

8.6 Parsing Context-Free Grammars

Compilers identify whether the given program is legal in the programming language, and reward you with syntax errors if not. This requires a precise description of the language syntax typically given by a *context-free grammar* as shown in Figure 8.9(l). Each *rule* or *production* of the grammar defines an interpretation for the named symbol on the left side of the rule as a sequence of symbols on the right side of the rule. The right side can be a combination of *nonterminals* (themselves defined by rules) or *terminal* symbols defined simply as strings, such as "the", "a", "cat", "milk", and "drank."

Parsing a given text string S according to a given context-free grammar G is the algorithmic problem of constructing a *parse tree* of rule substitutions defining S as a single nonterminal symbol of G. Figure 8.9(r) gives the parse tree of a simple sentence using our sample grammar.

Parsing seemed like a horribly complicated subject when I took a compilers course as a graduate student. But, a friend easily explained it to me over lunch a few years ago. The difference is that I now understand dynamic programming much better than when I was a student.

We assume that the text string S has length n while the grammar G itself is of constant size. This is fair, since the grammar defining a particular programming language (say C or Java) is of fixed length regardless of the size of the program we are trying to compile.

Further, we assume that the definitions of each rule are in *Chomsky normal form*. This means that the right sides of every nontrivial rule consists of (a) exactly two nonterminals, i.e. $X \rightarrow YZ$, or (b) exactly one terminal symbol, $X \rightarrow \alpha$. Any context-free grammar can be easily and mechanically transformed into Chomsky normal form by repeatedly shortening long right-hand sides at the cost of adding extra nonterminals and productions. Thus, there is no loss of generality with this assumption.

So how can we efficiently parse a string S using a context-free grammar where each interesting rule consists of two nonterminals? The key observation is that the rule applied at the root of the parse tree (say $X \rightarrow YZ$) splits S at some position i such that the left part of the string ($S[1, i]$) must be *generated* by nonterminal Y, and the right part ($S[i + 1, n]$) generated by Z.

This suggests a dynamic programming algorithm, where we keep track of all of the nonterminals generated by each substring of S. Define $M[i, j, X]$ to be a boolean function that is true iff substring $S[i, j]$ is generated by nonterminal X. This is true if there exists a production $X \rightarrow YZ$ and breaking point k between i and j such that the left part generates Y and the right part Z. In other words,

$$M[i, j, X] = \bigvee_{(X \rightarrow YZ) \in G} \left(\bigvee_{i=k}^{j} M[i, k, Y] \cdot M[k + 1, j, Z] \right)$$

where $\vee$ denotes the logical *or* over all productions and split positions, and $\cdot$ denotes the logical *and* of two boolean values.

The one-character terminal symbols define the boundary conditions of the recurrence. In particular, $M[i, i, X]$ is true iff there exists a production $X \rightarrow \alpha$ such that $S[i] = \alpha$.

What is the complexity of this algorithm? The size of our state-space is $O(n^2)$, as there are $n(n + 1)/2$ substrings defined by (i, j) pairs. Multiplying this by the number of nonterminals (say v) has no impact on the big-Oh, because the grammar was defined to be of constant size. Evaluating the value $M[i, j, X]$ requires testing all intermediate values k, so it takes $O(n)$ in the worst case to evaluate each of the $O(n^2)$ cells. This yields an $O(n^3)$ or cubic-time algorithm for parsing.

Stop and Think: Parsimonious Parserization

Problem: Programs often contain trivial syntax errors that prevent them from compiling. Given a context-free grammar G and input string S, find the smallest number of character substitutions you must make to S so that the resulting string is accepted by G.

Solution: This problem seemed extremely difficult when I first encountered it. But on reflection, it seemed like a very general version of edit distance, which is addressed naturally by dynamic programming. Parsing initially sounded hard, too, but fell to the same technique. Indeed, we can solve the combined problem by generalizing the recurrence relation we used for simple parsing.

Define $M'[i, j, X]$ to be an *integer* function that reports the minimum number of changes to substring $S[i, j]$ so it can be generated by nonterminal X. This symbol will be generated by some production $x \rightarrow yz$. Some of the changes to s may be

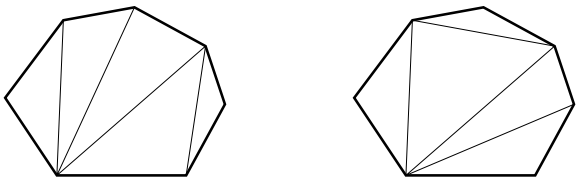


Figure 8.10: Two different triangulations of a given convex seven-gon

to the left of the breaking point and some to the right, but all we care about is minimizing the sum. In other words,

$$M'[i,j,X] = \min_{(X \to YZ) \in G} \left(\min_{i=k}^{j} M'[i,k,Y] + M'[k+1,j,Z] \right)$$

The boundary conditions also change mildly. If there exists a production $X \to \alpha$, the cost of matching at position i depends on the contents of $S[i]$, where $S[i] = \alpha$, $M[i,i,X] = 0$. Otherwise, it is one substitution away, so $M[i,i,X] = 1$ if $S[i] \neq \alpha$. If the grammar does not have a production of the form $X \to \alpha$, there is no way to substitute a single character string into something generating X, so $M[i,i,X] = \infty$ for all i. ∎

8.6.1 Minimum Weight Triangulation

The same basic recurrence relation encountered in the parsing algorithm above can also be used to solve an interesting computational geometry problem. A *triangulation* of a polygon $P = \{v_1, \ldots, v_n, v_1\}$ is a set of nonintersecting diagonals that partitions the polygon into triangles. We say that the *weight* of a triangulation is the sum of the lengths of its diagonals. As shown in Figure 8.10, any given polygon may have many different triangulations. We seek to find its minimum weight triangulation for a given polygon p. Triangulation is a fundamental component of most geometric algorithms, as discussed in Section 17.3 (page 572).

To apply dynamic programming, we need a way to carve up the polygon into smaller pieces. Observe that every edge of the input polygon must be involved in exactly one triangle. Turning this edge into a triangle means identifying the third vertex, as shown in Figure 8.11. Once we find the correct connecting vertex, the polygon will be partitioned into two smaller pieces, both of which need to be triangulated optimally. Let $T[i,j]$ be the cost of triangulating from vertex v_i to vertex v_j, ignoring the length of the chord d_{ij} from v_i to v_j. The latter clause avoids double counting these internal chords in the following recurrence:

$$T[i,j] \quad = \quad \min_{k=i+1}^{j-1} (T[i,k] + T[k,j] + d_{ik} + d_{kj})$$

The basis condition applies when i and j are immediate neighbors, as $T[i,i+1] = 0$.

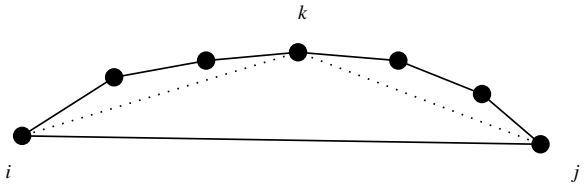


Figure 8.11: Selecting the vertex k to pair with an edge (i, j) of the polygon

Since the number of vertices in each subrange of the right side of the recurrence is smaller than that on the left side, evaluation can proceed in terms of the gap size from i to j:

```
Minimum-Weight-Triangulation(P)
    for i = 1 to n − 1 do T[i, i + 1] = 0
    for gap = 2 to n − 1
        for i = 1 to n − gap do
            j = i + gap
            T[i, j] = min_{k=i+1}^{j−1}(T[i, k] + T[k, j] + d_{P_i,P_k} + d_{P_k,P_j})
    return T[1, n]
```

There are $\binom{n}{2}$ values of T, each of which takes $O(j - i)$ time if we evaluate the sections in order of increasing size. Since $j - i = O(n)$, complete evaluation takes $O(n^3)$ time and $O(n^2)$ space.

What if there are points in the interior of the polygon? Then dynamic programming does not apply in the same way, because triangulation edges do not necessarily cut the boundary into two distinct pieces as before. Instead of only $\binom{n}{2}$ possible subregions, the number of subregions now grows exponentially. In fact, the more general version of this problem is known to be NP-complete.

Take-Home Lesson: For any optimization problem on left-to-right objects, such as characters in a string, elements of a permutation, points around a polygon, or leaves in a search tree, dynamic programming likely leads to an efficient algorithm to find the optimal solution.

8.7 Limitations of Dynamic Programming: TSP

Dynamic programming doesn't always work. It is important to see why it can fail, to help avoid traps leading to incorrect or inefficient algorithms.

Our algorithmic poster child will once again be the traveling salesman, where we seek the shortest tour visiting all the cities in a graph. We will limit attention here to an interesting special case:

Problem: Longest Simple Path
Input: A weighted graph G, with specified start and end vertices s and t.
Output: What is the most expensive path from s to t that does not visit any vertex more than once?

This problem differs from TSP in two quite unimportant ways. First, it asks for a path instead of a closed tour. This difference isn't substantial: we get a closed tour by simply including the edge (t, s). Second, it asks for the most expensive path instead of the least expensive tour. Again this difference isn't very significant: it encourages us to visit as many vertices as possible (ideally all), just as in TSP. The big word in the problem statement is *simple*, meaning we are not allowed to visit any vertex more than once.

For *unweighted* graphs (where each edge has cost 1), the longest possible simple path from s to t is $n - 1$. Finding such *Hamiltonian paths* (if they exist) is an important graph problem, discussed in Section 16.5 (page 538).

8.7.1 When are Dynamic Programming Algorithms Correct?

Dynamic programming algorithms are only as correct as the recurrence relations they are based on. Suppose we define $LP[i, j]$ as a function denoting the length of the longest simple path from i to j. Note that the longest simple path from i to j had to visit some vertex x right before reaching j. Thus, the last edge visited must be of the form (x, j). This suggests the following recurrence relation to compute the length of the longest path, where $c(x, j)$ is the cost/weight of edge (x, j):

$$LP[i, j] = \max_{(x,j) \in E} LP[i, x] + c(x, j)$$

The idea seems reasonable, but can you see the problem? I see at least two of them.

First, this recurrence does nothing to enforce simplicity. How do we know that vertex j has not appeared previously on the longest simple path from i to x? If it did, adding the edge (x, j) will create a cycle. To prevent such a thing, we must define a different recursive function that explicitly remembers where we have been. Perhaps we could define $LP'[i, j, k]$ to be the function denoting the length of the longest path from i to j avoiding vertex k? This would be a step in the right direction, but still won't lead to a viable recurrence.

A second problem concerns evaluation order. What can you evaluate first? Because there is no left-to-right or smaller-to-bigger ordering of the vertices on the graph, it is not clear what the *smaller* subprograms are. Without such an ordering, we get are stuck in an infinite loop as soon as we try to do anything.

Dynamic programming can be applied to any problem that observes the *principle of optimality*. Roughly stated, this means that partial solutions can be optimally extended with regard to the *state* after the partial solution, instead of the specifics of the partial solution itself. For example, in deciding whether to extend an approximate string matching by a substitution, insertion, or deletion, we did not need to

know which sequence of operations had been performed to date. In fact, there may be several different edit sequences that achieve a cost of C on the first p characters of pattern P and t characters of string T. Future decisions are made based on the *consequences* of previous decisions, not the actual decisions themselves.

Problems do not satisfy the principle of optimality when the specifics of the operations matter, as opposed to just the cost of the operations. Such would be the case with a form of edit distance where we are not allowed to use combinations of operations in certain particular orders. Properly formulated, however, many combinatorial problems respect the principle of optimality.

8.7.2 When are Dynamic Programming Algorithms Efficient?

The running time of any dynamic programming algorithm is a function of two things: (1) number of partial solutions we must keep track of, and (2) how long it take to evaluate each partial solution. The first issue—namely the size of the state space—is usually the more pressing concern.

In all of the examples we have seen, the partial solutions are completely described by specifying the stopping *places* in the input. This is because the combinatorial objects being worked on (strings, numerical sequences, and polygons) have an implicit order defined upon their elements. This order cannot be scrambled without completely changing the problem. Once the order is fixed, there are relatively few possible stopping places or states, so we get efficient algorithms.

When the objects are not firmly ordered, however, we get an exponential number of possible partial solutions. Suppose the state of our partial solution is entire path P taken from the start to end vertex. Thus $LP[i, j, P]$ denotes the longest simple path from i to j, where P is the exact sequence of intermediate vertices between i and j on this path. The following recurrence relation works to compute this, where $P + x$ denotes appending x to the end of P:

$$LP[i, j, P + x] = \max_{(x,j) \in E, x, j \notin P} LP[i, x, P] + c(x, j)$$

This formulation is correct, but how efficient is it? The path P consists of an ordered sequence of up to $n - 3$ vertices. There can be up to $(n - 3)!$ such paths! Indeed, this algorithm is really using combinatorial search (*a la* backtracking) to construct all the possible intermediate paths. In fact, the max is somewhat misleading, as there can only be one value of x and one value of P to construct the state $LP[i, j, P + x]$.

We can do something better with this idea, however. Let $LP'[i, j, S]$ denote the longest simple path from i to j, where the intermediate vertices on this path are exactly those in the *subset* S. Thus, if $S = \{a, b, c\}$, there are exactly six paths consistent with S: *iabcj*, *iacbj*, *ibacj*, *ibcaj*, *icabj*, and *icbaj*. This state space is at most 2^n, and thus smaller than enumerating the paths. Further, this function can be evaluated using the following recurrence relation:

$$LP'[i, j, S \cup \{x\}] = \max_{(x,j) \in E, x, j \notin S} LP[i, x, S] + c(x, j)$$

where $S \cup \{x\}$ denotes unioning S with x.

The longest simple path from i to j can then be found by maximizing over all possible intermediate vertex subsets:

$$LP[i,j] = \max_S LP'[i,j,S]$$

There are only 2^n subsets of n vertices, so this is a big improvement over enumerating all $n!$ tours. Indeed, this method could certainly be used to solve TSPs for up to thirty vertices or so, where $n = 20$ would be impossible using the $O(n!)$ algorithm. Still, dynamic programming is most effective on well-ordered objects.

> *Take-Home Lesson:* Without an inherent left-to-right ordering on the objects, dynamic programming is usually doomed to require exponential space and time.

8.8 War Story: What's Past is Prolog

"But our heuristic works very, very well in practice." My colleague was simultaneously boasting and crying for help.

Unification is the basic computational mechanism in logic programming languages like Prolog. A Prolog program consists of a set of rules, where each rule has a head and an associated action whenever the rule head matches or unifies with the current computation.

An execution of a Prolog program starts by specifying a goal, say $p(a, X, Y)$, where a is a constant and X and Y are variables. The system then systematically matches the head of the goal with the head of each of the rules that can be *unified* with the goal. Unification means binding the variables with the constants, if it is possible to match them. For the nonsense program below, $p(X, Y, a)$ unifies with either of the first two rules, since X and Y can be bound to match the extra characters. The goal $p(X, X, a)$ would only match the first rule, since the variable bound to the first and second positions must be the same.

$$p(a,a,a) := h(a);$$
$$p(b,a,a) := h(a) * h(b);$$
$$p(c,b,b) := h(b) + h(c);$$
$$p(d,b,b) := h(d) + h(b);$$

"In order to speed up unification, we want to preprocess the set of rule heads so that we can quickly determine which rules match a given goal. We must organize the rules in a trie data structure for fast unification."

Tries are extremely useful data structures in working with strings, as discussed in Section 12.3 (page 377). Every leaf of the trie represents one string. Each node on the path from root to leaf is labeled with exactly one character of the string, with the ith node of the path corresponding to the ith character of the string.

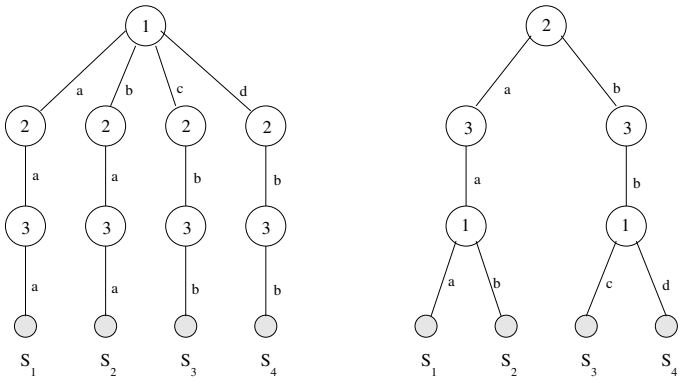


Figure 8.12: Two different tries for the same set of rule heads.

"I agree. A trie is a natural way to represent your rule heads. Building a trie on a set of strings of characters is straightforward: just insert the strings starting from the root. So what is your problem?" I asked.

"The efficiency of our unification algorithm depends very much on minimizing the number of edges in the trie. Since we know all the rules in advance, we have the freedom to reorder the character positions in the rules. Instead of the root node always representing the first argument in the rule, we can choose to have it represent the third argument. We would like to use this freedom to build a minimum-size trie for a set of rules."

He showed me the example in Figure 8.12. A trie constructed according to the original string position order $(1, 2, 3)$ uses a total of 12 edges. However, by permuting the character order to $(2, 3, 1)$, we can obtain a trie with only 8 edges.

"Interesting..." I started to reply before he cut me off again.

"There's one other constraint. We must keep the leaves of the trie ordered, so that the leaves of the underlying tree go left-to-right in the same order as the rules appear on the page."

"But why must you keep the leaves of the trie in the given order?" I asked.

"The order of rules in Prolog programs is very, very important. If you change the order of the rules, the program returns different results."

Then came my mission.

"We have a greedy heuristic for building good, but not optimal, tries based on picking as the root the character position that minimizes the degree of the root. In other words, it picks the character position that has the smallest number of distinct characters in it. This heuristic works very, very well in practice. But we need you to prove that finding the best trie is NP-complete so our paper is, well, complete."

I agreed to try to prove the hardness of the problem, and chased him from my office. The problem did seem to involve some nontrivial combinatorial optimization to build the minimal tree, but I couldn't see how to factor the left-to-right order of the rules into a hardness proof. In fact, I couldn't think of any NP-complete problem that had such a left-right ordering constraint. After all, if a given set of rules contained a character position in common to all the rules, this character position must be probed first in any minimum-size tree. Since the rules were ordered, each node in the subtree must represent the root of a run of consecutive rules. Thus there were only $\binom{n}{2}$ possible nodes to choose from for this tree....

Bingo! That settled it.

The next day I went back to the professor and told him. "I can't prove that your problem is NP-complete. But how would you feel about an efficient dynamic programming algorithm to find the best trie!" It was a pleasure watching his frown change to a smile as the realization took hold. An efficient algorithm to compute what he needed was infinitely better than a proof saying you couldn't do it!

My recurrence looked something like this. Suppose that we are given n ordered rule heads $s_1, \ldots, s_n$, each with m arguments. Probing at the pth position, $1 \leq p \leq m$, partitioned the rule heads into runs $R_1, \ldots, R_r$, where each rule in a given run $R_x = s_i, \ldots, s_j$ had the same character value of $s_i[p]$. The rules in each run must be consecutive, so there are only $\binom{n}{2}$ possible runs to worry about. The cost of probing at position p is the cost of finishing the trees formed by each created run, plus one edge per tree to link it to probe p:

$$C[i,j] = \min_{p=1}^{m} \sum_{k=1}^{r} (C[i_k, j_k] + 1)$$

A graduate student immediately set to work implementing this algorithm to compare with their heuristic. On many inputs, the optimal and greedy algorithms constructed the exact same trie. However, for some examples, dynamic programming gave a 20% performance improvement over greedy—i.e. , 20% better than very, very well in practice. The run time spent in doing the dynamic programming was a bit larger than with greedy, but in compiler optimization you are always happy to trade off a little extra compilation time for better execution time in the performance of your program. Is a 20% improvement worth this effort? That depends upon the situation. How useful would you find a 20% increase in your salary?

The fact that the rules had to remain ordered was the crucial property that we exploited in the dynamic programming solution. Indeed, without it I was able to prove that the problem *was* NP-complete, something we put in the paper to make it complete.

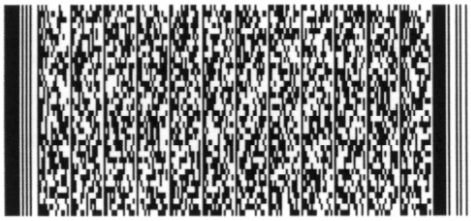


Figure 8.13: A two-dimensional bar-code label of the Gettysburg Address using PDF-417.

Take-Home Lesson: The global optimum (found, for example, using dynamic programming) is often noticeably better than the solution found by typical heuristics. How important this improvement is depends on your application, but it can never hurt.

8.9 War Story: Text Compression for Bar Codes

Ynjiun waved his laser wand over the torn and crumpled fragments of a bar code label. The system hesitated for a few seconds, then responded with a pleasant *blip* sound. He smiled at me in triumph. "Virtually indestructible."

I was visiting the research laboratories of Symbol Technologies, the world's leading manufacturer of bar code scanning equipment. Next time you are in the checkout line at a grocery store, check to see what type of scanning equipment they are using. Likely it will say Symbol on the housing.

Although we take bar codes for granted, there is a surprising amount of technology behind them. Bar codes exist because conventional optical character recognition (OCR) systems are not sufficiently reliable for inventory operations. The bar code symbology familiar to us on each box of cereal or pack of gum encodes a ten-digit number with enough error correction that it is virtually impossible to scan the wrong number, even if the can is upside-down or dented. Occasionally, the cashier won't be able to get a label to scan at all, but once you hear that *blip* you know it was read correctly.

The ten-digit capacity of conventional bar code labels provides room enough only to store a single ID number in a label. Thus any application of supermarket bar codes must have a database mapping 11141-47011 to a particular size and brand of soy sauce. The holy grail of the bar code world has long been the development of higher-capacity bar code symbologies that can store entire documents, yet still be read reliably.

"PDF-417 is our new, two-dimensional bar code symbology," Ynjiun explained. A sample label is shown in Figure 8.13.

"How much data can you fit in a typical one-inch square label?" I asked him.

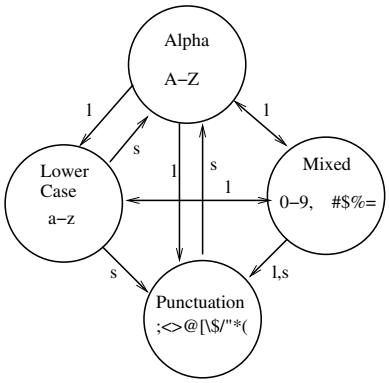


Figure 8.14: Mode switching in PDF-417

"It depends upon the level of error correction we use, but about 1,000 bytes. That's enough for a small text file or image," he said.

"Interesting. You will probably want to use some data compression technique to maximize the amount of text you can store in a label." See Section 18.5 (page 637) for a discussion of standard data compression algorithms.

"We do incorporate a data compaction method," he explained. "We think we understand the different types of files our customers will want to make labels for. Some files will be all in uppercase letters, while others will use mixed-case letters and numbers. We provide four different text modes in our code, each with a different subset of alphanumeric characters available. We can describe each character using only five bits as long as we stay within a mode. To switch modes, we issue a mode switch command first (taking an extra five bits) and then the new character code."

"I see. So you designed the mode character sets to minimize the number of mode switch operations on typical text files." The modes are illustrated in Figure 8.14.

"Right. We put all the digits in one mode and all the punctuation characters in another. We also included both mode *shift* and mode *latch* commands. In a mode shift, we switch into a new mode just for the next character, say to produce a punctuation mark. This way, we don't pay a cost for returning back to text mode after a period. Of course, we can also latch permanently into a different mode if we will be using a run of several characters from there."

"Wow!" I said. "With all of this mode switching going on, there must be many different ways to encode any given text as a label. How do you find the smallest of such encoding."

"We use a greedy algorithm. We look a few characters ahead and then decide which mode we would be best off in. It works fairly well."

I pressed him on this. "How do you know it works fairly well? There might be significantly better encodings that you are simply not finding."

"I guess I don't know. But it's probably NP-complete to find the optimal coding." Ynjiun's voice trailed off. "Isn't it?"

I started to think. Every encoding started in a given mode and consisted of a sequence of intermixed character codes and mode shift/latch operations. At any given position in the text, we could output the next character code (if it was available in our current mode) or decide to shift. As we moved from left to right through the text, our current state would be completely reflected by our current character position and current mode state. For a given position/mode pair, we would have been interested in the cheapest way of getting there, over all possible encodings getting to this point....

My eyes lit up so bright they cast shadows on the walls.

"The optimal encoding for any given text in PDF-417 can be found using dynamic programming. For each possible mode $1 \leq m \leq 4$, and each character position $1 \leq i \leq n$, we will maintain the cheapest encoding found of the first i characters ending in mode m. Our next move from each mode/position is either match, shift, or latch, so there are only a few possible operations to consider."

Basically,

$$M[i,j] = \min_{1 \leq m \leq 4} (M[i-1,m] + c(S_i, m, j))$$

where $c(S_i, m, j)$ is the cost of encoding character S_i and switching from mode m to mode j. The cheapest possible encoding results from tracing back from $M[n,m]$, where m is the value of i that minimizes $\min_{1 \leq i \leq 4} M[n,i]$. Each of the $4n$ cells can be filled in constant time, so it takes time linear in the length of the string to find the optimal encoding.

Ynjiun was skeptical, but he encouraged us to implement an optimal encoder. A few complications arose due to weirdnesses of PDF-417 mode switching, but my student Yaw-Ling Lin rose to the challenge. Symbol compared our encoder to theirs on 13,000 labels and concluded that dynamic programming lead to an 8% tighter encoding on average. This was significant, because no one wants to waste 8% of their potential storage capacity, particularly in an environment where the capacity is only a few hundred bytes. For certain applications, this 8% margin permitted one bar code label to suffice where previously two had been required. Of course, an 8% *average* improvement meant that it did much better than that on certain labels. While our encoder took slightly longer to run than the greedy encoder, this was not significant, since the bottleneck would be the time needed to print the label anyway.

Our observed impact of replacing a heuristic solution with the global optimum is probably typical of most applications. Unless you really botch your heuristic, you are probably going to get a decent solution. Replacing it with an optimal result, however, usually gives a small but nontrivial improvement, which can have pleasing consequences for your application.

Chapter Notes

Bellman [Bel58] is credited with developing the technique of dynamic programming. The edit distance algorithm is originally due to Wagner and Fischer [WF74]. A faster algorithm for the book partition problem appears in [KMS97].

The computational complexity of finding the minimum weight triangulation of disconnected point sets (as opposed to polygons) was a longstanding open problem that finally fell to Mulzer and Rote [MR06].

Techniques such as dynamic programming and backtracking searches can be used to generate worst-case efficient (although still non-polynomial) algorithms for many NP-complete problems. See Woeginger [Woe03] for a nice survey of such techniques.

The morphing system that was the subject of the war story in Section 8.4 (page 291) is described in [HWK94]. See our paper [DRR+95] for more on the Prolog trie minimization problem, subject of the war story of Section 8.8 (page 304). Two-dimensional bar codes, subject of the war story in Section 8.9 (page 307), were developed largely through the efforts of Theo Pavlidis and Ynjiun Wang at Stony Brook [PSW92].

The dynamic programming algorithm presented for parsing is known as the *CKY* algorithm after its three independent inventors (Cocke, Kasami, and Younger) [You67]. The generalization of parsing to edit distance is due to Aho and Peterson [AP72].

8.10 Exercises

Edit Distance

8-1. *[3]* Typists often make transposition errors exchanging neighboring characters, such as typing "setve" when you mean "steve." This requires two substitutions to fix under the conventional definition of edit distance.

Incorporate a swap operation into our edit distance function, so that such neighboring transposition errors can be fixed at the cost of one operation.

8-2. *[4]* Suppose you are given three strings of characters: X, Y, and Z, where $|X| = n$, $|Y| = m$, and $|Z| = n + m$. Z is said to be a *shuffle* of X and Y iff Z can be formed by interleaving the characters from X and Y in a way that maintains the left-to-right ordering of the characters from each string.

(a) Show that *cchocohilaptes* is a shuffle of *chocolate* and *chips*, but *chocochilatspe* is not.

(b) Give an efficient dynamic-programming algorithm that determines whether Z is a shuffle of X and Y. Hint: the values of the dynamic programming matrix you construct should be Boolean, not numeric.

8-3. *[4]* The longest common *substring* (not subsequence) of two strings X and Y is the longest string that appears as a run of consecutive letters in both strings. For example, the longest common substring of *photograph* and *tomography* is *ograph*.

(a) Let $n = |X|$ and $m = |Y|$. Give a $\Theta(nm)$ dynamic programming algorithm for longest common substring based on the longest common subsequence/edit distance algorithm.

(b) Give a simpler $\Theta(nm)$ algorithm that does not rely on dynamic programming.

8-4. *[6]* The *longest common subsequence (LCS)* of two sequences T and P is the longest sequence L such that L is a subsequence of both T and P. The *shortest common supersequence (SCS)* of T and P is the smallest sequence L such that both T and P are a subsequence of L.

(a) Give efficient algorithms to find the LCS and SCS of two given sequences.

(b) Let $d(T, P)$ be the minimum edit distance between T and P when no substitutions are allowed (i.e., the only changes are character insertion and deletion). Prove that $d(T, P) = |SCS(T, P)| - |LCS(T, P)|$ where $|SCS(T, P)|$ ($|LCS(T, P)|$) is the size of the shortest SCS (longest LCS) of T and P.

Greedy Algorithms

8-5. *[4]* Let $P_1, P_2, \ldots, P_n$ be n programs to be stored on a disk with capacity D megabytes. Program P_i requires s_i megabytes of storage. We cannot store them all because $D < \sum_{i=1}^{n} s_i$

(a) Does a greedy algorithm that selects programs in order of nondecreasing s_i maximize the number of programs held on the disk? Prove or give a counterexample.

(b) Does a greedy algorithm that selects programs in order of nonincreasing order s_i use as much of the capacity of the disk as possible? Prove or give a counterexample.

8-6. *[5]* Coins in the United States are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations of $\{d_1, \ldots, d_k\}$ units. We seek an algorithm to make change of n units using the minimum number of coins for this country.

(a) The greedy algorithm repeatedly selects the biggest coin no bigger than the amount to be changed and repeats until it is zero. Show that the greedy algorithm does not always use the minimum number of coins in a country whose denominations are $\{1, 6, 10\}$.

(b) Give an efficient algorithm that correctly determines the minimum number of coins needed to make change of n units using denominations $\{d_1, \ldots, d_k\}$. Analyze its running time.

8-7. *[5]* In the United States, coins are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations of $\{d_1, \ldots, d_k\}$ units. We want to count how many distinct ways $C(n)$ there are to make change of n units. For example, in a country whose denominations are $\{1, 6, 10\}$, $C(5) = 1$, $C(6)$ to $C(9) = 2$, $C(10) = 3$, and $C(12) = 4$.

(a) How many ways are there to make change of 20 units from $\{1, 6, 10\}$?

(b) Give an efficient algorithm to compute $C(n)$, and analyze its complexity. (Hint: think in terms of computing $C(n, d)$, the number of ways to make change of n units with highest denomination d. Be careful to avoid overcounting.)

8-8. *[6]* In the *single-processor scheduling problem*, we are given a set of n jobs J. Each job i has a processing time t_i, and a deadline d_i. A feasible schedule is a permutation of the jobs such that when the jobs are performed in that order, every job is finished before its deadline. The greedy algorithm for single-processor scheduling selects the job with the earliest deadline first.

Show that if a feasible schedule exists, then the schedule produced by this greedy algorithm is feasible.

Number Problems

8-9. *[6]* The *knapsack problem* is as follows: given a set of integers $S = \{s_1, s_2, \ldots, s_n\}$, and a given target number T, find a subset of S that adds up exactly to T. For example, within $S = \{1, 2, 5, 9, 10\}$ there is a subset that adds up to $T = 22$ but not $T = 23$.

Give a correct programming algorithm for knapsack that runs in $O(nT)$ time.

8-10. *[6]* The *integer partition* takes a set of positive integers $S = s_1, \ldots, s_n$ and asks if there is a subset $I \in S$ such that

$$\sum_{i \in I} s_i = \sum_{i \notin I} s_i$$

Let $\sum_{i \in S} s_i = M$. Give an $O(nM)$ dynamic programming algorithm to solve the integer partition problem.

8-11. *[5]* Assume that there are n numbers (some possibly negative) on a circle, and we wish to find the maximum contiguous sum along an arc of the circle. Give an efficient algorithm for solving this problem.

8-12. *[5]* A certain string processing language allows the programmer to break a string into two pieces. It costs n units of time to break a string of n characters into two pieces, since this involves copying the old string. A programmer wants to break a string into many pieces, and the order in which the breaks are made can affect the total amount of time used. For example, suppose we wish to break a 20-character string after characters 3, 8, and 10. If the breaks are made in left-right order, then the first break costs 20 units of time, the second break costs 17 units of time, and the third break costs 12 units of time, for a total of 49 steps. If the breaks are made in right-left order, the first break costs 20 units of time, the second break costs 10 units of time, and the third break costs 8 units of time, for a total of only 38 steps.

Give a dynamic programming algorithm that takes a list of character positions after which to break and determines the cheapest break cost in $O(n^3)$ time.

8-13. *[5]* Consider the following data compression technique. We have a table of m text strings, each at most k in length. We want to encode a data string D of length n using as few text strings as possible. For example, if our table contains *(a,ba,abab,b)* and the data string is *bababbaababa*, the best way to encode it is *(b,abab,ba,abab,a)*—a total of five code words. Give an $O(nmk)$ algorithm to find the length of the best

encoding. You may assume that every string has at least one encoding in terms of the table.

8-14. *[5]* The traditional world chess championship is a match of 24 games. The current champion retains the title in case the match is a tie. Each game ends in a win, loss, or draw (tie) where wins count as 1, losses as 0, and draws as 1/2. The players take turns playing white and black. White has an advantage, because he moves first. The champion plays white in the first game. He has probabilities w_{w}, w_{d}, and w_{l} of winning, drawing, and losing playing white, and has probabilities b_{w}, b_{d}, and b_{l} of winning, drawing, and losing playing black.

(a) Write a recurrence for the probability that the champion retains the title. Assume that there are g games left to play in the match and that the champion needs to win i games (which may end in a 1/2).

(b) Based on your recurrence, give a dynamic programming to calculate the champion's probability of retaining the title.

(c) Analyze its running time for an n game match.

8-15. *[8]* Eggs break when dropped from great enough height. Specifically, there must be a floor f in any sufficiently tall building such that an egg dropped from the fth floor breaks, but one dropped from the $(f-1)$st floor will not. If the egg always breaks, then $f = 1$. If the egg never breaks, then $f = n + 1$.

You seek to find the critical floor f using an n-story building. The only operation you can perform is to drop an egg off some floor and see what happens. You start out with k eggs, and seek to drop eggs as few times as possible. Broken eggs cannot be reused. Let $E(k, n)$ be the minimum number of egg droppings that will always suffice.

(a) Show that $E(1, n) = n$.

(b) Show that $E(k, n) = \Theta(n^{\frac{1}{k}})$.

(c) Find a recurrence for $E(k, n)$. What is the running time of the dynamic program to find $E(k, n)$?

Graph Problems

8-16. *[4]* Consider a city whose streets are defined by an $X \times Y$ grid. We are interested in walking from the upper left-hand corner of the grid to the lower right-hand corner.

Unfortunately, the city has bad neighborhoods, whose intersections we do not want to walk in. We are given an $X \times Y$ matrix BAD, where *BAD[i,j] = "yes"* if and only if the intersection between streets i and j is in a neighborhood to avoid.

(a) Give an example of the contents of BAD such that there is no path across the grid avoiding bad neighborhoods.

(b) Give an $O(XY)$ algorithm to find a path across the grid that avoids bad neighborhoods.

(c) Give an $O(XY)$ algorithm to find the *shortest* path across the grid that avoids bad neighborhoods. You may assume that all blocks are of equal length. For partial credit, give an $O(X^2Y^2)$ algorithm.

8-17. *[5]* Consider the same situation as the previous problem. We have a city whose streets are defined by an $X \times Y$ grid. We are interested in walking from the upper left-hand corner of the grid to the lower right-hand corner. We are given an $X \times Y$ matrix BAD, where $BAD[i,j]$ = *"yes"* if and only if the intersection between streets i and j is somewhere we want to avoid.

If there were no bad neighborhoods to contend with, the shortest path across the grid would have length $(X-1)+(Y-1)$ blocks, and indeed there would be many such paths across the grid. Each path would consist of only rightward and downward moves.

Give an algorithm that takes the array BAD and returns the *number* of safe paths of length $X+Y-2$. For full credit, your algorithm must run in $O(XY)$.

Design Problems

8-18. *[4]* Consider the problem of storing n books on shelves in a library. The order of the books is fixed by the cataloging system and so cannot be rearranged. Therefore, we can speak of a book b_i, where $1 \leq i \leq n$, that has a thickness t_i and height h_i. The length of each bookshelf at this library is L.

Suppose all the books have the same height h (i.e. , $h = h_i = h_j$ for all i, j) and the shelves are all separated by a distance of greater than h, so any book fits on any shelf. The greedy algorithm would fill the first shelf with as many books as we can until we get the smallest i such that b_i does not fit, and then repeat with subsequent shelves. Show that the greedy algorithm always finds the optimal shelf placement, and analyze its time complexity.

8-19. *[6]* This is a generalization of the previous problem. Now consider the case where the height of the books is not constant, but we have the freedom to adjust the height of each shelf to that of the tallest book on the shelf. Thus the cost of a particular layout is the sum of the heights of the largest book on each shelf.

- Give an example to show that the greedy algorithm of stuffing each shelf as full as possible does not always give the minimum overall height.
- Give an algorithm for this problem, and analyze its time complexity. Hint: use dynamic programming.

8-20. *[5]* We wish to compute the laziest way to dial given n-digit number on a standard push-button telephone using two fingers. We assume that the two fingers start out on the * and # keys, and that the effort required to move a finger from one button to another is proportional to the Euclidean distance between them. Design an algorithm that computes the method of dialing that involves moving your fingers the smallest amount of total distance, where k is the number of distinct keys on the keypad ($k = 16$ for standard telephones). Try to use $O(nk^3)$ time.

8-21. *[6]* Given an array of n real numbers, consider the problem of finding the maximum sum in any contiguous subvector of the input. For example, in the array

$$\{31, -41, 59, 26, -53, 58, 97, -93, -23, 84\}$$

the maximum is achieved by summing the third through seventh elements, where $59+26+(-53)+58+97 = 187$. When all numbers are positive, the entire array is the answer, while when all numbers are negative, the empty array maximizes the total at 0.

- Give a simple, clear, and correct $\Theta(n^2)$-time algorithm to find the maximum contiguous subvector.
- Now give a $\Theta(n)$-time dynamic programming algorithm for this problem. To get partial credit, you may instead give a *correct* $O(n \log n)$ divide-and-conquer algorithm.

8-22. *[7]* Consider the problem of examining a string $x = x_1 x_2 \ldots x_n$ from an alphabet of k symbols, and a multiplication table over this alphabet. Decide whether or not it is possible to parenthesize x in such a way that the value of the resulting expression is a, where a belongs to the alphabet. The multiplication table is neither commutative or associative, so the order of multiplication matters.

	a	b	c
a	a	c	c
b	a	a	b
c	c	c	c

For example, consider the above multiplication table and the string *bbbba*. Parenthesizing it $(b(bb))(ba)$ gives a, but $((((bb)b)b)a)$ gives c.
Give an algorithm, with time polynomial in n and k, to decide whether such a parenthesization exists for a given string, multiplication table, and goal element.

8-23. *[6]* Let α and β be constants. Assume that it costs α to go left in a tree, and β to go right. Devise an algorithm that builds a tree with optimal worst case cost, given keys $k_1, \ldots, k_n$ and the probabilities that each will be searched $p_1, \ldots, p_n$.

Interview Problems

8-24. *[5]* Given a set of coin denominators, find the minimum number of coins to make a certain amount of change.

8-25. *[5]* You are given an array of n numbers, each of which may be positive, negative, or zero. Give an efficient algorithm to identify the index positions i and j to the maximum sum of the ith through jth numbers.

8-26. *[7]* Observe that when you cut a character out of a magazine, the character on the reverse side of the page is also removed. Give an algorithm to determine whether you can generate a given string by pasting cutouts from a given magazine. Assume that you are given a function that will identify the character and its position on the reverse side of the page for any given character position.

Programming Challenges

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

8-1. "Is Bigger Smarter?" – Programming Challenges 111101, UVA Judge 10131.

8-2. "Weights and Measures" – Programming Challenges 111103, UVA Judge 10154.

8-3. "Unidirectional TSP" – Programming Challenges 111104, UVA Judge 116.

8-4. "Cutting Sticks" – Programming Challenges 111105, UVA Judge 10003.

8-5. "Ferry Loading" – Programming Challenges 111106, UVA Judge 10261.

9

Intractable Problems and Approximation Algorithms

We now introduce techniques for proving that *no* efficient algorithm exists for a given problem. The practical reader is probably squirming at the notion of proving anything, and will be particularly alarmed at the idea of investing time to prove that something does not exist. Why are you better off knowing that something you don't know how to do in fact can't be done at all?

The truth is that the theory of NP-completeness is an immensely useful tool for the algorithm designer, even though all it provides are negative results. The theory of NP-completeness enables the algorithm designer to focus her efforts more productively, by revealing that the search for an efficient algorithm for this particular problem is doomed to failure. When one *fails* to show a problem is hard, that suggests there may well be an efficient algorithm to solve it. Two of the war stories in this book described happy results springing from bogus claims of hardness.

The theory of NP-completeness also enables us to identify what properties make a particular problem hard. This provides direction for us to model it in different ways or exploit more benevolent characteristics of the problem. Developing a sense for which problems are hard is an important skill for algorithm designers, and only comes from hands-on experience with proving hardness.

The fundamental concept we will use is that of *reductions* between pairs of problems, showing that the problems are really equivalent. We illustrate this idea through a series of reductions, each of which either yields an efficient algorithm or an argument that no such algorithm can exist. We also provide brief introductions to (1) the complexity-theoretic aspects of NP-completeness, one of the most fundamental notions in Computer Science, and (2) the theory of approximation algorithms, which leads to heuristics that probably return something *close* to the optimal solution.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_9,

9.1 Problems and Reductions

We have encountered several problems in this book for which we couldn't find any efficient algorithm. The theory of NP-completeness provides the tools needed to show that all these problems are on some level really the same problem.

The key idea to demonstrating the hardness of a problem is that of a *reduction*, or translation, between two problems. The following allegory of NP-completeness may help explain the idea. A bunch of kids take turns fighting each other in the schoolyard to prove how "tough" they are. Adam beats up Bill, who then beats up Chuck. So who if any among them is "tough?" The truth is that there is no way to know without an external standard. If I told you that Chuck was in fact Chuck Norris, certified tough guy, you have to be impressed—both Adam and Bill are at least as tough as he is. On the other hand, suppose I tell you it is a kindergarten school yard. No one would call me tough, but even I can take out Adam. This proves that none of the three of them should be called be tough. In this allegory, each fight represents a reduction. Chuck Norris takes on the role of satisfiability—a certifiably hard problem. The part of an inefficient algorithm with a possible shot at redemption is played by me.

Reductions are operations that convert one problem into another. To describe them, we must be somewhat rigorous in our definitions. An algorithmic *problem* is a general question, with parameters for input and conditions on what constitutes a satisfactory answer or solution. An *instance* is a problem with the input parameters specified. The difference can be made clear by an example:

Problem: The Traveling Salesman Problem (TSP)
Input: A weighted graph G.
Output: Which tour $\{v_1, v_2, ..., v_n\}$ minimizes $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$?

Any weighted graph defines an instance of TSP. Each particular *problem* has at least one minimum cost tour. The general traveling salesman *instance* asks for an algorithm to find the optimal tour for all possible instances.

9.1.1 The Key Idea

Now consider two algorithmic problems, called *Bandersnatch* and *Bo-billy*. Suppose that I gave you the following reduction/algorithm to solve the *Bandersnatch* problem:

Bandersnatch(G)
    Translate the input G to an instance Y of the Bo-billy problem.
    Call the subroutine Bo-billy on Y to solve this instance.
    Return the answer of Bo-billy(Y) as the answer to Bandersnatch(G).

This algorithm will *correctly* solve the Bandersnatch problem provided that the translation to Bo-billy always preserves the correctness of the answer. In other words, the translation has the property that for any instance of G,

$$\text{Bandersnatch}(G) = \text{Bo-billy}(Y)$$

A translation of instances from one type of problem to instances of another such that the answers are preserved is what is meant by a *reduction*.

Now suppose this reduction translates G to Y in $O(P(n))$ time. There are two possible implications:

- *If* my Bo-billy subroutine ran in $O(P'(n))$, this means I could solve the Bandersnatch problem in $O(P(n) + P'(n))$ by spending the time to translate the problem and then the time to execute the Bo-Billy subroutine.
- *If* I know that $\Omega(P'(n))$ is a lower bound on computing Bandersnatch, meaning there definitely exists no faster way to solve it, then $\Omega(P'(n) - P(n))$ *must* be a lower bound to compute Bo-billy. Why? If I could solve Bo-billy any faster, then I could violate my lower bound by solving Bandersnatch using the above reduction. This implies that there can be no way to solve Bo-billy any faster than claimed.

This first argument is Steve demonstrating the weakness of the entire schoolyard with a quick right to Adam's chin. The second highlights the Chuck Norris approach we will use to prove that problems are hard. Essentially, this reduction shows that Bo-billy is no easier than Bandersnatch. Therefore, if Bandersnatch is hard this means Bo-billy must also be hard.

We will illustrate this point by giving several problem reductions in this chapter.

> *Take-Home Lesson:* Reductions are a way to show that two problems are essentially identical. A fast algorithm (or the lack of one) for one of the problems implies a fast algorithm (or the lack of one) for the other.

9.1.2 Decision Problems

Reductions translate between problems so that their answers are identical in every problem instance. Problems differ in the *range* or *type* of possible answers. The traveling salesman problem returns a permutation of vertices as the answer, while other types of problems return numbers as answers, perhaps restricted to positive numbers or integers.

The simplest interesting class of problems have answers restricted to true and false. These are called *decision problems*. It proves convenient to reduce/translate answers between decision problems because both only allow true and false as possible answers.

Fortunately, most interesting optimization problems can be phrased as decision problems that capture the essence of the computation. For example, the traveling salesman decision problem could be defined as:

Problem: The Traveling Salesman Decision Problem
Input: A weighted graph G and integer k.
Output: Does there exist a TSP tour with cost $\leq k$?

The decision version captures the heart of the traveling salesman problem, in that if you had a fast algorithm for the decision problem, you could use it to do a binary search with different values of k and quickly hone in on the optimal solution. With a little more cleverness, you could reconstruct the actual tour permutation using a fast solution to the decision problem.

From now on we will generally talk about decision problems, because it proves easier and still captures the power of the theory.

9.2 Reductions for Algorithms

An engineer and an algorist are sitting in a kitchen. The algorist asks the engineer to boil some water, so the engineer gets up, picks up the kettle from the counter top, adds water from the sink, brings it to the burner, turns on the burner, waits for the whistling sound, and turns off the burner. Sometime later, the engineer asks the algorist to boil more water. She gets up, takes the kettle from the burner, moves it over to the counter top, and sits down. "Done." she says, "I have *reduced* the task to a solved problem."

This boiling water reduction illustrates an honorable way to generate new algorithms from old. If we can translate the input for a problem we *want to solve* into input for a problem we *know how to solve*, we can compose the translation and the solution into an algorithm for our problem.

In this section, we look at several reductions that lead to efficient algorithms. To solve problem a, we translate/reduce the a instance to an instance of b, then solve this instance using an efficient algorithm for problem b. The overall running time is the time needed to perform the reduction plus that solve the b instance.

9.2.1 Closest Pair

The *closest pair* problem asks to find the pair of numbers within a set that have the smallest difference between them. We can make it a decision problem by asking if this value is less than some threshold:

Input: A set S of n numbers, and threshold t.
Output: Is there a pair $s_i, s_j \in S$ such that $|s_i - s_j| \leq t$?

The closest pair is a simple application of sorting, since the closest pair must be neighbors after sorting. This gives the following algorithm:

CloseEnoughPair(S,t)
    Sort S.
    Is $\min_{1 \leq i < n} |s_i - s_{i+1}| \leq t$?

There are several things to note about this simple reduction.

1. The decision version captured what is interesting about the problem, meaning it is no easier than finding the actual closest pair.
2. The complexity of this algorithm depends upon the complexity of sorting. Use an $O(n \log n)$ algorithm to sort, and it takes $O(n \log n + n)$ to find the closest pair.
3. This reduction and the fact that there is an $\Omega(n \log n)$ lower bound on sorting *does not* prove that a close-enough pair must take $\Omega(n \log n)$ time in the worst case. Perhaps this is just a slow algorithm for a close-enough pair, and there is a faster one lurking somewhere?
4. On the other hand, *if* we knew that a close-enough pair required $\Omega(n \log n)$ time to solve in the worst case, this reduction would suffice to prove that sorting couldn't be solved any faster than $\Omega(n \log n)$ because that would imply a faster algorithm for a close-enough pair.

9.2.2 Longest Increasing Subsequence

In Chapter 8, we demonstrated how dynamic programming can be used to solve a variety of problems, including string edit distance (Section 8.2 (page 280)) and longest increasing subsequence (Section 8.3 (page 289)). To review,

Problem: Edit Distance
Input: Integer or character sequences S and T; penalty costs for each insertion (c_{ins}), deletion (c_{del}), and substitution (c_{del}).
Output: What is the minimum cost sequence of operations to transform S to T?

Problem: Longest Increasing Subsequence
Input: An integer or character sequence S.
Output: What is the longest sequence of integer positions $\{p_1, \ldots, p_m\}$ such that $p_i < p_{i+1}$ and $S_{p_i} < S_{p_{i+1}}$?

In fact, longest increasing subsequence (LIS) can be solved as a special case of edit distance:

```
Longest Increasing Subsequence(S)
    T = Sort(S)
    c_ins = c_del = 1
    c_sub = ∞
    Return (|S| - EditDistance(S,T,c_ins,c_del,c_del)/2)
```

Why does this work? By constructing the second sequence T as the elements of S sorted in increasing order, we ensure that any common subsequence must be an

increasing subsequence. If we are never allowed to do any substitutions (because $c_{sub} = \infty$), the optimal alignment of two sequences finds the longest common subsequence between them and removes everything else. Thus, transforming $\{3, 1, 2\}$ to $\{1, 2, 3\}$ costs two, namely inserting and deleting the unmatched 3. The length of S minus half this cost gives the length of the LIS.

What are the implications of this reduction? The reduction takes $O(n \log n)$ time. Because edit distance takes time $O(|S| \cdot |T|)$, this gives a quadratic algorithm to find the longest increasing subsequence of S, which is the same complexity as the the algorithm presented in Section 8.3 (page 289). In fact, there exists a faster $O(n \log n)$ algorithm for LIS using clever data structures, while edit distance is known to be quadratic in the worst case. Here, our reduction gives us a simple but not optimal polynomial-time algorithm.

9.2.3 Least Common Multiple

The *least common multiple* and *greatest common divisor* problems arise often in working with integers. We say b *divides* a ($b|a$) if there exists an integer d that $a = bd$. Then:

Problem: Least Common Multiple (lcm)
Input: Two integers x and y.
Output: Return the smallest integer m such that m is a multiple of x and m is also a multiple of y.

Problem: Greatest Common Divisor (gcd)
Input: Two integers x and y.
Output: Return the largest integer d such that d divides x and d divides y.

For example, $lcm(24, 36) = 72$ and $gcd(24, 36) = 12$. Both problems can be solved easily after reducing x and y to their prime factorizations, but no efficient algorithm is known for factoring integers (see Section 13.8 (page 420)). Fortunately, Euclid's algorithm gives an efficient way to solve greatest common divisor without factoring. It is a recursive algorithm that rests on two observations. First,

> if $b|a$, then $gcd(a, b) = b$.

This should be pretty clear. if b divides a, then $a = bk$ for some integer k, and thus $gcd(bk, b) = b$. Second,

> If $a = bt + r$ for integers t and r, then $gcd(a, b) = gcd(b, r)$.

Since $x \cdot y$ is a multiple of both x and y, $lcm(x, y) \leq xy$. The only way that there can be a smaller common multiple is if there is some nontrivial factor shared between x and y. This observation, coupled with Euclid's algorithm, provides an efficient way to compute least common multiple, namely

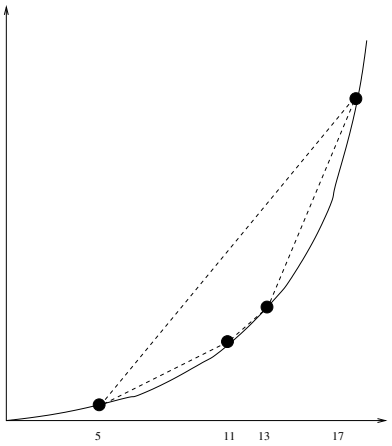


Figure 9.1: Reducing convex hull to sorting by mapping points to a parabola

LeastCommonMultiple(x,y)
Return $(xy/gcd(x, y))$.

This reduction gives us a nice way to reuse Euclid's efforts on another problem.

9.2.4 Convex Hull (*)

Our final example of a reduction from an "easy" problem (i.e. , one that can be solved in polynomial time) goes from finding convex hulls to sorting numbers. A polygon is *convex* if the straight line segment drawn between any two points inside the polygon P must lie completely within the polygon. This is the case when P contains no notches or *concavities*, so convex polygons are nicely shaped. The convex hull provides a very useful way to provide structure to a point set. Applications are presented in Section 17.2 (page 568).

Problem: Convex Hull
Input: A set S of n points in the plane.
Output: Find the smallest convex polygon containing all the points of S.

We now show how to transform from sorting to convex hull. This means we must translate each number to a point. We do so by mapping x to (x, x^2). Why? It means each integer is mapped to a point on the parabola $y = x^2$. Since this parabola is convex, every point must be on the convex hull. Furthermore, since neighboring points on the convex hull have neighboring x values, the convex hull returns the points sorted by the x-coordinate—i.e., the original numbers. Creating and reading off the points takes $O(n)$ time:

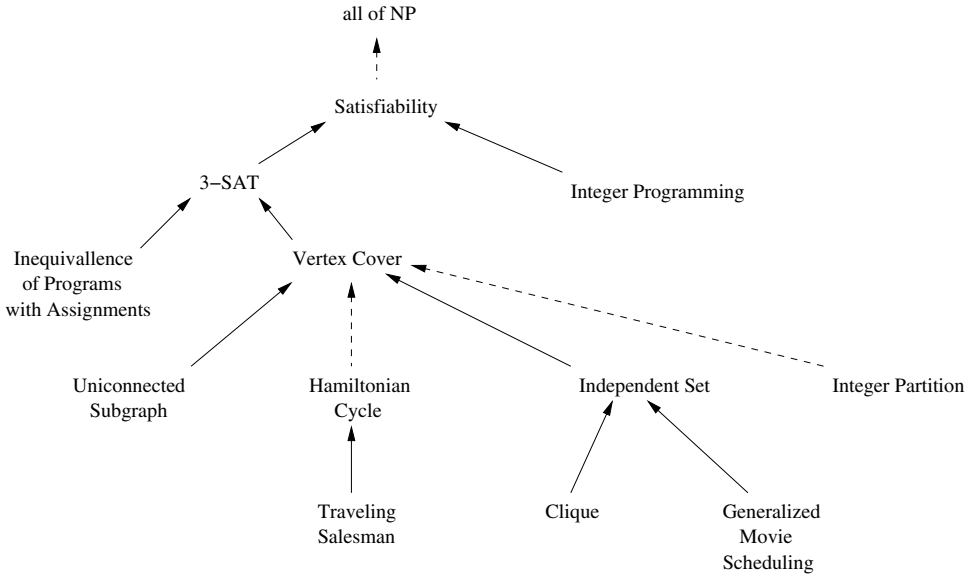


Figure 9.2: A portion of the reduction tree for NP-complete problems. Solid lines denote the reductions presented in this chapter

---

Sort(S)
    For each $i \in S$, create point (i, i^2).
    Call subroutine convex-hull on this point set.
    From the leftmost point in the hull,
        read off the points from left to right.

What does this mean? Recall the sorting lower bound of $\Omega(n \lg n)$. If we could compute convex hull in better than $n \lg n$, this reduction implies that we could sort faster than $\Omega(n \lg n)$, which violates our lower bound. Thus, convex hull must take $\Omega(n \lg n)$ as well! Observe that any $O(n \lg n)$ convex hull algorithm also gives us a complicated but correct $O(n \lg n)$ sorting algorithm as well.

9.3 Elementary Hardness Reductions

The reductions in the previous section demonstrate transformations between pairs of problems for which efficient algorithms exist. However, we are mainly concerned with using reductions to prove hardness, by showing that a fast algorithm for *Bandersnatch* would imply one that cannot exist for *Bo-billy*.

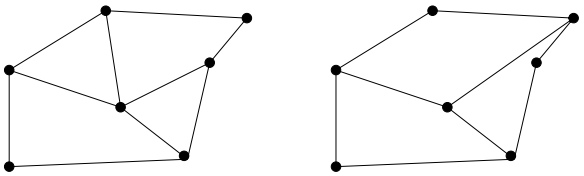


Figure 9.3: Graphs with (l) and without (r) Hamiltonian cycles

For now, I want you to trust me when I say that *Hamiltonian cycle* and *vertex cover* are hard problems. The entire picture (presented in Figure 9.2) will become clear by the end of the chapter.

9.3.1 Hamiltonian Cycle

The Hamiltonian cycle problem is one of the most famous in graph theory. It seeks a tour that visits each vertex of a given graph exactly once. It has a long history and many applications, as discussed in Section 16.5. Formally, it is defined as:

Problem: Hamiltonian Cycle
Input: An unweighted graph G.
Output: Does there exist a simple tour that visits each vertex of G without repetition?

Hamiltonian cycle has some obvious similarity to the traveling salesman problem. Both problems seek a tour that visits each vertex exactly once. There are also differences between the two problems. TSP works on weighted graphs, while Hamiltonian cycle works on unweighted graphs. The following reduction from Hamiltonian cycle to traveling salesman shows that the similarities are greater than the differences:

HamiltonianCycle($G = (V, E)$)
    Construct a complete weighted graph $G' = (V', E')$ where $V' = V$.
    $n = |V|$
    for $i = 1$ to n do
        for $j = 1$ to n do
            if $(i, j) \in E$ then $w(i, j) = 1$ else $w(i, j) = 2$
    Return the answer to Traveling-Salesman-Decision-Problem(G', n).

The actual reduction is quite simple, with the translation from unweighted to weighted graph easily performed in $O(n^2)$ time. Further, this translation is designed to ensure that the answers of the two problems will be identical. If the graph G has a Hamiltonian cycle $\{v_1, \ldots, v_n\}$, then this exact same tour will correspond to n edges in E', each with weight 1. This gives a TSP tour in G' of weight exactly

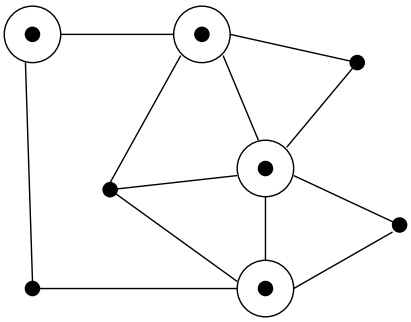


Figure 9.4: Circled vertices form a vertex cover, and the others form an independent set

n. If G does not have a Hamiltonian cycle, then there can be no such TSP tour in G' because the only way to get a tour of cost n in G would be to use only edges of weight 1, which implies a Hamiltonian cycle in G.

This reduction is both efficient and truth preserving. A fast algorithm for TSP would imply a fast algorithm for Hamiltonian cycle, while a hardness proof for Hamiltonian cycle would imply that TSP is hard. Since the latter is the case, this reduction shows that TSP is hard, at least as hard as the Hamiltonian cycle.

9.3.2 Independent Set and Vertex Cover

The vertex cover problem, discussed more thoroughly in Section 16.3 (page 530), asks for a small set of vertices that contacts each edge in a graph. More formally:

Problem: Vertex Cover
Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
Output: Is there a subset S of at most k vertices such that every $e \in E$ contains at least one vertex in S?

It is trivial to find *a* vertex cover of a graph, namely the cover that consists of all the vertices. More tricky is to cover the edges using as small a set of vertices as possible. For the graph in Figure 9.4, four of the eight vertices are sufficient to cover.

A set of vertices S of graph G is *independent* if there are no edges (x, y) where both $x \in S$ and $y \in S$. This means there are no edges between any two vertices in independent set. As discussed in Section 16.2 (page 528), independent set arises in facility location problems. The maximum independent set decision problem is formally defined:

Problem: Independent Set
Input: A graph G and integer $k \leq |V|$.
Output: Does there exist an independent set of k vertices in G?

Both vertex cover and independent set are problems that revolve around finding special subsets of vertices: the first with representatives of every edge, the second with no edges. If S is the vertex cover of G, the remaining vertices $S - V$ must form an independent set, for if an edge had both vertices in $S - V$, then S could not have been a vertex cover. This gives us a reduction between the two problems:

VertexCover(G, k)
    $G' = G$
    $k' = |V| - k$
    Return the answer to IndependentSet(G', k')

Again, a simple reduction shows that the two problems are identical. Notice how this translation occurs without any knowledge of the answer. We transform the *input*, not the solution. This reduction shows that the hardness of vertex cover implies that independent set must also be hard. It is easy to reverse the roles of the two problems in this particular reduction, thus proving that both problems are equally hard.

Stop and Think: Hardness of General Movie Scheduling

Problem: Prove that the *general* movie scheduling problem is NP-complete, with a reduction from independent set.

Problem: General Movie Scheduling Decision Problem
Input: A set I of n *sets of* intervals on the line, integer k.
Output: Can a subset of at least k mutually nonoverlapping interval *sets* which can be selected from I?

Solution: Recall the movie scheduling problem, discussed in Section 1.2 (page 9). Each possible movie project came with a single time interval during which filming took place. We sought the largest possible subset of movie projects such that no two conflicting projects (i.e. , both requiring the actor at the same time) were selected.

The general problem allows movie projects to have discontinuous schedules. For example, Project A running from January-March and May-June does not intersect Project B running in April and August, but *does* collide with Project C running from June-July.

If we are going to prove general movie scheduling hard from independent set, what is Bandersnatch and what is Bo-billy? We need to show how to translate *all* independent set problems into instances of movie scheduling—i.e. , sets of disjointed line intervals.

What is the correspondence between the two problems? Both problems involve selecting the largest subsets possible—of vertices and movies, respectively. This

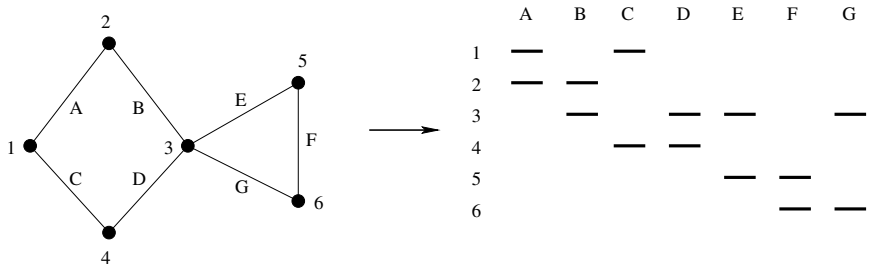


Figure 9.5: Reduction from independent set to generalized movie scheduling, with numbered vertices and lettered edges

---

suggests we must translate vertices into movies. Further, both require the selected elements not to interfere, by sharing an edge or overlapping an interval, respectively.

IndependentSet(G, k)
    $I = \emptyset$
    For the ith edge (x, y), $1 \leq i \leq m$
        Add interval $[i, i + 0.5]$ for movie X to I
        Add interval $[i, i + 0.5]$ for movie y to I
    Return the answer to GeneralMovieScheduling(I, k)

My construction is as follows. Create an interval on the line for each of the m edges of the graph. The movie associated with each vertex will contain the intervals for the edges adjacent with it, as shown in Figure 9.5.

Each pair of vertices sharing an edge (forbidden to be in independent set) defines a pair of movies sharing a time interval (forbidden to be in the actor's schedule). Thus, the largest satisfying subsets for both problems are the same, and a fast algorithm for solving general movie scheduling gives us a fast algorithm for solving independent set. Thus, general movie scheduling must be hard as hard as independent set, and hence NP-complete. ∎

9.3.3 Clique

A social clique is a group of mutual friends who all hang around together. A graph-theoretic *clique* is a complete subgraph where each vertex pair has an edge between them. Cliques are the densest possible subgraphs:

Problem: Maximum Clique
Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
Output: Does the graph contain a clique of k vertices; i.e. , is there a subset $S \subset V$, where $|S| \leq k$, such that every pair of vertices in S defines an edge of G?

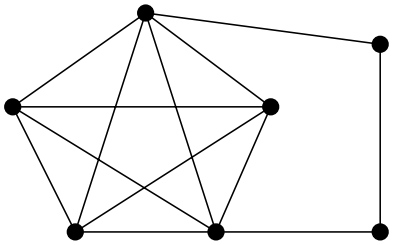


Figure 9.6: A small graph with a five-vertex clique

The graph in Figure 9.6 contains a clique of five vertices. Within the friendship graph, we would expect to see large cliques corresponding to workplaces, neighborhoods, religious organizations, and schools. Applications of cliques are further discussed in Section 16.1 (page 525).

In the independent set problem, we looked for a subset S with no edges between two vertices of S. This contrasts with clique, where we insist that there always be an edge between two vertices. A reduction between these problems follows by reversing the roles of edges and nonedges—an operation known as *complementing* the graph:

IndependentSet(G, k)
    Construct a graph $G' = (V', E')$ where $V' = V$, and
        For all (i, j) not in E, add (i, j) to E'
    Return the answer to Clique(G', k)

These last two reductions provide a chain linking of three different problems. The hardness of clique is implied by the hardness of independent set, which is implied by the hardness of vertex cover. By constructing reductions in a chain, we link together pairs of problems in implications of hardness. Our work is done as soon as all these chains begin with a single problem that is accepted as hard. Satisfiability is the problem that will serve as the first link in this chain.

9.4 Satisfiability

To demonstrate the hardness of all problems using reductions, we must start with a single problem that is absolutely, certifiably, undeniably hard. The mother of all NP-complete problems is a logic problem named *satisfiability*:

Problem: Satisfiability
Input: A set of Boolean variables V and a set of clauses C over V.
Output: Does there exist a satisfying truth assignment for C—i.e. , a way to set the variables $v_1, \ldots, v_n$ true or false so that each clause contains at least one true literal?

This can be made clear with two examples. Suppose that $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$ over the Boolean variables $V = \{v_1, v_2\}$. We use $\overline{v}_i$ to denote the complement of the variable v_i, so we get credit for satisfying a particular clause containing v_i if $v_i =$ true, or a clause containing $\overline{v}_i$ if $v_i =$ false. Therefore, satisfying a particular set of clauses involves making a series of n true or false decisions, trying to find the right truth assignment to satisfy all of them.

These example clauses $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$ can be satisfied by either setting $v_1 = v_2 =$ true or $v_1 = v_2 =$ false. However, consider the set of clauses $C = \{\{v_1, v_2\}, \{v_1, \overline{v}_2\}, \{\overline{v}_1\}\}$. Here there can be no satisfying assignment, because v_1 must be false to satisfy the third clause, which means that v_2 must be false to satisfy the second clause, which then leaves the first clause unsatisfiable. Although you try, and you try, and you try, and you try, you can't get no satisfaction.

For a combination of social and technical reasons, it is well accepted that satisfiability is a hard problem; one for which no worst-case polynomial-time algorithm exists. Literally every top-notch algorithm expert in the world (and countless lesser lights) have directly or indirectly tried to come up with a fast algorithm to test whether a given set of clauses is satisfiable. All have failed. Furthermore, many strange and impossible-to-believe things in the field of computational complexity have been shown to be true if there exists a fast satisfiability algorithm. Satisfiability is a hard problem, and we should feel comfortable accepting this. See Section 14.10 (page 472) for more on the satisfiability problem and its applications.

9.4.1 3-Satisfiability

Satisfiability's role as the first NP-complete problem implies that the problem is hard to solve in the worst case. But certain special-case instances of the problem are not necessarily so tough. Suppose that each clause contains exactly one literal. We must appropriately set that literal to satisfy such a clause. We can repeat this argument for every clause in the problem instance. Thus only when we have two clauses that directly contradict each other, such as $C = \{\{v_1\}, \{\overline{v}_1\}\}$, will the set not be satisfiable.

Since clause sets with only one literal per clause are easy to satisfy, we are interested in slightly larger classes. How many literals per clause do you need to turn the problem from polynomial to hard? This transition occurs when each clause contains three literals, i.e.

Problem: 3-Satisfiability (3-SAT)
Input: A collection of clauses C where each clause contains exactly 3 literals, over a set of Boolean variables V.
Output: Is there a truth assignment to V such that each clause is satisfied?

Since this is a restricted case of satisfiability, the hardness of 3-SAT implies that satisfiability is hard. The converse isn't true, since the hardness of general satisfiability might depend upon having long clauses. We can show the hardness

of 3-SAT using a reduction that translates every instance of satisfiability into an instance of 3-SAT without changing whether it is satisfiable.

This reduction transforms each clause independently based on its *length*, by adding new clauses and Boolean variables along the way. Suppose clause C_i contained k literals:

- $k = 1$, meaning that $C_i = \{z_1\}$ – We create two new variables v_1, v_2 and four new 3-literal clauses: $\{v_1, v_2, z_1\}$, $\{v_1, \overline{v}_2, z_1\}$, $\{\overline{v}_1, v_2, z_1\}$, $\{\overline{v}_1, \overline{v}_2, z_1\}$. Note that the only way that all four of these clauses can be simultaneously satisfied is if $z_1 = \text{true}$, which also means the original C_i will be satisfied.
- $k = 2$, meaning that $C_i = \{z_1, z_2\}$ – We create one new variable v_1 and two new clauses: $\{v_1, z_1, z_2\}$, $\{\overline{v}_1, z_1, z_2\}$. Again, the only way to satisfy both of these clauses is to have at least one of z_1 and z_2 be true, thus satisfying c_i.
- $k = 3$, meaning that $C_i = \{z_1, z_2, z_3\}$ – We copy C_i into the 3-SAT instance unchanged: $\{z_1, z_2, z_3\}$.
- $k > 3$, meaning that $C_i = \{z_1, z_2, ..., z_n\}$ – We create $n-3$ new variables and $n-2$ new clauses in a chain, where for $2 \leq j \leq n-3$, $C_{i,j} = \{v_{i,j-1}, z_{j+1}, \overline{v}_{i,j}\}$, $C_{i,1} = \{z_1, z_2, \overline{v}_{i,1}\}$, and $C_{i,n-2} = \{v_{i,n-3}, z_{n-1}, z_n\}$.

The most complicated case here is that of the large clauses. If none of the original literals in C_i are true, then there are not enough new variables to be able to satisfy all of the new subclauses. You can satisfy $C_{i,1}$ by setting $v_{i,1} = \text{false}$, but this forces $v_{i,2} = \text{false}$, and so on until finally $C_{i,n-2}$ cannot be satisfied. However, if any single literal $z_i = \text{true}$, then we have $n-3$ free variables and $n-3$ remaining 3-clauses, so we can satisfy each of them.

This transform takes $O(m+n)$ time if there were n clauses and m total literals in the SAT instance. Since any SAT solution also satisfies the 3-SAT instance and any 3-SAT solution describes how to set the variables giving a SAT solution, the transformed problem is equivalent to the original.

Note that a slight modification to this construction would serve to prove that 4-SAT, 5-SAT, or any ($k \geq 3$)-SAT is also NP-complete. However, this construction breaks down if we try to use it for 2-SAT, since there is no way to stuff anything into the chain of clauses. It turns out that a depth-first search on an appropriate graph can be used to give a linear-time algorithm for 2-SAT, as discussed in Section 14.10 (page 472).

9.5 Creative Reductions

Since both satisfiability and 3-SAT are known to be hard, we can use either of them in reductions. Usually 3-SAT is the better choice, because it is simpler to work with. What follows are a pair of more complicated reductions, designed to serve as examples and also increase our repertoire of known hard problems. Many

reductions are quite intricate, because we are essentially programming one problem in the language of a significantly different problem.

One perpetual point of confusion is getting the direction of the reduction right. Recall that we must transform *every* instance of a known NP-complete problem into an instance of the problem we are interested in. If we perform the reduction the other way, all we get is a slow way to solve the problem of interest, by using a subroutine that takes exponential time. This always is confusing at first, for this direction of reduction seems backwards. Make sure you understand the direction of reduction now, and think back to this whenever you get confused.

9.5.1 Integer Programming

As discussed in Section 13.6 (page 411), integer programming is a fundamental combinatorial optimization problem. It is best thought of as linear programming with the variables restricted to take only integer (instead of real) values.

Problem: Integer Programming
Input: A set of integer variables V, a set of inequalities over V, a maximization function $f(V)$, and an integer B.
Output: Does there exist an assignment of integers to V such that all inequalities are true and $f(V) \geq B$?

Consider the following two examples. Suppose

$$v_1 \geq 1, \quad v_2 \geq 0$$

$$v_1 + v_2 \leq 3$$

$$f(v) : 2v_2, \quad B = 3$$

A solution to this would be $v_1 = 1$, $v_2 = 2$. Not all problems have realizable solutions, however. For the following problem:

$$v_1 \geq 1, \quad v_2 \geq 0$$

$$v_1 + v_2 \leq 3$$

$$f(v) : 2v_2, \quad B = 5$$

The maximum possible value of $f(v)$ given the constraints is $2 \times 2 = 4$, so there can be no solution to the associated decision problem.

We show that integer programming is hard using a reduction from 3-SAT. For this particular reduction, general satisfiability would work just as well, although usually 3-SAT makes reductions easier.

In which direction must the reduction go? We want to prove integer programming is hard, and know that 3-SAT is hard. If I could solve 3-SAT using integer

programming and integer programming were easy, this would mean that satisfiability would be easy. Now the direction should be clear; we must translate 3-SAT into integer programming.

What should the translation look like? Every satisfiability instance contains Boolean (true/false) variables and clauses. Every integer programming instance contains integer variables (values restricted to 0,1,2,...) and constraints. A reasonable idea is to make the integer variables correspond to Boolean variables and use constraints to serve the same role as the clauses do in the original problem.

Our translated integer programming problem will have twice as many variables as the SAT instance—one for each variable and one for its complement. For each variable v_i in the set problem, we will add the following constraints:

- We restrict each integer programming variable V_i to values of either 0 or 1, but adding constraints $1 \geq V_i \geq 0$ and $1 \geq \overline{V}_i \geq 0$. Thus coupled with integrality, they correspond to values of true and false.

- We ensure that exactly one of the two integer programming variables associated with a given SAT variable is true, by adding constraints so that $1 \geq V_i + \overline{V}_i \geq 1$.

For each 3-SAT clause $C_i = \{z_1, z_2, z_3\}$, construct a constraint: $V_1 + V_2 + V_3 \geq 1$. To satisfy this constraint, at least one of the literals per clause must be set to 1, thus corresponding to a true literal. Satisfying this constraint is therefore equivalent to satisfying the clause.

The maximization function and bound prove relatively unimportant, since we have already encoded the entire 3-SAT instance. By using $f(v) = V_1$ and $B = 0$, we ensure that they will not interfere with any variable assignment satisfying all the inequalities. Clearly, this reduction can be done in polynomial time. To establish that this reduction preserves the answer, we must verify two things:

- *Any SAT solution gives a solution to the IP problem* – In any SAT solution, a true literal corresponds to a 1 in the integer program, since the clause is satisfied. Therefore, the sum in each clause inequality is ≥ 1.

- *Any IP solution gives a solution to the original SAT problem* – All variables must be set to either 0 or 1 in any solution to this integer programming instance. If $V_i = 1$, then set literal $z_i =$ true. If $V_i = 0$, then set literal $z_i =$ false. This is a legal assignment which must also satisfy all the clauses.

The reduction works both ways, so integer programming must be hard. Notice the following properties, which hold true in general for NP-completeness proofs:

1. This reduction preserved the structure of the problem. It did not *solve* the problem, just put it into a different format.

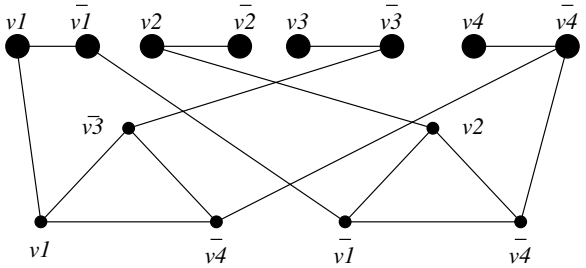


Figure 9.7: Reducing satisfiability instance $\{\{v_1, \bar{v_3}, \bar{v_4}\}, \{\bar{v_1}, v_2, \bar{v_4}\}\}$ to vertex cover

2. The possible IP instances that can result from this transformation are only a small subset of all possible IP instances. However, since some of them are hard, the general problem must be hard.

3. The transformation captures the essence of *why* IP is hard. It has nothing to do with having big coefficients or big ranges of variables, because restricting them to 0/1 is enough. It has nothing to do with having inequalities with large numbers of variables. Integer programming is hard because satisfying a set of constraints is hard. A careful study of the properties needed for a reduction can tell us a lot about the problem.

9.5.2 Vertex Cover

Algorithmic graph theory proves to be a fertile ground for hard problems. The prototypical NP-complete graph problem is vertex cover, previously defined in Section 9.3.2 (page 325) as follows:

Problem: Vertex Cover
Input: A graph $G = (V, E)$ and integer $k \leq |V|$.
Output: Is there a subset S of at most k vertices such that every $e \in E$ has at least one vertex in S?

Demonstrating the hardness of vertex cover proves more difficult than the previous reductions we have seen, because the structure of the two relevant problems is very different. A reduction from 3-satisfiability to vertex cover has to construct a graph G and bound k from the variables and clauses of the satisfiability instance.

First, we translate the variables of the 3-SAT problem. For each Boolean variable v_i, we create two vertices v_i and $\bar{v_i}$ connected by an edge. At least n vertices will be needed to cover all these edges, since no two of the edges will share a vertex.

Second, we translate the clauses of the 3-SAT problem. For each of the c clauses, we create three new vertices, one for each literal in each clause. The three vertices of each clause will be connected so as to form c triangles. At least two vertices per

triangle must be included in any vertex cover of these triangles, for a total of $2c$ cover vertices.

Finally, we will connect these two sets of components together. Each literal in the vertex "gadgets" is connected to vertices in the clause gadgets (triangles) that share the same literal. From a 3-SAT instance with n variables and c clauses, this constructs a graph with $2n + 3c$ vertices. The complete reduction for the 3-SAT problem $\{\{v_1, \bar{v_3}, \bar{v_4}\}, \{\bar{v_1}, v_2, \bar{v_4}\}\}$ is shown in Figure 9.7.

This graph has been designed to have a vertex cover of size $n + 2c$ if and only if the original expression is satisfiable. By the earlier analysis, every vertex cover must have at least $n + 2c$ vertices, since adding the connecting edges to G cannot shrink the size of the vertex cover to less than that of the disconnected vertex and clause gadgets. To show that our reduction is correct, we must demonstrate that:

- *Every satisfying truth assignment gives a vertex cover* – Given a satisfying truth assignment for the clauses, select the n vertices from the vertex gadgets that correspond to true literals to be members of the vertex cover. Since this defines a satisfying truth assignment, a true literal from each clause must cover at least one of the three cross edges connecting each triangle vertex to a vertex gadget. Therefore, by selecting the other two vertices of each clause triangle, we also pick up all remaining cross edges.

- *Every vertex cover gives a satisfying truth assignment* – In any vertex cover C of size $n + 2c$, exactly n of the vertices must belong to the vertex gadgets. Let these first stage vertices define the truth assignment, while the remaining $2c$ cover vertices must be distributed at two per clause gadget. Otherwise a clause gadget edge must go uncovered. These clause gadget vertices can cover only two of the three connecting cross edges per clause. Therefore, if C gives a vertex cover, at least one cross edge per clause must be covered, meaning that the corresponding truth assignment satisfies all clauses.

This proof of the hardness of vertex cover, chained with the clique and independent set reductions of Section 9.3.2 (page 325), gives us a library of hard graph problems that we can use to make future hardness proofs easier.

Take-Home Lesson: A small set of NP-complete problems (3-SAT, vertex cover, integer partition, and Hamiltonian cycle) suffice to prove the hardness of most other hard problems.

9.6 The Art of Proving Hardness

Proving that problems are hard is a skill. But once you get the hang of it, reductions can be surprisingly straightforward and pleasurable to do. Indeed, the dirty little secret of NP-completeness proofs is that they are usually easier to create than explain, in much the same way that it can be easier to rewrite old code than understand and modify it.

It takes experience to judge whether a problem is likely to be hard. Perhaps the quickest way to gain this experience is through careful study of the catalog. Slightly changing the wording of a problem can make the difference between it being polynomial or NP-complete. Finding the shortest path in a graph is easy, while finding the longest path in a graph is hard. Constructing a tour that visits all the edges once in a graph is easy (Eulerian cycle), while constructing a tour that visits all the vertices once is hard (Hamiltonian cycle).

The first thing to do when you suspect a problem might be NP-complete is look in Garey and Johnson's book *Computers and Intractability* [GJ79], which contains a list of several hundred problems known to be NP-complete. Likely you will find the problem you are interested in.

Otherwise I offer the following advice to those seeking to prove the hardness of a given problem:

- *Make your source problem as simple (i.e. , restricted) as possible.*

  Never try to use the general traveling salesman problem (TSP) as a source problem. Better, use Hamiltonian cycle (i.e. , TSP) where all the weights are 1 or ∞. Even better, use Hamiltonian path instead of cycle, so you don't have to worry about closing up the cycle. Best of all, use Hamiltonian path on directed planar graphs where each vertex has total degree 3. All of these problems are equally hard, but the more you can restrict the problem that you are reducing, the less work your reduction has to do.

  As another example, never try to use full satisfiability to prove hardness. Start with 3-satisfiability. In fact, you don't even have to use full 3-satisfiability. Instead, you can use *planar 3-satisfiability*, where there exists a way to draw the clauses as a graph in the plane such that you can connect all instances of the same literal together without edges crossing. This property tends to be useful in proving the hardness of geometric problems. All these problems are equally hard, and hence NP-completeness reductions using any of them are equally convincing.

- *Make your target problem as hard as possible.*

  Don't be afraid to add extra constraints or freedoms to make your target problem more general. Perhaps your undirected graph problem can be generalized into a directed graph problem and can hence be easier to prove hard. Once you have a proof of hardness for the general problem, you can then go back and try to simplify the target.

- *Select the right source problem for the right reason.*

  Selecting the right source problem makes a big difference in how difficult it is to prove a problem hard. This is the first and easiest place to go wrong, although theoretically any NP-complete problem works as well as any other. When trying to prove that a problem is hard, some people fish around through

lists of dozens of problems, looking for the best fit. These people are amateurs; odds are they never will recognize the problem they are looking for when they see it.

I use four (and only four) problems as candidates for my hard source problem. Limiting them to four means that I can know a lot about each of these problems, such as which variants of the problems are hard and which are not. My favorite source problems are:

- *3-SAT*: The old reliable. When none of the three problems below seem appropriate, I go back to the original source.
- *Integer partition*: This is the one and only choice for problems whose hardness seems to require using large numbers.
- *Vertex cover*: This is the answer for any graph problem whose hardness depends upon *selection*. Chromatic number, clique, and independent set all involve trying to select the correct subset of vertices or edges.
- *Hamiltonian path*: This is my choice for any graph problem whose hardness depends upon *ordering*. If you are trying to route or schedule something, Hamiltonian path is likely your lever into the problem.

- *Amplify the penalties for making the undesired selection.*

  Many people are too timid in their thinking about hardness proofs. You are trying to translate one problem into another, while keeping the problems as close to their original identities as as possible. The easiest way to do this is to be bold with your penalties; to punish anyone for trying to deviate from your intended solution. Your thinking should be, "if you select this element, then you must pick up this huge set that prevents you from finding an optimal solution." The sharper the consequences for doing what is undesired, the easier it is to prove the equivalence of the problems.

- *Think strategically at a high level, then build gadgets to enforce tactics.*

  You should be asking yourself the following types of questions: "How can I force that A or B is chosen but not both?" "How can I force that A is taken before B?" "How can I clean up the things I did not select?" Once you have an idea of what you want your gadgets to do, you can worry about how to actually craft them.

- *When you get stuck, alternate between looking for an algorithm or a reduction.*

  Sometimes the reason you cannot prove hardness is that there exists an efficient algorithm to solve your problem! Techniques such as dynamic programming or reducing problems to powerful but polynomial-time graph problems, such as matching or network flow, can yield surprising algorithms. Whenever you can't prove hardness, it pays to alter your opinion occasionally to keep yourself honest.

9.7 War Story: Hard Against the Clock

My class's attention span was running down like sand through an hourglass. Eyes were starting to glaze, even in the front row. Breathing had become soft and regular in the middle of the room. Heads were tilted back and eyes shut in the back.

There were twenty minutes left to go in my lecture on NP-completeness, and I couldn't really blame them. They had already seen several reductions like the ones presented here. But NP-completeness reductions are easier to create than to understand or explain. They had to watch one being created to appreciate how things worked.

I reached for my trusty copy of Garey and Johnson's book [GJ79], which contains a list of over four hundred different known NP-complete problems in an appendix in the back.

"Enough of this!" I announced loudly enough to startle those in the back row. "NP-completeness proofs are sufficiently routine that we can construct them on demand. I need a volunteer with a finger. Can anyone help me?"

A few students in the front held up their hands. A few students in the back held up their fingers. I opted for one from the front row.

"Select a problem at random from the back of this book. I can prove the hardness of any of these problems in the now seventeen minutes remaining in this class. Stick your finger in and read me a problem."

I had definitely gotten their attention. But I could have done that by offering to juggle chain-saws. Now I had to deliver results without cutting myself into ribbons.

The student picked out a problem. "OK, prove that *Inequivalence of Programs with Assignments* is hard," she said.

"Huh? I've never heard of that problem before. What is it? Read me the entire description of the problem so I can write it on the board." The problem was as follows:

Problem: Inequivalence of Programs with Assignments
Input: A finite set X of variables, two programs P_1 and P_2, each a sequence of assignments of the form

$$x_0 \leftarrow \text{ if } (x_1 = x_2) \text{ then } x_3 \text{ else } x_4$$

where the x_i are in X; and a value set V.
Output: Is there an initial assignment of a value from V to each variable in X such that the two programs yield different final values for some variable in X?

I looked at my watch. Fifteen minutes to go. But now everything was on the table. I was faced with a language problem. The input was two programs with variables, and I had to test to see whether they always do the same thing.

"First things first. We need to select a source problem for our reduction. Do we start with integer partition? 3-satisfiability? Vertex cover or Hamiltonian path?"

Since I had an audience, I tried thinking out loud. "Our target is not a graph problem or a numerical problem, so let's start thinking about the old reliable: 3-

satisfiability. There seem to be some similarities. 3-SAT has variables. This thing has variables. To be more like 3-SAT, we could try limiting the variables in this problem so they only take on two values—i.e. , $V = \{\text{true}, \text{false}\}$. Yes. That seems convenient."

My watch said fourteen minutes left. "So, class, which way does the reduction go? 3-SAT to language or language to 3-SAT?"

The front row correctly murmured, "3-SAT to language."

"Right. So we have to translate our set of clauses into two programs. How can we do that? We can try to split the clauses into two sets and write separate programs for each of them. But how do we split them? I don't see any natural way do it, because eliminating any single clause from the problem might suddenly make an unsatisfiable formula satisfiable, thus completely changing the answer. Instead, let's try something else. We can translate all the clauses into one program, and then make the second program be trivial. For example, the second program might ignore the input and always output either only true or only false. This sounds better. *Much* better."

I was still talking out loud to myself, which wasn't that unusual. But I had people listening to me, which was.

"Now, how can we turn a set of clauses into a program? We want to know whether the set of clauses can be satisfied, or if there is an assignment of the variables such that it is true. Suppose we constructed a program to evaluate whether $c_1 = (x_1, \overline{x}_2, x_3)$ is satisfied."

It took me a few minutes worth of scratching before I found the right program to simulate a clause. I assumed that I had access to constants for true and false:

$c_1 =$ if $(x_1 = \text{true})$ then true else false
$c_1 =$ if $(x_2 = \text{false})$ then true else c_1
$c_1 =$ if $(x_3 = \text{true})$ then true else c_1

"Great. Now I have a way to evaluate the truth of each clause. I can do the same thing to evaluate whether all the clauses are satisfied."

$sat =$ if $(c_1 = \text{true})$ then true else false
$sat =$ if $(c_2 = \text{true})$ then sat else false
$\vdots$
$sat =$ if $(c_n = \text{true})$ then sat else false

Now the back of the classroom was getting excited. They were starting to see a ray of hope that they would get to leave on time. There were two minutes left in class.

"Great. So now we have a program that can evaluate to be true if and only if there is a way to assign the variables to satisfy the set of clauses. We need a second program to finish the job. What about $sat = \text{false}$? Yes, that is all we need. Our language problem asks whether the two programs always output the same

thing, regardless of the possible variable assignments. If the clauses are satisfiable, that means that there must be an assignment of the variables such that the long program would output true. Testing whether the programs are equivalent is exactly the same as asking if the clauses are satisfiable."

I lifted my arms in triumph. "And so, the problem is neat, sweet, and NP-complete." I got the last word out just before the bell rang.

9.8 War Story: And Then I Failed

This exercise of picking a random NP-complete problem from the 400+ problems in Garey and Johnson's book and proving hardness on demand was so much fun that I have repeated it each time I have taught the algorithms course. Sure enough, I got it eight times in a row. But just as Joe DiMaggio's 56-game hitting streak came to an end, and Google will eventually have a losing quarter financially, the time came for me to get my comeuppance.

The class had voted to see a reduction from the graph theory section of the catalog, and a randomly selected student picked number 30. Problem GT30 turned out to be:

Problem: Uniconnected Subgraph
Input: Directed graph $G = (V, A)$, positive integer $k \leq |A|$.
Output: Is there a subset of arcs $A' \in A$ with $|A'| \geq k$ such that $G' = (V, A')$ has at most one directed path between any pair of vertices?

"It is a selection problem," I realized as soon as the problem was revealed. After all, we had to select the largest possible subset of arcs so that there were no pair of vertices with multiple paths between them. This meant that vertex cover was the problem of choice.

I worked through how the two problems stacked up. Both sought subsets, although vertex cover wanted subsets of vertices and uniconnected subgraph wanted subsets of edges. Vertex cover wanted the smallest possible subset, while unidirected subgraph wanted the largest possible subset. My source problem had undirected edges while my target had directed arcs, so somehow I would have to add edge direction into the reduction.

I had to do something to direct the edges of the vertex cover graph. I could try to replace each undirected edge (x, y) with a single arc, say from y to x. But quite different directed graphs would result depending upon which direction I selected. Finding the "right" orientation of edges might be a hard problem, certainly too hard to use in the translation phase of the reduction.

I realized I could direct the edges so the resulting graph was a DAG. But then, so what? DAGs certainly can have many different directed paths between pairs of vertices.

Alternately, I could try to replace each undirected edge (x, y) with *two* arcs, from x to y and y to x. Now there was no need to chose the right arcs for my

reduction, but the graph certainly got complicated. I couldn't see how to force things to prevent vertex pairs from having unwanted multiple paths between them.

Meanwhile, the clock was running and I knew it. A sense of panic set in during the last ten minutes of the class, and I realized I wasn't going to get it this time.

There is no feeling worse for a professor than botching up a lecture. You stand up there flailing away, knowing (1) that the students don't understand what you are saying, but (2) they do understand that you also don't understand what you are saying. The bell rang and the students left the room with faces either sympathetic or smirking.

I promised them a solution for the next class, but somehow I kept getting stuck in the same place each time I thought about it. I even tried to cheat and look up the proof in a journal. But the reference that Garey and Johnson cited was a 30-year old unpublished technical report. It wasn't on the web or in our library.

I dreaded the idea of returning to give my next class, the last lecture of the semester. But the night before class the answer came to me in a dream. *"Split the edges,"* the voice said. I awoke with a start and looked at the clock. It was 3:00 AM.

I sat up in bed and scratched out the proof. Suppose I replace each undirected edge (x, y) with a gadget consisting of a new central vertex v_{xy} with arcs going from it to x and y, respectively. This is nice. Now, which vertices are capable of having multiple paths between them? The new vertices had only outgoing edges, so they can only serve as the source of multiple paths. The old vertices had only incoming edges. There was at most one way to get from one of the new source vertices to any of the original vertices of the vertex cover graph, so these could not result in multiple paths.

But now add a sink node s with edges from all original vertices. There were exactly two paths from each new source vertex to this sink—one through each of the two original vertices it was adjacent to. One of these had to be broken to create a uniconnected subgraph. How could we break it? We could pick one of these two vertices to disconnect from the sink by deleting either arc (x, s) or (y, s) for new vertex v_{xy}. We maximize the size of our subgraph by finding the smallest number of arcs to delete. We must delete the outgoing arc from at least one of the two vertices defining each original edge. *This is exactly the same as finding the vertex cover in this graph!* The reduction is illustrated in Figure 9.8.

Presenting this proof in class provided some personal vindication, but more to the point validates the principles I teach for proving hardness. Observe that the reduction really wasn't all that difficult after all: just split the edges and add a sink node. NP-completeness reductions are often surprisingly simple once you look at them the right way.

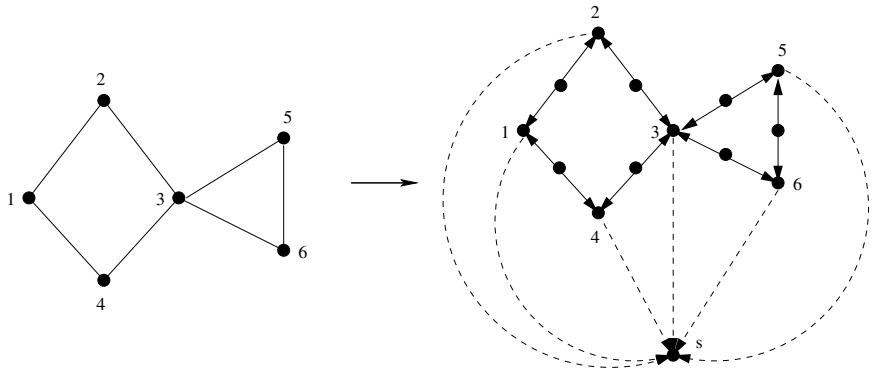


Figure 9.8: Reducing vertex cover to unidirected subgraph, by dividing edges and adding a sink node

9.9 P vs. NP

The theory of NP-completeness rests on a foundation of rigorous but subtle definitions from automata and formal language theory. This terminology is typically confusing to or misused by beginners who lack a mastery of these foundations. It is not really essential to the practical aspects of designing and applying reductions. That said, the question "Is P=NP?" is the most profound open problem in computer science, so any educated algorist should have some idea what the stakes are.

9.9.1 Verification vs. Discovery

The primary question in P vs NP is whether *verification* is really an easier task than initial *discovery*. Suppose that while taking an exam you "happen" to notice the answer of the student next to you. Are you now better off? You wouldn't dare to turn it in without checking, since an able student such as yourself could answer the question correctly if you took enough time to solve it. The issue is whether you can really verify the answer faster than you could find it from scratch.

For the NP-complete decision problems we have studied here, the answer *seems* obvious:

- Can you verify that a graph has a TSP tour of at most k weight given the order of vertices on the tour? Sure. Just add up the weights of the edges on the tour and show it is at most k. That is easier than finding the tour, *isn't it*?

- Can you verify that a given truth assignment represents a solution to a given satisfiability problem? Sure. Just check each clause and make sure it contains

at least one true literal from the given truth assignment. That is easier than finding the satisfying assignment, *isn't it*?

- Can you verify that a graph G has a vertex cover of at most k vertices if given the subset S of at most k vertices forming such a cover? Sure. Just traverse each edge (u, v) of G, and check that either u or v is in S. That is easier than finding the vertex cover, *isn't it*?

At first glance, this seems obvious. The given solutions can be verified in linear time for all three of these problems, while no algorithm faster than brute force search is known to find the solutions for any of them. The catch is that we have no rigorous lower bound *proof* that prevents the existence of fast algorithms to solve these problems. Perhaps there are in fact polynomial algorithms (say $O(n^7)$) that we have just been too blind to see yet.

9.9.2 The Classes P and NP

Every well-defined algorithmic problem must have an asymptotically fastest-possible algorithm solving it, as measured in the Big-Oh, worst-case sense of fastest.

We can think of the class P as an exclusive club for algorithmic problems that a problem can only join after demonstrating that there exists a polynomial-time algorithm to solve it. Shortest path, minimum spanning tree, and the original movie scheduling problem are all members in good standing of this class P. The P stands for *polynomial-time*.

A less-exclusive club welcomes all the algorithmic problems whose solutions can be *verified* in polynomial-time. As shown above, this club contains traveling salesman, satisfiability, and vertex cover, none of which currently have the credentials to join P. However, all the members of P get a free pass into this less exclusive club. If you can solve the problem from scratch in polynomial time, you certainly can verify a solution that fast: just solve it from scratch and see if the solution you get is as good as the one you were given.

We call this less-exclusive club NP. You can think of this as standing for *not-necessarily polynomial-time*.[1]

The $1,000,000 question is whether there are in fact problems in NP that cannot be members of P. If no such problem exists, the classes must be the same and $P = NP$. If even one such a problem exists, the two classes are different and $P \neq NP$. The opinion of most algorists and complexity theorists is that the classes differ, meaning $P \neq NP$, but a much stronger proof than "I can't find a fast enough algorithm" is needed.

[1] In fact, it stands for *nondeterministic polynomial-time*. This is in the sense of nondeterministic automata, if you happen to know about such things.

9.9.3 Why is Satisfiability the Mother of All Hard Problems?

An enormous tree of NP-completeness reductions has been established that entirely rests on the hardness of satisfiability. The portion of this tree demonstrated and/or stated in this chapter (and proven elsewhere) is shown in Figure 9.2.

This may look like a delicate affair. What would it mean if someone *does* find a polynomial-time algorithm for satisfiability? A fast algorithms for any given NP-complete problem (say traveling salesman) implies fast algorithm for all the problems on the path in the reduction tree between TSP and satisfiability (Hamiltonian cycle, vertex cover, and 3-SAT). But a fast algorithm for satisfiability doesn't immediately yield us anything because the reduction path from SAT to SAT is empty.

Fear not. There exists an extraordinary super-reduction (called Cook's theorem) reducing *all* the problems in NP to satisfiability. Thus, if you prove that satisfiability (or equivalently any single NP-complete problem) is in P, then *all* other problems in NP follow and $P = NP$. Since essentially every problem mentioned in this book is in NP, this would be an enormously powerful and surprising result.

Cook's theorem proves that satisfiability is as hard as any problem in NP. Furthermore, it proves that every NP-complete problem is as hard as any other. Any domino falling (i.e. , a polynomial-time algorithm for any NP-complete problem) knocks them all down. Our inability to find a fast algorithm for any of these problems is a strong reason for believing that they are all truly hard, and probably $P \neq NP$.

9.9.4 NP-hard vs. NP-complete?

The final technicality we will discuss is the difference between a problem being NP-hard and NP-complete. I tend to be somewhat loose with my terminology, but there is a subtle (usually irrelevant) distinction between the two concepts.

We say that a problem is *NP-hard* if, like satisfiability, it is at least as hard as any problem in NP. We say that a problem is *NP-complete* if it is NP-hard, and also in NP itself. Because NP is such a large class of problems, most NP-hard problems you encounter will actually be complete, and the issue can always be settled by giving a (usually simple) verification strategy for the problem. All the NP-hard problems we have encountered in this book are also NP-complete.

That said, there are problems that appear to be NP-hard yet not in NP. These problems might be *even harder* than NP-complete! Two-player games such as chess provide examples of problems that are not in NP. Imagine sitting down to play chess with some know-it-all, who is playing white. He pushes his central pawn up two squares to start the game, and announces *checkmate*. The only obvious way to show he is right would be to construct the full tree of all your possible moves with his irrefutable replies and demonstrate that you, in fact, cannot win from the current position. This full tree will have a number of nodes exponential in its height, which is the number of moves before you lose playing your most spirited possible defense.

Clearly this tree cannot be constructed and analyzed in polynomial time, so the problem is not in NP.

9.10 Dealing with NP-complete Problems

For the practical person, demonstrating that a problem is NP-complete is never the end of the line. Presumably, there was a reason why you wanted to solve it in the first place. That application will not go away when told that there is no polynomial-time algorithm. You still seek a program that solves the problem of interest. All you know is that you won't find one that quickly solves the problem to optimality in the worst case. You still have three options:

- *Algorithms fast in the average case* – Examples of such algorithms include backtracking algorithms with substantial pruning.

- *Heuristics* – Heuristic methods like simulated annealing or greedy approaches can be used to quickly find a solution with no guarantee that it will be the best one.

- *Approximation algorithms* – The theory of NP-completeness only stipulates that it is hard to get close to the answer. With clever, problem-specific heuristics, we can probably get *close* to the optimal answer on all possible instances.

Approximation algorithms return solutions with a guarantee attached, namely that the optimal solution can never be much better than this given solution. Thus you can never go too far wrong when using an approximation algorithm. No matter what your input instance is and how lucky you are, you are doomed to do all right. Furthermore, approximation algorithms realizing probably good bounds are often conceptually simple, very fast, and easy to program.

One thing that is usually not clear, however, is how well the solution from an approximation algorithm compares to what you might get from a heuristic that gives you no guarantees. The answer could be worse or it could be better. Leaving your money in a bank savings account guarantees you 3% interest without risk. Still, you likely will do much better investing your money in stocks than leaving it in the bank, even though performance is not guaranteed.

One way to get the best of approximation algorithms and heuristics is to run both of them on the given problem instance and pick the solution giving the better result. This way, you get a solution that comes with a guarantee and a second chance to do even better. When it comes to heuristics for hard problems, sometimes you can have it both ways.

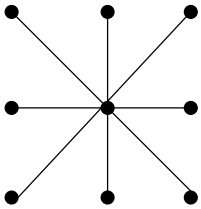


Figure 9.9: Neglecting to pick the center vertex leads to a terrible vertex cover

9.10.1 Approximating Vertex Cover

As we have seen before, finding the minimum vertex cover of a graph is NP-complete. However, a very simple procedure can efficiently find a cover that is at most twice as large as the optimal cover:

VertexCover($G = (V, E)$)
    While ($E \neq \emptyset$) do:
        Select an arbitrary edge $(u, v) \leq E$
        Add both u and v to the vertex cover
        Delete all edges from E that are incident to either u or v.

It should be apparent that this procedure always produces a vertex cover, since each edge is only deleted after an incident vertex has been added to the cover. More interesting is the claim that any vertex cover must use at least half as many vertices as this one. Why? Consider only the $\leq n/2$ edges selected by the algorithm that constitute a matching in the graph. No two of these edges can share a vertex. Therefore, any cover of just these edges must include at least one vertex per edge, which makes it at least half the size of this greedy cover.

There are several interesting things to notice about this algorithm:

- *Although the procedure is simple, it is not stupid* – Many seemingly smarter heuristics can give a far worse performance in the worst case. For example, why not modify the above procedure to select only one of the two vertices for the cover instead of both. After all, the selected edge will be equally well covered by only one vertex. However, consider the star-shaped graph of Figure 9.9. This heuristic will produce a two-vertex cover, while the single-vertex heuristic can return a cover as large as $n - 1$ vertices, should we get unlucky and repeatedly select the leaf instead of the center as the cover vertex we retain.

- *Greedy isn't always the answer* – Perhaps the most natural heuristic for vertex cover would repeatedly select and delete the vertex of highest remaining degree for the vertex cover. After all, this vertex will cover the largest number

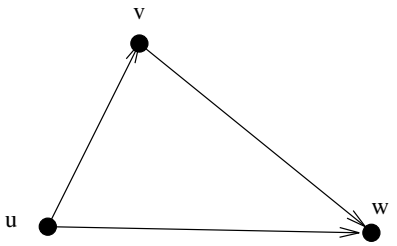


Figure 9.10: The triangle inequality ($d(u,w) \leq d(u,v) + d(v,w)$) typically holds in geometric and weighted graph problems

of possible edges. However, in the case of ties or near ties, this heuristic can go seriously astray. In the worst case, it can yield a cover that is $\Theta(\lg n)$ times optimal.

- *Making a heuristic more complicated does not necessarily make it better* – It is easy to complicate heuristics by adding more special cases or details. For example, the procedure above does not specify which edge should be selected next. It might seem reasonable to next select the edge whose endpoints have the highest degree. However, this does not improve the worst-case bound and just makes it more difficult to analyze.

- *A postprocessing cleanup step can't hurt* – The flip side of designing simple heuristics is that they can often be modified to yield better-in-practice solutions without weakening the approximation bound. For example, a postprocessing step that deletes any unnecessary vertex from the cover can only improve things in practice, even though it won't help the worst-case bound.

The important property of approximation algorithms is relating the size of the solution produced directly to a lower bound on the optimal solution. Instead of thinking about how well we might do, we must think about the worst case—i.e. , how badly we might perform.

9.10.2 The Euclidean Traveling Salesman

In most natural applications of the traveling salesman problem, direct routes are inherently shorter than indirect routes. For example, if a graph's edge weights were the straight-line distances between pairs of cities, the shortest path from x to y will always be "as the crow flies."

The edge weights induced by Euclidean geometry satisfy the triangle inequality, namely that $d(u,w) \leq d(u,v) + d(v,w)$ for all triples of vertices u, v, and w. The general reasonableness of this condition is demonstrated in Figure 9.10. The cost of airfare is an example of a distance function that *violates* the triangle inequality,

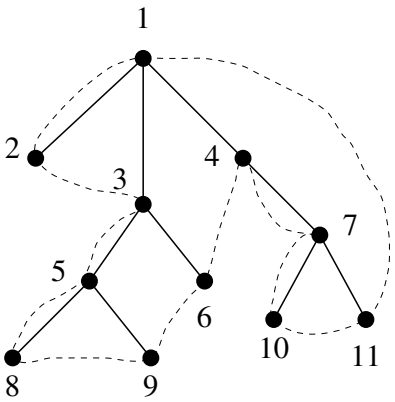


Figure 9.11: A depth-first traversal of a spanning tree, with the shortcut tour

since it is sometimes cheaper to fly through an intermediate city than to fly to the destination directly. TSP remains hard when the distances are Euclidean distances in the plane.

We can approximate the optimal traveling salesman tour using minimum spanning trees or graphs that obey the triangle inequality. First, observe that the weight of a minimum spanning tree is a lower bound on the cost of the optimal tour. Why? Deleting any edge from a tour leaves a path, the total weight of which must be no greater than that of the original tour. This path has no cycles, and hence is a tree, which means its weight is at least that of the minimum spanning tree. Thus the weight of the minimum spanning tree gives a lower bound on the optimal tour.

Consider now what happens in performing a depth-first traversal of a spanning tree. We will visit each edge twice, once going down the tree when discovering the edge and once going up after exploring the entire subtree. For example, in the depth-first search of Figure 9.11, we visit the vertices in order $1-2-1-3-5-8-5-9-5-3-6-3-1-4-7-10-7-11-7-4-1$, thus using every tree edge exactly twice. This tour repeats each edge of the minimum spanning tree twice, and hence costs at most twice the optimal tour.

However, vertices will be repeated on this depth-first search tour. To remove the extra vertices, we can take a shortest path to the next unvisited vertex at each step. The shortcut tour for the tree above is $1-2-3-5-8-9-6-4-7-10-11-1$. Because we have replaced a chain of edges by a single direct edge, the triangle inequality ensures that the tour can only get shorter. Thus, this shortcut tour is also within weight and twice that of optimal. Better, more complicated approximation algorithms for Euclidean TSP exist, as described in Section 16.4 (page 533). No approximation algorithms are known for TSPs that do not satisfy the triangle inequality.

9.10.3 Maximum Acyclic Subgraph

Directed acyclic graphs (DAGs) are easier to work with than general digraphs. Sometimes it is useful to simplify a given graph by deleting a small set of edges or vertices that suffice to break all cycles. Such *feedback set* problems are discussed in Section 16.11 (page 559).

Here we consider an interesting problem in this class where seek to retain as many edges as possible while breaking all directed cycles:

Problem: Maximum Directed Acyclic Subgraph
Input: A directed graph $G = (V, E)$.
Output: Find the largest possible subset $E' \in E$ such that $G' = (V, E')$ is acyclic.

In fact, there is a very simple algorithm that guarantees you a solution with at least half as many edges as optimum. I encourage you to try to find it now before peeking.

Construct *any* permutation of the vertices, and interpret it as a left-right ordering akin to topological sorting. Now some of the edges will point from left to right, while the remainder point from right to left.

One of these two edge subsets must be at least as large as the other. This means it contains at least half the edges. Furthermore each of these two edge subsets must be acyclic for the same reason only DAGs can be topologically sorted—you cannot form a cycle by repeatedly moving in one direction. Thus, the larger edge subset must be acyclic and contain at least half the edges of the optimal solution!

This approximation algorithm *is* simple to the point of almost being stupid. But note that heuristics can make it perform better in practice without losing this guarantee. Perhaps we can try many random permutations, and pick the best. Or we can try to exchange pairs of vertices in the permutations retaining those swaps, which throw more edges onto the bigger side.

9.10.4 Set Cover

The previous sections may encourage a false belief that every problem can be approximated to within a factor of two. Indeed, several catalog problems such as maximum clique cannot be approximated to *any* interesting factor.

Set cover occupies a middle ground between these extremes, having a factor-$\Theta(\lg n)$ approximation algorithm. Set cover is a more general version of the vertex cover problem. As defined in Section 18.1 (page 621),

Problem: Set Cover
Input: A collection of subsets $S = \{S_1, \ldots, S_m\}$ of the universal set $U = \{1, \ldots, n\}$.
Output: What is the smallest subset T of S whose union equals the universal set—i.e. , $\cup_{i=1}^{|T|} T_i = U$?

milestone class	6	5		4					3			2	1	0
uncovered elements	64	51	40	30	25	22	19	16	13	10	7	4	2	1
selected subset size	13	11	10	5	3	3	3	3	3	3	3	2	1	1

Figure 9.12: The coverage process for greedy on a particular instance of set cover

The natural heuristic is greedy. Repeatedly select the subset that covers the largest collection of thus-far uncovered elements until everything is covered. In pseudocode,

```
SetCover(S)
    While (U ≠ ∅) do:
        Identify the subset S_i with the largest intersection with U
        Select S_i for the set cover
        U = U − S_i
```

One consequence of this selection process is that the number of freshly-covered elements defines a nonincreasing sequence as the algorithm proceeds. Why? If not, greedy would have picked the more powerful subset earlier if it, in fact, existed.

Thus we can view this heuristic as reducing the number of uncovered elements from n down to zero by progressively smaller amounts. A trace of such an execution is shown in Figure 9.12.

An important milestone in such a trace occurs each time the number of remaining uncovered elements reduces past a power of two. Clearly there can be at most $\lceil \lg n \rceil$ such events.

Let w_i denote the number of subsets that were selected by the heuristic to cover elements between milestones $2^{i+1} - 1$ and 2^i. Define the width w to be the maximum w_i, where $0 \leq i \leq \lg_2 n$. In the example of Figure 9.12, the maximum width is given by the five subsets needed to go from $2^5 - 1$ down to 2^4.

Since there are at most $\lg n$ such milestones, the solution produced by the greedy heuristic must contain at most $w \cdot \lg n$ subsets. But I claim that the optimal solution must contain *at least* w subsets, so the heuristic solution is no worse than $\lg n$ times optimal.

Why? Consider the average number of new elements covered as we move between milestones $2^{i+1} - 1$ and 2^i. These 2^i elements require w_i subsets, so the average coverage is $\mu_i = 2^i / w_i$. More to the point, the last/smallest of these subsets covers at most μ_i subsets. Thus, *no subset exists in S that can cover more than μ_i of the remaining 2^i elements.* So, to finish the job, we need at least $2^i / \mu_i = w_i$ subsets.

The somewhat surprising thing is that there do exist set cover instances where this heuristic takes $\Omega(\lg n)$ times optimal. The logarithmic factor is a property of the problem/heuristic, not an artifact of weak analysis.

Take-Home Lesson: Approximation algorithms guarantee answers that are always close to the optimal solution. They can provide a practical approach to dealing with NP-complete problems.

Chapter Notes

The notion of NP-completeness was first developed by Cook [Coo71]. Satisfiability really is a $1,000,000 problem, and the Clay Mathematical Institute has offered such a prize to any person who resolves the P=NP question. See *http://www.claymath.org/* for more on the problem and the prize.

Karp [Kar72] showed the importance of Cook's result by providing reductions from satisfiability to more than 20 important algorithmic problems. I recommend Karp's paper for its sheer beauty and economy—he reduces each reduction to three line descriptions showing the problem equivalence. Together, these provided the tools to resolve the complexity of literally hundreds of important problems where no efficient algorithms were known.

The best introduction to the theory of NP-completeness remains Garey and Johnson's book *Computers and Intractability*. It introduces the general theory, including an accessible proof of Cook's theorem [Coo71] that satisfiability is as hard as anything in NP. They also provide an essential reference catalog of more than 300 NP-complete problems, which is a great resource for learning what is known about the most interesting hard problems. The reductions claimed, but missing from this chapter can be found in Garey and Johnson, or textbooks such as [CLRS01].

A few catalog problems exist in a limbo state where it is not known whether the problem has a fast algorithm or is NP-complete. The most prominent of these are graph isomorphism (see Section 16.9 (page 550)) and integer factorization (see Section 13.8 (page 420)). That this limbo list is so short is quite a tribute to the state of the art in algorithm design and the power of NP-completeness. For almost every important problem we either have a fast algorithm or a good solid reason for why one doesn't exist.

The war story problem on unidirected subgraph was originally proven hard in [Mah76].

9.11 Exercises

Transformations and Satisfiability

9-1. *[2]* Give the 3-SAT formula that results from applying the reduction of SAT to 3-SAT for the formula:

$$(x + y + \overline{z} + w + u + \overline{v}) \cdot (\overline{x} + \overline{y} + z + \overline{w} + u + v) \cdot (x + \overline{y} + \overline{z} + w + u + \overline{v}) \cdot (x + \overline{y})$$

9-2. *[3]* Draw the graph that results from the reduction of 3-SAT to vertex cover for the expression

$$(x + \overline{y} + z) \cdot (\overline{x} + y + \overline{z}) \cdot (\overline{x} + y + z) \cdot (x + \overline{y} + \overline{x})$$

9-3. *[4]* Suppose we are given a subroutine which can solve the traveling salesman decision problem of page 318 in, say, linear time. Give an efficient algorithm to find the actual TSP tour by making a polynomial number of calls to this subroutine.

9-4. *[7]* Implement a translator that translates satisfiability instances into equivalent 3-SAT instances.

9-5. *[7]* Design and implement a backtracking algorithm to test whether a set of formulae are satisfiable. What criteria can you use to prune this search?

9-6. *[8]* Implement the vertex cover to satisfiability reduction, and run the resulting clauses through a satisfiability testing code. Does this seem like a practical way to compute things?

Basic Reductions

9-7. *[4]* An instance of the *set cover* problem consists of a set X of n elements, a family F of subsets of X, and an integer k. The question is, does there exist k subsets from F whose union is X?

For example, if $X = \{1, 2, 3, 4\}$ and $F = \{\{1, 2\}, \{2, 3\}, \{4\}, \{2, 4\}\}$, there does not exist a solution for $k = 2$, but there does for $k = 3$ (for example, $\{1, 2\}, \{2, 3\}, \{4\}$).

Prove that set cover is NP-complete with a reduction from vertex cover.

9-8. *[4]* The *baseball card collector problem* is as follows. Given packets $P_1, \ldots, P_m$, each of which contains a subset of this year's baseball cards, is it possible to collect all the year's cards by buying $\leq k$ packets?

For example, if the players are $\{Aaron, Mays, Ruth, Skiena\}$ and the packets are

$$\{\{Aaron, Mays\}, \{Mays, Ruth\}, \{Skiena\}, \{Mays, Skiena\}\},$$

there does not exist a solution for $k = 2$, but there does for $k = 3$, such as

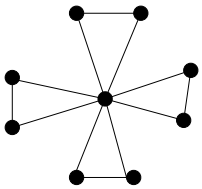
$$\{Aaron, Mays\}, \{Mays, Ruth\}, \{Skiena\}$$

Prove that the baseball card collector problem is NP-hard using a reduction from vertex cover.

9-9. *[4]* The *low-degree spanning tree problem* is as follows. Given a graph G and an integer k, does G contain a spanning tree such that all vertices in the tree have degree *at most* k (obviously, only tree edges count towards the degree)? For example, in the following graph, there is no spanning tree such that all vertices have a degree less than three.

(a) Prove that the low-degree spanning tree problem is NP-hard with a reduction from Hamiltonian *path*.

(b) Now consider the *high-degree spanning tree problem*, which is as follows. Given a graph G and an integer k, does G contain a spanning tree whose highest degree vertex is *at least* k? In the previous example, there exists a spanning tree with a highest degree of 8. Give an efficient algorithm to solve the high-degree spanning tree problem, and an analysis of its time complexity.



9-10. *[4]* Show that the following problem is NP-complete:

Problem: Dense subgraph
Input: A graph G, and integers k and y.
Output: Does G contain a subgraph with exactly k vertices and at least y edges?

9-11. *[4]* Show that the following problem is NP-complete:

Problem: Clique, no-clique
Input: An undirected graph $G = (V, E)$ and an integer k.
Output: Does G contain both a clique of size k and an independent set of size k.

9-12. *[5]* An *Eulerian cycle* is a tour that visits every edge in a graph exactly once. An *Eulerian subgraph* is a subset of the edges and vertices of a graph that has an Eulerian cycle. Prove that the problem of finding the number of edges in the largest Eulerian subgraph of a graph is NP-hard. (Hint: the Hamiltonian circuit problem is NP-hard even if each vertex in the graph is incident upon exactly three edges.)

Creative Reductions

9-13. *[5]* Prove that the following problem is NP-complete:

Problem: Hitting Set
Input: A collection C of subsets of a set S, positive integer k.
Output: Does S contain a subset S' such that $|S'| \leq k$ and each subset in C contains at least one element from S'?

9-14. *[5]* Prove that the following problem is NP-complete:

Problem: Knapsack
Input: A set S of n items, such that the ith item has value v_i and weight w_i. Two positive integers: weight limit W and value requirement V.
Output: Does there exist a subset $S' \in S$ such that $\sum_{i \in S'} w_i \leq W$ and $\sum_{i \in S'} v_i \geq V$? (Hint: start from integer partition.)

9-15. *[5]* Prove that the following problem is NP-complete:

Problem: Hamiltonian Path
Input: A graph G, and vertices s and t.

Output: Does G contain a path which starts from s, ends at t, and visits all vertices without visiting any vertex more than once? (Hint: start from Hamiltonian cycle.)

9-16. *[5]* Prove that the following problem is NP-complete:

Problem: Longest Path
Input: A graph G and positive integer k.
Output: Does G contain a path that visits at least k different vertices without visiting any vertex more than once?

9-17. *[6]* Prove that the following problem is NP-complete:

Problem: Dominating Set
Input: A graph $G = (V, E)$ and positive integer k.
Output: Is there a subset $V' \in V$ such that $|V'| \leq k$ where for each vertex $x \in V$ either $x \in V'$ or there exists an edge (x, y), where $y \in V'$.

9-18. *[7]* Prove that the vertex cover problem (does there exist a subset S of k vertices in a graph G such that every edge in G is incident upon at least one vertex in S?) remains NP-complete even when all the vertices in the graph are restricted to have even degrees.

9-19. *[7]* Prove that the following problem is NP-complete:

Problem: Set Packing
Input: A collection C of subsets of a set S, positive integer k.
Output: Does S contain at least k disjoint subsets (i.e. , such that none of these subsets have any elements in common?)

9-20. *[7]* Prove that the following problem is NP-complete:

Problem: Feedback Vertex Set
Input: A directed graph $G = (V, A)$ and positive integer k.
Output: Is there a subset $V' \in V$ such that $|V'| \leq k$, such that deleting the vertices of V' from G leaves a DAG?

Algorithms for Special Cases

9-21. *[5]* A Hamiltonian path P is a path that visits each vertex exactly once. The problem of testing whether a graph G contains a Hamiltonian path is NP-complete. There does not have to be an edge in G from the ending vertex to the starting vertex of P, unlike in the Hamiltonian cycle problem.

Give an $O(n + m)$-time algorithm to test whether a directed acyclic graph G (a DAG) contains a Hamiltonian path. (Hint: think about topological sorting and DFS.)

9-22. *[8]* The 2-SAT problem is, given a Boolean formula in 2-conjunctive normal form (CNF), to decide whether the formula is satisfiable. 2-SAT is like 3-SAT, except

that each clause can have only two literals. For example, the following formula is in 2-CNF:

$(x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3)$

Give a polynomial-time algorithm to solve 2-SAT.

P=NP?

9-23. *[4]* Show that the following problems are in NP:

- Does graph G have a simple path (i.e. , with no vertex repeated) of length k?
- Is integer n composite (i.e. , not prime)?
- Does graph G have a vertex cover of size k?

9-24. *[7]* It was a long open question whether the decision problem "Is integer n a composite number, in other words, not prime?" can be computed in time polynomial in the size of its input. Why doesn't the following algorithm suffice to prove it is in P, since it runs in $O(n)$ time?

```
PrimalityTesting(n)
    composite := false
    for i := 2 to n − 1 do
        if (n mod i) = 0 then
            composite := true
```

Approximation Algorithms

9-25. *[4]* In the *maximum-satisfiability problem*, we seek a truth assignment that satisfies as many clauses as possible. Give an heuristic that always satisfies at least half as many clauses as the optimal solution.

9-26. *[5]* Consider the following heuristic for vertex cover. Construct a DFS tree of the graph, and delete all the leaves from this tree. What remains must be a vertex cover of the graph. Prove that the size of this cover is at most twice as large as optimal.

9-27. *[5]* The *maximum cut* problem for a graph $G = (V, E)$ seeks to partition the vertices V into disjoint sets A and B so as to maximize the number of edges $(a, b) \in E$ such that $a \in A$ and $b \in B$. Consider the following heuristic for max cut. First assign v_1 to A and v_2 to B. For each remaining vertex, assign it to the side that adds the most edges to the cut. Prove that this cut is at least half as large as the optimal cut.

9-28. *[5]* In the *bin-packing problem*, we are given n items with weights $w_1, w_2, ..., w_n$, respectively. Our goal is to find the smallest number of bins that will hold the n objects, where each bin has capacity of at most one kilogram.

The *first-fit heuristic* considers the objects in the order in which they are given. For each object, place it into first bin that has room for it. If no such bin exists, start a new bin. Prove that this heuristic uses at most twice as many bins as the optimal solution.

9-29. *[5]* For the first-fit heuristic described just above, give an example where the packing it fits uses at least 5/3 times as many bins as optimal.

9-30. *[5]* A *vertex coloring* of graph $G = (V, E)$ is an assignment of colors to vertices of V such that each edge (x, y) implies that vertices x and y are assigned different colors. Give an algorithm for vertex coloring G using at most $\Delta + 1$ colors, where Δ is the maximum vertex degree of G.

Programming Challenges

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

Geometry

9-1. "The Monocycle" – Programming Challenges 111202, UVA Judge 10047.

9-2. "Dog and Gopher" – Programming Challenges 10310, UVA Judge 111301.

9-3. "Chocolate Chip Cookies" – Programming Challenges 111304, UVA Judge 10136.

9-4. "Birthday Cake" – Programming Challenges 111305, UVA Judge 10167.

Computational Geometry

9-5. "Closest Pair Problem" – Programming Challenges 111402, UVA Judge 10245.

9-6. "Chainsaw Massacre" – Programming Challenges 111403, UVA Judge 10043.

9-7. "Hotter Colder" – Programming Challenges 111404, UVA Judge 10084.

9-8. "Useless Tile Packers" – Programming Challenges 111405, UVA Judge 10065.

Note: These are not particularly relevant to NP-completeness, but are added for completeness.

10

How to Design Algorithms

Designing the right algorithm for a given application is a major creative act—that of taking a problem and pulling a solution out of the ether. The space of choices you can make in algorithm design is enormous, leaving you plenty of freedom to hang yourself.

This book is designed to make you a better algorithm designer. The techniques presented in Part I of this book provide the basic ideas underlying all combinatorial algorithms. The problem catalog of Part II will help you with modeling your application, and tell you what is known about the relevant problems. However, being a successful algorithm designer requires more than book knowledge. It requires a certain attitude—the right problem-solving approach. It is difficult to teach this mindset in a book, yet getting it is essential to becoming a successful designer.

The key to algorithm design (or any other problem-solving task) is to proceed by asking yourself questions to guide your thought process. *What if we do this? What if we do that?* Should you get stuck on the problem, the best thing to do is move onto the next question. In any group brainstorming session, the most useful person in the room is the one who keeps asking, *"Why can't we do it this way?"* not the person who later tells them why, because she will eventually stumble on an approach that can't be shot down.

Towards this end, we provide a sequence of questions to guide your search for the right algorithm for your problem. To use it effectively, you must not only ask the questions, but answer them. The key is working through the answers carefully by writing them down in a log. The correct answer to *"Can I do it this way?"* is never "no," but "no, because. . . ." By clearly articulating your reasoning as to why something doesn't work, you can check whether you have glossed over a possibility that you didn't want to think hard enough about. It is amazing how often the reason

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_10,

you can't find a convincing explanation for something is because your conclusion is wrong.

The distinction between *strategy* and *tactics* is important to keep aware of during any design process. Strategy represents the quest for the big picture—the framework around which we construct our path to the goal. Tactics are used to win the minor battles we must fight along the way. In problem-solving, it is important to check repeatedly whether you are thinking on the right level. If you do not have a global strategy of how you are going to attack your problem, it is pointless to worry about the tactics. An example of a strategic question is "Can I model my application as a graph algorithm problem?" A tactical question might be, "Should I use an adjacency list or adjacency matrix data structure to represent my graph?" Of course, such tactical decisions are critical to the ultimate quality of the solution, but they can be properly evaluated only in light of a successful strategy.

Too many people freeze up in their thinking when faced with a design problem. After reading or hearing the problem, they sit down and realize that they *don't know what to do next*. Avoid this fate. Follow the sequence of questions provided below and in most of the catalog problem sections. We'll *tell* you what to do next!

Obviously, the more experience you have with algorithm design techniques such as dynamic programming, graph algorithms, intractability, and data structures, the more successful you will be at working through the list of questions. Part I of this book has been designed to strengthen this technical background. However, it pays to work through these questions regardless of how strong your technical skills are. The earliest and most important questions on the list focus on obtaining a detailed understanding of your problem and do not require specific expertise.

This list of questions was inspired by a passage in [Wol79]—a wonderful book about the space program entitled *The Right Stuff*. It concerned the radio transmissions from test pilots just before their planes crashed. One might have expected that they would panic, so ground control would hear the pilot yelling *Ahhhhhhhhhhh* —, terminated only by the sound of smacking into a mountain. Instead, the pilots ran through a list of what their possible actions could be. *I've tried the flaps. I've checked the engine. Still got two wings. I've reset the* —. They had "the Right Stuff." Because of this, they sometimes managed to miss the mountain.

I hope this book and list will provide you with "the Right Stuff" to be an algorithm designer. And I hope it prevents you from smacking into any mountains along the way.

1. Do I really understand the problem?
   (a) What exactly does the input consist of?
   (b) What exactly are the desired results or output?
   (c) Can I construct an input example small enough to solve by hand? What happens when I try to solve it?
   (d) How important is it to my application that I always find the optimal answer? Can I settle for something close to the optimal answer?

(e) How large is a typical instance of my problem? Will I be working on 10 items? 1,000 items? 1,000,000 items?

(f) How important is speed in my application? Must the problem be solved within one second? One minute? One hour? One day?

(g) How much time and effort can I invest in implementation? Will I be limited to simple algorithms that can be coded up in a day, or do I have the freedom to experiment with a couple of approaches and see which is best?

(h) Am I trying to solve a numerical problem? A graph algorithm problem? A geometric problem? A string problem? A set problem? Which formulation seems easiest?

2. Can I find a simple algorithm or heuristic for my problem?

   (a) Will brute force solve my problem *correctly* by searching through all subsets or arrangements and picking the best one?

      i. If so, why am I sure that this algorithm always gives the correct answer?

      ii. How do I measure the quality of a solution once I construct it?

      iii. Does this simple, slow solution run in polynomial or exponential time? Is my problem small enough that this brute-force solution will suffice?

      iv. Am I certain that my problem is sufficiently well defined to actually *have* a correct solution?

   (b) Can I solve my problem by repeatedly trying some simple rule, like picking the biggest item first? The smallest item first? A random item first?

      i. If so, on what types of inputs does this heuristic work well? Do these correspond to the data that might arise in my application?

      ii. On what types of inputs does this heuristic work badly? If no such examples can be found, can I show that it always works well?

      iii. How fast does my heuristic come up with an answer? Does it have a simple implementation?

3. Is my problem in the catalog of algorithmic problems in the back of this book?

   (a) What is known about the problem? Is there an implementation available that I can use?

   (b) Did I look in the right place for my problem? Did I browse through all pictures? Did I look in the index under all possible keywords?

(c) Are there relevant resources available on the World Wide Web? Did I do a Google web and Google Scholar search? Did I go to the page associated with this book, *http://www.cs.sunysb.edu/∼algorith*?

4. Are there special cases of the problem that I know how to solve?

   (a) Can I solve the problem efficiently when I ignore some of the input parameters?

   (b) Does the problem become easier to solve when I set some of the input parameters to trivial values, such as 0 or 1?

   (c) Can I simplify the problem to the point where I *can* solve it efficiently?

   (d) Why can't this special-case algorithm be generalized to a wider class of inputs?

   (e) Is my problem a special case of a more general problem in the catalog?

5. Which of the standard algorithm design paradigms are most relevant to my problem?

   (a) Is there a set of items that can be sorted by size or some key? Does this sorted order make it easier to find the answer?

   (b) Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?

   (c) Do the input objects or desired solution have a natural left-to-right order, such as characters in a string, elements of a permutation, or leaves of a tree? Can I use dynamic programming to exploit this order?

   (d) Are there certain operations being done repeatedly, such as searching, or finding the largest/smallest element? Can I use a data structure to speed up these queries? What about a dictionary/hash table or a heap/priority queue?

   (e) Can I use random sampling to select which object to pick next? What about constructing many random configurations and picking the best one? Can I use some kind of directed randomness like simulated annealing to zoom in on the best solution?

   (f) Can I formulate my problem as a linear program? How about an integer program?

   (g) Does my problem seem something like satisfiability, the traveling salesman problem, or some other NP-complete problem? Might the problem be NP-complete and thus not have an efficient algorithm? Is it in the problem list in the back of Garey and Johnson [GJ79]?

6. Am I still stumped?

   (a) Am I willing to spend money to hire an expert to tell me what to do? If so, check out the professional consulting services mentioned in Section 19.4 (page 664).

   (b) Why don't I go back to the beginning and work through these questions again? Did any of my answers change during my latest trip through the list?

Problem-solving is not a science, but part art and part skill. It is one of the skills most worth developing. My favorite book on problem-solving remains Pólya's *How to Solve It* [P57], which features a catalog of problem-solving techniques that are fascinating to browse through, both before and after you have a problem.

11

A Catalog of Algorithmic Problems

This is a catalog of algorithmic problems that arise commonly in practice. It describes what is known about them and gives suggestions about how best to proceed if the problem arises in your application.

What is the best way to use this catalog? First, think about your problem. If you recall the name, look up the catalog entry in the index or table of contents and start reading. Read through the entire entry, since it contains pointers to other relevant problems. Leaf through the catalog, looking at the pictures and problem names to see if something strikes a chord. Don't be afraid to use the index, for every problem in the book is listed there under several possible keywords and applications. If you *still* don't find something relevant, your problem is either not suitable for attack by combinatorial algorithms or else you don't fully understand it. In either case, go back to step one.

The catalog entries contain a variety of different types of information that have never been collected in one place before. Different fields in each entry present information of practical and historical interest.

To make this catalog more easily accessible, we introduce each problem with a pair of graphics representing the problem instance or input on the left and the result of solving the problem in this instance on the right. We have invested considerable thought in creating stylized examples that illustrate desired behaviors more than just definitions. For example, the minimum spanning tree example illustrates how points can be clustered using minimum spanning trees. We hope that people will be able to flip through the pictures and identify which problems might be relevant to them. We augment these pictures with more formal written input and problem descriptions to eliminate the ambiguity inherent in a purely pictorial representation.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_11,

Once you have identified your problem, the discussion section tells you what you should do about it. We describe applications where the problem is likely to arise and the special issues associated with data from them. We discuss the kind of results you can hope for or expect and, most importantly, what you should do to get them. For each problem, we outline a quick-and-dirty algorithm and provide pointers to more powerful algorithms to try next if the first attempt is not sufficient.

For each problem, we identify available software implementations that are discussed in the implementation field of each entry. Many of these routines are quite good, and they can perhaps be plugged directly into your application. Others maybe inadequate for production use, but they hopefully can provide a good model for your own implementation. In general, the implementations are listed in order of descending usefulness, but we will explicitly recommend the best one available for each problem if a clear winner exists. More detailed information for many of these implementations appears in Chapter 19. Essentially all of the implementations are available via the WWW site associated with this book—reachable at *http://www.cs.sunysb.edu/~algorith*.

Finally, in deliberately smaller print, we discuss the history of each problem and present results of primarily theoretical interest. We have attempted to report the best results known for each problem and point out empirical comparisons of algorithms or survey articles if they exist. This should be of interest to students and researchers, and also to practitioners for whom our recommended solutions prove inadequate and need to know if anything better is possible.

Caveats

This is a catalog of algorithmic problems. It is not a cookbook. It cannot be because there are too many recipes and too many possible variations on what people want to eat. My goal is to point you in the right direction so that you can solve your own problems. I try to identify the issues you will encounter along the way—problems that you will have to work out for yourself. In particular:

- For each problem, I suggest algorithms and directions to attack it. These recommendations are based on my experiences, and are aimed toward what I see as typical applications. I felt it was more important to make concrete recommendations for the masses rather than to try to cover all possible situations. If you don't agree with my advice, don't follow it, but before you ignore me, try to understand the reasoning behind my recommendations and articulate a reason why your application violates my assumptions.

- The implementations I recommend are not necessarily complete solutions to your problem. I point to an implementation whenever I feel it might be more useful to someone than just a textbook description of the algorithm. Some programs are useful only as models for you to write your own codes. Others are embedded in large systems and so might be too painful to extract and run

on their own. Assume that all of them contain bugs. Many are quite serious, so beware.

- Please respect the licensing conditions for any implementations you use commercially. Many of these codes are not open source and have licence restrictions. See Section 19.1 for a further discussion of this issue.
- I would be interested in hearing about your experiences with my recommendations, both positive and negative. I would be especially interested in learning about any other implementations that you know about. Feel free to drop me a line at feedback@algorist.com.

12

Data Structures

Data structures are not so much algorithms as they are the fundamental constructs around which you build your application. Becoming fluent in what the standard data structures can do for you is essential to get full value from them.

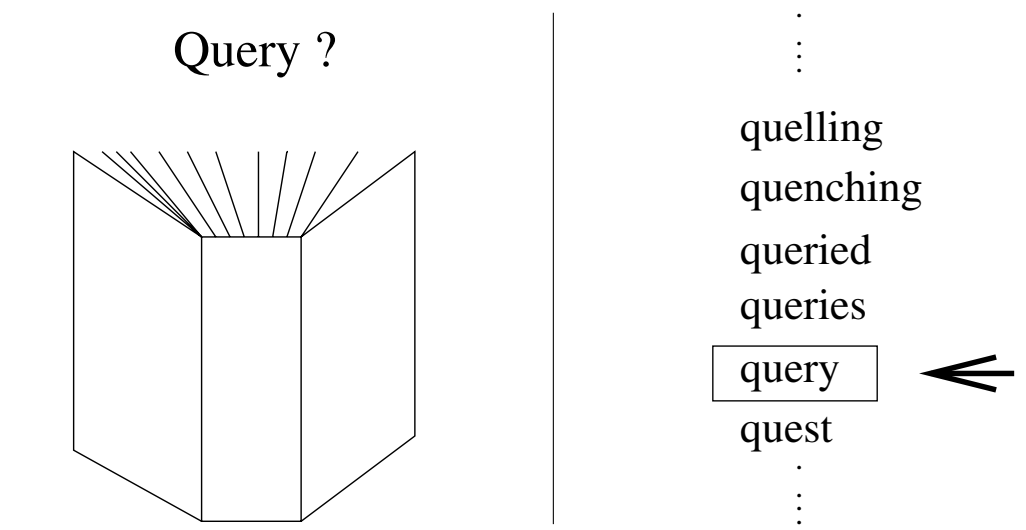
This puts data structures slightly out of sync with the rest of the catalog. Perhaps the most useful aspect of it will be the pointers to various implementations and data structure libraries. Many of these data structures are nontrivial to implement well, so the programs we point to will be useful as models even if they do not do exactly what you need. Certain fundamental data structures, like kd-trees and suffix trees, are not as well known as they should be. Hopefully, this catalog will serve to better publicize them.

There are a large number of books on elementary data structures available. My favorites include:

- *Sedgewick* [Sed98] – This comprehensive introduction to algorithms and data structures stands out for the clever and beautiful images of algorithms in action. It comes in C, C++, and Java editions.
- *Weiss* [Wei06] – A nice text, emphasizing data structures more than algorithms. Comes in Java, C, C++, and Ada editions.
- *Goodrich and Tamassia* [GT05] – The Java edition makes particularly good use of the author's Java Data Structures Library (JDSL).

The *Handbook of Data Structures and Applications* [MS05] provides a comprehensive and up-to-date survey of research in data structures. The student who took only an elementary course in data structures is likely to be impressed and surprised by the volume of recent work on the subject.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_12,



12.1 Dictionaries

Input description: A set of n records, each identified by one or more key fields.

Problem description: Build and maintain a data structure to efficiently locate, insert, and delete the record associated with any query key q.

Discussion: The abstract data type "dictionary" is one of the most important structures in computer science. Dozens of data structures have been proposed for implementing dictionaries, including hash tables, skip lists, and balanced/unbalanced binary search trees. This means that choosing the best one can be tricky. It can significantly impact performance. *However, in practice, it is more important to avoid using a bad data structure than to identify the single best option available.*

An essential piece of advice is to carefully isolate the implementation of the dictionary data structure from its interface. Use explicit calls to methods or subroutines that initialize, search, and modify the data structure, rather than embedding them within the code. This leads to a much cleaner program, but it also makes it easy to experiment with different implementations to see how they perform. Do not obsess about the procedure call overhead inherent in such an abstraction. If your application is so time-critical that such overhead can impact performance, then it is even more essential that you be able to identify the right dictionary implementation.

In choosing the right data structure for your dictionary, ask yourself the following questions:

- *How many items will you have in your data structure?* – Will you know this number in advance? Are you looking at a problem small enough that a simple data structure will suffice, or one so large that we must worry about running out of memory or virtual memory performance?

- *Do you know the relative frequencies of insert, delete, and search operations?* – Static data structures (like sorted arrays) suffice in applications when there are no modifications to the data structure after it is first constructed. *Semi-dynamic* data structures, which support insertion but not deletion, can have significantly simpler implementations than fully dynamic ones.

- *Can we assume that the access pattern for keys will be uniform and random?* – Search queries exhibit a skewed access distribution in many applications, meaning certain elements are much more popular than others. Further, queries often have a sense of temporal locality, meaning elements are likely to be repeatedly accessed in clusters instead of at fairly regular intervals. Certain data structures (such as splay trees) can take advantage of a skewed and clustered universe.

- *Is it critical that individual operations be fast, or only that the total amount of work done over the entire program be minimized?* – When response time is critical, such as in a program controlling a heart-lung machine, you can't wait too long between steps. When you have a program that is doing a lot of queries over the database, such as identifying all criminals who happen to be politicians, it is not as critical that you pick out any particular congressman quickly as it is that you get them all with the minimum total effort.

An object-oriented generation has emerged as no more likely to write a container class than fix the engine in their car. This is good; default containers should do just fine for most applications. Still, it is good sometimes to know what you have under the hood:

- *Unsorted linked lists or arrays* – For small data sets, an unsorted array is probably the easiest data structure to maintain. Linked structures can have terrible cache performance compared with sleek, compact arrays. However, once your dictionary becomes larger than (say) 50 to 100 items, the linear search time will kill you for either lists or arrays. Details of elementary dictionary implementations appear in Section 3.3 (page 72).

  A particularly interesting and useful variant is the *self-organizing list*. Whenever a key is accessed or inserted, we always move it to head of the list. Thus, if the key is accessed again sometime in the near future, it will be near the front and so require only a short search to find it. Most applications exhibit

both uneven access frequencies and locality of reference, so the average time for a successful search in a self-organizing list is typically much better than in a sorted or unsorted list. Of course, self-organizing data structures can be built from arrays as well as linked lists and trees.

- *Sorted linked lists or arrays* – Maintaining a sorted linked list is usually not worth the effort unless you are trying to eliminate duplicates, since we cannot perform binary searches in such a data structure. A sorted array will be appropriate if and only if there are not many insertions or deletions.

- *Hash tables* – For applications involving a moderate-to-large number of keys (say between 100 and 10,000,000), a hash table is probably the right way to go. We use a function that maps keys (be they strings, numbers, or whatever) to integers between 0 and $m-1$. We maintain an array of m buckets, each typically implemented using an unsorted linked list. The hash function immediately identifies which bucket contains a given key. If we use a hash function that spreads the keys out nicely, and a sufficiently large hash table, each bucket should contain very few items, thus making linear searches acceptable. Insertion and deletion from a hash table reduce to insertion and deletion from the bucket/list. Section 3.7 (page 89) provides a more detailed discussion of hashing and its applications.

  A well-tuned hash table will outperform a sorted array in most applications. However, several design decisions go into creating a well-tuned hash table:

  - *How do I deal with collisions?:* Open addressing can lead to more concise tables with better cache performance than bucketing, but performance will be more brittle as the load factor (ratio of occupancy to capacity) of the hash table starts to get high.
  - *How big should the table be?:* With bucketing, m should about the same as the maximum number of items you expect to put in the table. With open addressing, make it (say) 30% larger or more. Selecting m to be a prime number minimizes the dangers of a bad hash function.
  - *What hash function should I use?:* For strings, something like

    $$H(S,j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j}) \bmod m$$

    should work, where α is the size of the alphabet and *char(x)* is the function that maps each character x to its ASCII character code. Use Horner's rule (or precompute values of α^x) to implement this hash function computation efficiently, as discussed in Section 13.9 (page 423). This hash function has the nifty property that

    $$H(S,j+1) = (H(S,j) - \alpha^{m-1} char(s_j))\alpha + s_{j+m}$$

so hash codes of successive m-character windows of a string can be computed in constant time instead of $O(m)$.

Regardless of which hash function you decide to use, print statistics on the distribution of keys per bucket to see how uniform it *really* is. Odds are the first hash function you try will not prove to be the best. Botching up the hash function is an excellent way to slow down any application.

- *Binary search trees* – Binary search trees are elegant data structures that support fast insertions, deletions, and queries. They are reviewed in Section 3.4 (page 77). The big distinction between different types of trees is whether they are explicitly rebalanced after insertion or deletion, and how this rebalancing is done. In *random search trees*, we simply insert a node at the leaf position where we can find it and no rebalancing takes place. Although such trees perform well under random insertions, most applications are not really random. Indeed, unbalanced search trees constructed by inserting keys in sorted order are a disaster, performing like a linked list.

  Balanced search trees use local *rotation* operations to restructure search trees, moving more distant nodes closer to the root while maintaining the in-order search structure of the tree. Among balanced search trees, AVL and 2/3 trees are now passé, and *red-black trees* seem to be more popular. A particularly interesting self-organizing data structure is the *splay tree*, which uses rotations to move any accessed key to the root. Frequently used or recently accessed nodes thus sit near the top of the tree, allowing faster searches.

  Bottom line: Which tree is best for your application? Probably the one of which you have the best implementation. The flavor of balanced tree is probably not as important as the skill of the programmer who coded it.

- *B-trees* – For data sets so large that they will not fit in main memory (say more than 1,000,000 items) your best bet will be some flavor of a B-tree. Once a data structure has to be stored outside of main memory, the search time grows by several orders of magnitude. With modern cache architectures, similar effects can also happen on a smaller scale, since cache is much faster than RAM.

  The idea behind a B-tree is to collapse several levels of a binary search tree into a single large node, so that we can make the equivalent of several search steps before another disk access is needed. With B-tree we can access enormous numbers of keys using only a few disk accesses. To get the full benefit from using a B-tree, it is important to understand how the secondary storage device and virtual memory interact, through constants such as page size and virtual/real address space. *Cache-oblivious algorithms* (described below) can mitigate such concerns.

Even for modest-sized data sets, unexpectedly poor performance of a data structure may result from excessive swapping, so listen to your disk to help decide whether you should be using a B-tree.

- *Skip lists* – These are somewhat of a cult data structure. A hierarchy of sorted linked lists is maintained, where a coin is flipped for each element to decide whether it gets copied into the next highest list. This implies roughly $\lg n$ lists, each roughly half as large as the one above it. Search starts in the smallest list. The search key lies in an interval between two elements, which is then explored in the next larger list. Each searched interval contains an expected constant number of elements per list, for a total expected $O(\lg n)$ query time. The primary benefits of skip lists are ease of analysis and implementation relative to balanced trees.

Implementations: Modern programming languages provide libraries offering complete and efficient container implementations. The C++ *Standard Template Library* (STL) is now provided with most compliers, and available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01], and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

LEDA (see Section 19.1.1 (page 658)) provides an extremely complete collection of dictionary data structures in C++, including hashing, perfect hashing, B-trees, red-black trees, random search trees, and skip lists. Experiments reported in [MN99] identified hashing as the best dictionary choice, with skip lists and 2-4 trees (a special case of B-trees) as the most efficient tree-like structures.

Java Collections (JC), a small library of data structures, is included in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*). The *Data Structures Library in Java* (JDSL) is more comprehensive, and available for non-commercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL.

Notes: Knuth [Knu97a] provides the most detailed analysis and exposition on fundamental dictionary data structures, but misses certain modern data structures as red-black and splay trees. Spending some time with his books is a worthwhile rite of passage for all computer science students.

The Handbook of Data Structures and Applications [MS05] provides up-to-date surveys on all aspects of dictionary data structures. Other surveys include Mehlhorn and Tsakalidis [MT90b] and Gonnet and Baeza-Yates [GBY91]. Good textbook expositions on dictionary data structures include Sedgewick [Sed98], Weiss [Wei06], and Goodrich/Tamassia [GT05]. We defer to all these sources to avoid giving original references for each of the data structures described above.

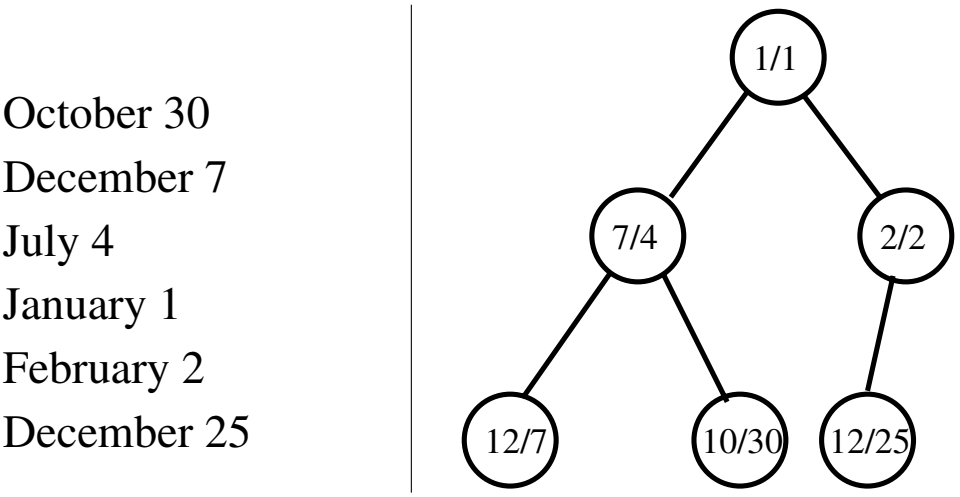
The 1996 DIMACS implementation challenge focused on elementary data structures, including dictionaries [GJM02]. Data sets, and codes are accessible from *http://dimacs.rutgers.edu/Challenges*.

The cost of transferring data back and forth between levels of the memory hierarchy (RAM-to-cache or disk-to-RAM) dominates the cost of actual computation for many

problems. Each data transfer moves one block of size b, so efficient algorithms seek to minimize the number of block transfers. The complexity of fundamental algorithm and data structure problems on such an external memory model has been extensively studied [Vit01]. *Cache-oblivious* data structures offer performance guarantees under such a model without explicit knowledge of the block-size parameter b. Hence, good performance can be obtained on any machine without architecture-specific tuning. See [ABF05] for an excellent survey on cache-oblivious data structures.

Several modern data structures, such as splay trees, have been studied using *amortized analysis*, where we bound the total amount of time used by any sequence of operations. In an amortized analysis, a single operation can be very expensive, but only because we have already benefited from enough cheap operations to pay off the higher cost. A data structure realizing an amortized complexity of $O(f(n))$ is less desirable than one whose worst-case complexity is $O(f(n))$ (since a very bad operation might still occur) but better than one with an average-case complexity $O(f(n))$, since the amortized bound will achieve this average on any input.

Related Problems: Sorting (see page 436), searching (see page 441).



INPUT OUTPUT

12.2 Priority Queues

Input description: A set of records with numerically or otherwise totally-ordered keys.

Problem description: Build and maintain a data structure for providing quick access to the *smallest* or *largest* key in the set.

Discussion: Priority queues are useful data structures in simulations, particularly for maintaining a set of future events ordered by time. They are called "priority" queues because they enable you to retrieve items not by the insertion time (as in a stack or queue), nor by a key match (as in a dictionary), but by which item has the highest priority of retrieval.

If your application will perform no insertions after the initial query, there is no need for an explicit priority queue. Simply sort the records by priority and proceed from top to bottom, maintaining a pointer to the last record retrieved. This situation occurs in Kruskal's minimum spanning tree algorithm, or when simulating a completely scripted set of events.

However, if you are mixing insertions, deletions, and queries, you will need a real priority queue. The following questions will help select the right one:

- *What other operations do you need?* – Will you be searching for arbitrary keys, or just searching for the smallest? Will you be deleting arbitrary elements from the data, or just repeatedly deleting the top or smallest element?

- *Do you know the maximum data structure size in advance?* – The issue here is whether you can preallocate space for the data structure.
- *Might you change the priority of elements already in the queue?* – Changing the priority of elements implies that we must be able to retrieve elements from the queue based on their key, in addition to being able to retrieve the largest element.

Your choices are between the following basic priority queue implementations:

- *Sorted array or list* – A sorted array is very efficient to both identify the smallest element and delete it by decrementing the top index. However, maintaining the total order makes inserting new elements slow. Sorted arrays are only suitable when there will be few insertions into the priority queue. Basic priority queue implementations are reviewed in Section 3.5 (page 83).
- *Binary heaps* – This simple, elegant data structure supports both insertion and extract-min in $O(\lg n)$ time each. Heaps maintain an implicit binary tree structure in an array, such that the key of the root of any subtree is less than that of all its descendents. Thus, the minimum key always sits at the top of the heap. New keys can be inserted by placing them at an open leaf and percolating the element upwards until it sits at its proper place in the partial order. An implementation of binary heap construction and retrieval in C appears in Section 4.3.1 (page 109)

  Binary heaps are the right answer when you know an upper bound on the number of items in your priority queue, since you must specify array size at creation time. Even this constraint can be mitigated by using dynamic arrays (see Section 3.1.1 (page 66)).
- *Bounded height priority queue* – This array-based data structure permits constant-time insertion and find-min operations whenever the range of possible key values is limited. Suppose we know that all key values will be integers between 1 and n. We can set up an array of n linked lists, such that the ith list serves as a bucket containing all items with key i. We will maintain a *top* pointer to the smallest nonempty list. To insert an item with key k into the priority queue, add it to the kth bucket and set $top = \min(top, k)$. To extract the minimum, report the first item from bucket *top*, delete it, and move *top* down if the bucket has become empty.

  Bounded height priority queues are very useful in maintaining the vertices of a graph sorted by degree, which is a fundamental operation in graph algorithms. Still, they are not as widely known as they should be. They are usually the right priority queue for any small, discrete range of keys.
- *Binary search trees* – Binary search trees make effective priority queues, since the smallest element is always the leftmost leaf, while the largest element is

always the rightmost leaf. The min (max) is found by simply tracing down left (right) pointers until the next pointer is nil. Binary tree heaps prove most appropriate when you need other dictionary operations, or if you have an unbounded key range and do not know the maximum priority queue size in advance.

- *Fibonacci and pairing heaps* – These complicated priority queues are designed to speed up *decrease-key* operations, where the priority of an item already in the priority queue is reduced. This arises, for example, in shortest path computations when we discover a shorter route to a vertex v than previously established.

  Properly implemented and used, they lead to better performance on very large computations.

Implementations: Modern programming languages provide libraries offering complete and efficient priority queue implementations. Member functions `push`, `top`, and `pop` of the C++ *Standard Template Library* (STL) `priority_queue` template mirror heap operations `insert`, `findmax`, and `deletemax`. STL is available with documentation at *http://www.sgi.com/tech/stl/*. See Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL.

LEDA (see Section 19.1.1 (page 658)) provides a complete collection of priority queues in C++, including Fibonacci heaps, pairing heaps, Emde-Boas trees, and bounded height priority queues. Experiments reported in [MN99] identified simple binary heaps as quite competitive in most applications, with pairing heaps beating Fibonacci heaps in head-to-head tests.

The *Java Collections* `PriorityQueue` class is included in the java.util package of Java standard edition (*http://java.sun.com/javase/*). The *Data Structures Library in Java* (JDSL) provides an alternate implementation, and is available for non-commercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL.

Sanders [San00] did extensive experiments demonstrating that his sequence heap, based on k-way merging, was roughly twice as fast as a well-implemented binary heap. See *http://www.mpi-inf.mpg.de/∼sanders/programs/spq/* for his implementations in C++.

Notes: *The Handbook of Data Structures and Applications* [MS05] provides several up-to-date surveys on all aspects of priority queues. Empirical comparisons between priority queue data structures include [CGS99, GBY91, Jon86, LL96, San00].

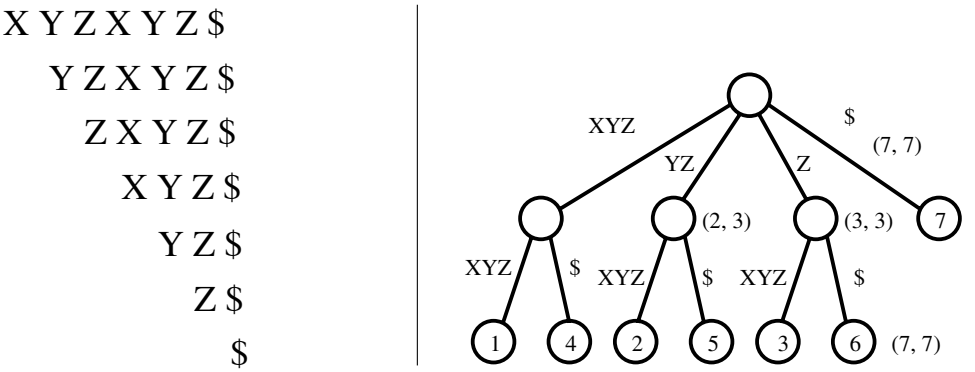
Double-ended priority queues extend the basic heap operations to simultaneously support both find-min and find-max. See [Sah05] for a survey of four different implementations of double-ended priority queues.

Bounded-height priority queues are useful data structures in practice, but do not promise good worst-case performance when arbitrary insertions and deletions are permitted. However, von Emde Boas priority queues [vEBKZ77] support $O(\lg \lg n)$ insertion, deletion, search, max, and min operations where each key is an element from 1 to n.

Fibonacci heaps [FT87] support insert and decrease-key operations in constant amortized time, with $O(\lg n)$ amortized time extract-min and delete operations. The constant-time decrease-key operation leads to faster implementations of classical algorithms for shortest-paths, weighted bipartite-matching, and minimum spanning tree. In practice, Fibonacci heaps are difficult to implement and have large constant factors associated with them. However, pairing heaps appear to realize the same bounds with less overhead. Experiments with pairing heaps are reported in [SV87].

Heaps define a partial order that can be built using a linear number of comparisons. The familiar linear-time merging algorithm for heap construction is due to Floyd [Flo64]. In the worst case, $1.625n$ comparisons suffice [GM86] and $1.5n - O(\lg n)$ comparisons are necessary [CC92].

Related Problems: Dictionaries (see page 367), sorting (see page 436), shortest path (see page 489).



INPUT OUTPUT

12.3 Suffix Trees and Arrays

Input description: A reference string S.

Problem description: Build a data structure to quickly find all places where an arbitrary query string q occurs in S.

Discussion: Suffix trees and arrays are phenomenally useful data structures for solving string problems elegantly and efficiently. Proper use of suffix trees often speeds up string processing algorithms from $O(n^2)$ to linear time—likely the answer. Indeed, suffix trees are the hero of the war story reported in Section 3.9 (page 94).

In its simplest instantiation, a suffix tree is simply a *trie* of the n suffixes of an n-character string S. A trie is a tree structure, where each edge represents one character, and the root represents the null string. Thus, each path from the root represents a string, described by the characters labeling the edges traversed. Any finite set of words defines a trie, and two words with common prefixes branch off from each other at the first distinguishing character. Each leaf denotes the end of a string. Figure 12.1 illustrates a simple trie.

Tries are useful for testing whether a given query string q is in the set. We traverse the trie from the root along branches defined by successive characters of q. If a branch does not exist in the trie, then q cannot be in the set of strings. Otherwise we find q in $|q|$ character comparisons *regardless* of how many strings are in the trie. Tries are very simple to build (repeatedly insert new strings) and very fast to search, although they can be expensive in terms of memory.

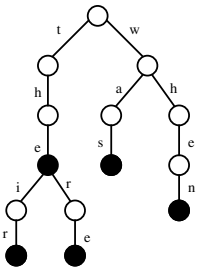


Figure 12.1: A trie on strings *the, their, there, was,* and *when*

A *suffix tree* is simply a trie of all the proper suffixes of S. The suffix tree enables you to test whether q is a substring of S, because any substring of S is the prefix of some suffix (got it?). The search time is again linear in the length of q.

The catch is that constructing a full suffix tree in this manner can require $O(n^2)$ time and, even worse, $O(n^2)$ space, since the average length of the n suffixes is $n/2$. However, linear space suffices to represent a full suffix tree, if we are clever. Observe that most of the nodes in a trie-based suffix tree occur on simple paths between branch nodes in the tree. Each of these simple paths corresponds to a substring of the original string. By storing the original string in an array and collapsing each such path into a single edge, we have all the information of the full suffix tree in only $O(n)$ space. The label for each edge is described by the starting and ending array indices representing the substring. The output figure for this section displays a collapsed suffix tree in all its glory.

Even better, there exist $O(n)$ algorithms to construct this collapsed tree, by making clever use of pointers to minimize construction time. These additional pointers can also be used to speed up many applications of suffix trees.

But what can you do with suffix trees? Consider the following applications. For more details see the books by Gusfield [Gus97] or Crochemore and Rytter [CR03]:

- *Find all occurrences of q as a substring of S* – Just as with a trie, we can walk from the root to the node n_q associated with q. The positions of all occurrences of q in S are represented by the descendents of n_q, which can be identified using a depth-first search from n_q. In collapsed suffix trees, it takes $O(|q| + k)$ time to find the k occurrences of q in S.

- *Longest substring common to a set of strings* – Build a single collapsed suffix tree containing all suffixes of all strings, with each leaf labeled with its original string. In the course of doing a depth-first search on this tree, we can label each node with both the length of its common prefix and the number of distinct strings that are children of it. From this information, the best node can be selected in linear time.

- *Find the longest palindrome in S* – A *palindrome* is a string that reads the same if the order of characters is reversed, such as *madam*. To find the longest palindrome in a string S, build a single suffix tree containing all suffixes of S and the reversal of S, with each leaf identified by its starting position. A palindrome is defined by any node in this tree that has forward and reversed children from the same position.

Since linear time suffix tree construction algorithms are nontrivial, I recommend using an existing implementation. Another good option is to use suffix arrays, discussed below.

Suffix arrays do most of what suffix trees do, while using roughly four times less memory. They are also easier to implement. A suffix array is in principle just an array that contains all the n suffixes of S in sorted order. Thus a binary search of this array for string q suffices to locate the prefix of a suffix that matches q, permitting an efficient substring search in $O(\lg n)$ string comparisons. With the addition of an index specifying the common prefix length of all bounding suffixes, only $\lg n + |q|$ *character* comparisons need be performed on any query, since we can identify the next character that must be tested in the binary search. For example, if the lower range of the search is *cowabunga* and the upper range is *cowslip*, all keys in between must share the same first three letters, so only the fourth character of any intermediate key must be tested against q. In practice, suffix arrays are typically as fast or faster to search than suffix trees.

Suffix arrays use less memory than suffix trees. Each suffix is represented completely by its unique starting position (from 1 to n) and read off as needed using a single reference copy of the input string.

Some care must be taken to construct suffix arrays efficiently, however, since there are $O(n^2)$ characters in the strings being sorted. One solution is to first build a suffix *tree*, then perform an in-order traversal of it to read the strings off in sorted order! However, recent breakthroughs have lead to space/time efficient algorithms for constructing suffix arrays directly.

Implementations: There now exist a wealth of suffix array implementations available. Indeed, all of the recent linear time construction algorithms have been implemented and benchmarked [PST07]. Schürmann and Stoye [SS07] provide an excellent C implementation at *http://bibiserv.techfak.uni-bielefeld.de/bpr/*.

No less than eight different C/C++ implementations of compressed text indexes appear at the *Pizza&Chili corpus* (*http://pizzachili.di.unipi.it/*). These data structures go to great lengths to minimize space usage, typically compressing the input string to near the empirical entropy while still achieving excellent query times!

Suffix tree implementations are also readily available. A `SuffixTree` class is provided in BioJava (*ttp://www.biojava.org/*)—an open source project providing a Java framework for processing biological data. `Libstree` is a C implementation of Ukkonen's algorithm, available at *http://www.icir.org/christian/libstree/*.

Nelson's C++ code [Nel96] is available from *http://marknelson.us/1996/08/01/suffix-trees/*.

Strmat is a collection of C programs implementing exact pattern matching algorithms in association with [Gus97], including an implementation of suffix trees. It is available at *http://www.cs.ucdavis.edu/~gusfield/strmat.html*.

Notes: Tries were first proposed by Fredkin [Fre62], the name coming from the central letters of the word "retrieval." A survey of basic trie data structures with extensive references appears in [GBY91].

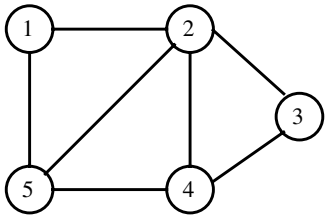
Efficient algorithms for suffix tree construction are due to Weiner [Wei73], McCreight [McC76], and Ukkonen [Ukk92]. Good expositions on these algorithms include Crochmore and Rytter [CR03] and Gusfield [Gus97].

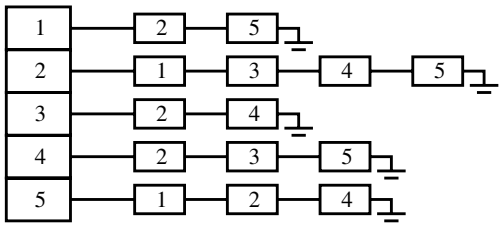
Suffix arrays were invented by Manber and Myers [MM93], although an equivalent idea called *Pat trees* due to Gonnet and Baeza-Yates appears in [GBY91]. Three teams independently emerged with linear-time suffix array algorithms in 2003 [KSPP03, KA03, KSB05], and progress has continued rapidly. See [PST07] for a recent survey covering all these developments.

Recent work has resulted in the development of compressed full text indexes that offer essentially all the power of suffix trees/arrays in a data structure whose size is proportional to the *compressed* text string. Makinen and Navarro [MN07] survey these remarkable data structures.

The power of suffix trees can be further augmented by using a data structure for computing the *least common ancestor (LCA)* of any pair of nodes x, y in a tree in constant time, after linear-time preprocessing of the tree. The original data structure due to Harel and Tarjan [HT84], has been progressively simplified by Schieber and Vishkin [SV88] and later Bender and Farach [BF00]. Expositions include Gusfield [Gus97]. The least common ancestor of two nodes in a suffix tree or trie defines the node representing the longest common prefix of the two associated strings. That we can answer such queries in constant time is amazing, and proves useful as a building block for many other algorithms.

Related Problems: String matching (see page 628), text compression (see page 637), longest common substring (see page 650).





INPUT OUTPUT

12.4 Graph Data Structures

Input description: A graph G.

Problem description: Represent the graph G using a flexible, efficient data structure.

Discussion: The two basic data structures for representing graphs are *adjacency matrices* and *adjacency lists*. Full descriptions of these data structures appear in Section 5.2 (page 151), along with an implementation of adjacency lists in C. In general, for most things, adjacency lists are the way to go.

The issues in deciding which data structure to use include:

- *How big will your graph be?* – How many vertices will it have, both typically and in the worse case? Ditto for the number of edges? Graphs with 1,000 vertices imply adjacency matrices with 1,000,000 entries. This seems too be the boundary of reality. Adjacency matrices make sense only for small or very dense graphs.

- *How dense will your graph be?* – If your graph is very dense, meaning that a large fraction of the vertex pairs define edges, there is probably no compelling reason to use adjacency lists. You will be doomed to using $\Theta(n^2)$ space anyway. Indeed, for complete graphs, matrices will be more concise due to the elimination of pointers.

- *Which algorithms will you be implementing?* – Certain algorithms are more natural on adjacency matrices (such as all-pairs shortest path) and others favor adjacency lists (such as most DFS-based algorithms). Adjacency matrices win for algorithms that repeatedly ask, "Is (i, j) in G?" However, most graph algorithms can be designed to eliminate such queries.

- *Will you be modifying the graph over the course of your application?* – Efficient *static graph* implementations can be used when no edge insertion/deletion operations will done following initial construction. Indeed, more

common than modifying the topology of the graph is modifying the *attributes* of a vertex or edge of the graph, such as size, weight, label, or color. Attributes are best handled as extra fields in the vertex or edge records of adjacency lists.

Building a good general purpose graph type is a substantial project. For this reason, we suggest that you check out existing implementations (particularly LEDA) before hacking up your own. Note that it costs only time linear in the size of the larger data structure to convert between adjacency matrices and adjacency lists. This conversion is unlikely to be the bottleneck in any application, so you may decide to use both data structures if you have the space to store them. This usually isn't necessary, but might prove simplest if you are confused about the alternatives.

Planar graphs are those that can be drawn in the plane so no two edges cross. Graphs arising in many applications are planar by definition, such as maps of countries. Others are planar by happenstance, like trees. Planar graphs are always sparse, since any n-vertex planar graph can have at most $3n - 6$ edges. Thus they should be represented using adjacency lists. If the planar drawing (or *embedding*) of the graph is fundamental to what is being computed, planar graphs are best represented geometrically. See Section 15.12 (page 520) for algorithms for constructing planar embeddings from graphs. Section 17.15 (page 614) discusses algorithms for maintaining the graphs implicit in the arrangements of geometric objects like lines and polygons.

Hypergraphs are generalized graphs where each edge may link subsets of more than two vertices. Suppose we want to represent who is on which Congressional committee. The vertices of our hypergraph would be the individual congressmen, while each hyperedge would represent one committee. Such arbitrary collections of subsets of a set are naturally thought of as hypergraphs.

Two basic data structures for hypergraphs are:

- *Incidence matrices*, which are analogous to adjacency matrices. They require $n \times m$ space, where m is the number of hyperedges. Each row corresponds to a vertex, and each column to an edge, with a nonzero entry in $M[i, j]$ iff vertex i is incident to edge j. On standard graphs there are two nonzero entries in each column. The degree of each vertex governs the number of nonzero entries in each row.

- *Bipartite incidence structures*, which are analogous to adjacency lists, and hence suited for sparse hypergraphs. There is a vertex of the incidence structure associated with each edge and vertex of the hypergraphs, and an edge (i, j) in the incidence structure if vertex i of the hypergraph appears in edge j of the hypergraph. Adjacency lists are typically used to represent this incidence structure. Drawing the associated bipartite graph provides a natural way to visualize the hypergraph.

Special efforts must be taken to represent very large graphs efficiently. However, interesting problems have been solved on graphs with millions of edges and

vertices. The first step is to make your data structure as lean as possible, by packing your adjacency matrix in a bit vector (see Section 12.5 (page 385)) or removing unnecessary pointers from your adjacency list representation. For example, in a static graph (which does not support edge insertions or deletions) each edge list can be replaced by a packed array of vertex identifiers, thus eliminating pointers and potentially saving half the space.

If your graph is extremely large, it may become necessary to switch to a hierarchical representation, where the vertices are clustered into subgraphs that are compressed into single vertices. Two approaches exist to construct such a hierarchical decomposition. The first breaks the graph into components in a natural or application-specific way. For example, a graph of roads and cities suggests a natural decomposition—partition the map into districts, towns, counties, and states. The other approach runs a graph partition algorithm discussed as in Section 16.6 (page 541). If you have one, a natural decomposition will likely do a better job than some naive heuristic for an NP-complete problem. If your graph is really unmanageably large, you cannot afford to do a very good job of algorithmically partitioning it. First verify that standard data structures fail on your problem before attempting such heroic measures.

Implementations: LEDA (see Section 19.1.1 (page 658)) provides the best graph data type currently implemented in C++. It is now a commercial product. You should at least study the methods it provides for graph manipulation, so as to see how the right level of abstract graph type makes implementing algorithms very clean and easy.

The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) is more readily available. Implementations of adjacency lists, matrices, and edge lists are included, along with a reasonable library of basic graph algorithms. Its interface and components are generic in the same sense as the C++ standard template library (STL).

JUNG (*http://jung.sourceforge.net/*) is a Java graph library particularly popular in the social networks community. The *Data Structures Library in Java* (JDSL) provides a comprehensive implementation with a decent algorithm library, and is available for noncommercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL. JGraphT (*http://jgrapht.sourceforge.net/*) is a more recent development with similar functionality.

The Stanford Graphbase (see Section 19.1.8 (page 660)) provides a simple but flexible graph data structure in CWEB, a literate version of the C language. It is instructive to see what Knuth does and does not place in his basic data structure, although we recommend other implementations as a better basis for further development.

My (biased) preference in C language graph types is the library from my book *Programming Challenges* [SR03]. See Section 19.1.10 (page 661) for details. Simple graph data structures in Mathematica are provided by *Combinatorica* [PS03], with a library of algorithms and display routines. See Section 19.1.9 (page 661).

Notes: The advantages of adjacency list data structures for graphs became apparent with the linear-time algorithms of Hopcroft and Tarjan [HT73b, Tar72]. The basic adjacency list and matrix data structures are presented in essentially all books on algorithms or data structures, including [CLRS01, AHU83, Tar83]. Hypergraphs are presented in Berge [Ber89]

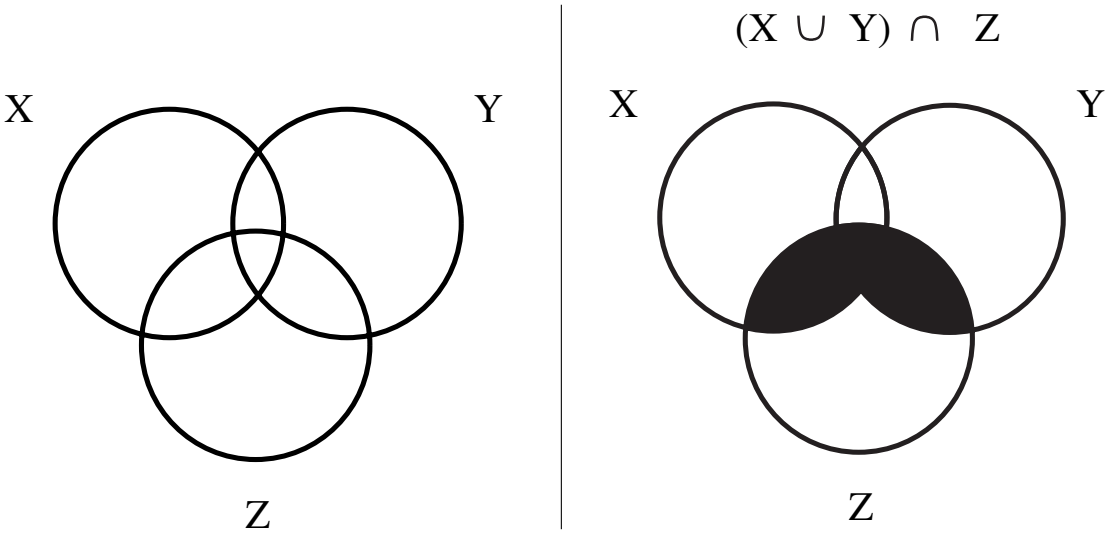
The improved efficiency of static graph types was revealed by Naher and Zlotowski [NZ02], who sped up certain LEDA graph algorithms by a factor of four by simply switching to a more compact graph structure.

An interesting question concerns minimizing the number of bits needed to represent arbitrary graphs on n vertices, particularly if certain operations must be supported efficiently. Such issues are surveyed in [vL90b].

Dynamic graph algorithms are data structures that maintain quick access to an invariant (such as minimum spanning tree or connectivity) under edge insertion and deletion. *Sparsification* [EGIN92] is a general approach to constructing dynamic graph algorithms. See [ACI92, Zar02] for experimental studies on the practicality of dynamic graph algorithms.

Hierarchically-defined graphs arise often in VLSI design problems, because designers make extensive use of cell libraries [Len90]. Algorithms specifically for hierarchically-defined graphs include planarity testing [Len89], connectivity [LW88], and minimum spanning trees [Len87a].

Related Problems: Set data structures (see page 385), graph partition (see page 541).



12.5 Set Data Structures

Input description: A universe of items $U = \{u_1, \ldots, u_n\}$ on which is defined a collection of subsets $S = \{S_1, \ldots, S_m\}$.

Problem description: Represent each subset so as to efficiently (1) test whether $u_i \in S_j$, (2) compute the union or intersection of S_i and S_j, and (3) insert or delete members of S.

Discussion: In mathematical terms, a set is an unordered collection of objects drawn from a fixed universal set. However, it is usually useful for implementation to represent each set in a single *canonical order*, typically sorted, to speed up or simplify various operations. Sorted order turns the problem of finding the union or intersection of two subsets into a linear-time operation—just sweep from left to right and see what you are missing. It makes possible element searching in sublinear time. Finally, printing the elements of a set in a canonical order paradoxically reminds us that order really doesn't matter.

We distinguish sets from two other kinds of objects: dictionaries and strings. A collection of objects *not* drawn from a fixed-size universal set is best thought of as a *dictionary*, discussed in Section 12.1 (page 367). Strings are structures where order matters—i.e. , if $\{A, B, C\}$ is not the same as $\{B, C, A\}$. Sections 12.3 and 18 discuss data structures and algorithms for strings.

Multisets permit elements to have more than one occurrence. Data structures for sets can generally be extended to multisets by maintaining a count field or linked list of equivalent entries for each element.

If each subset contains exactly two elements, they can be thought of as edges in a graph whose vertices represent the universal set. A system of subsets with no restrictions on the cardinality of its members is called a *hypergraph*. It is worth considering whether your problem has a graph-theoretical analogy, like connected components or shortest path in a graph/hypergraph.

Your primary alternatives for representing arbitrary subsets are:

- *Bit vectors* – An n-bit vector or array can represent any subset S on a universal set U containing n items. Bit i will be 1 if $i \in S$, and 0 if not. Since only one bit is needed per element, bit vectors can be very space efficient for surprisingly large values of $|U|$. Element insertion and deletion simply flips the appropriate bit. Intersection and union are done by "and-ing" or "or-ing" the bits together. The only drawback of a bit vector is its performance on sparse subsets. For example, it takes $O(n)$ time to explicitly identify all members of sparse (even empty) subset S.

- *Containers or dictionaries* – A subset can also be represented using a linked list, array, or dictionary containing exactly the elements in the subset. No notion of a fixed universal set is needed for such a data structure. For sparse subsets, dictionaries can be more space and time efficient than bit vectors, and easier to work with and program. For efficient union and intersection operations, it pays to keep the elements in each subset sorted, so a linear-time traversal through both subsets identifies all duplicates.

- *Bloom filters* – We can emulate a bit vector in the absence of a fixed universal set by hashing each subset element to an integer from 0 to n and setting the corresponding bit. Thus, bit $H(e)$ will be 1 if $e \in S$. Collisons leave some possibility for error under this scheme, however, because a different key might have hashed to the same position.

  Bloom filters use several (say k) different hash functions $H_1, \ldots H_k$, and set all k bits $H_i(e)$ upon insertion of key e. Now e is in S only if all k bits are 1. The probability of false positives can be made arbitrarily low by increasing the number of hash functions k and table size n. With the proper constants, each subset element can be represented using a constant number of bits independent of the size of the universal set.

  This hashing-based data structure is much more space-efficient than dictionaries for static subset applications that can tolerate a small probability of error. Many can. For instance, a spelling checker that left a rare random string undetected would prove no great tragedy.

Many applications involve collections of subsets that are pairwise disjoint, meaning that each element is in exactly one subset. For example, consider maintaining

the connected components of a graph or the party affiliations of politicians. Each vertex/hack is in exactly one component/party. Such a system of subsets is called a *set partition*. Algorithms for generating partitions of a given set are provided in Section 14.6 (page 456).

The primary issue with set partition data structures is maintaining changes over time, perhaps as edges are added or party members defect. Typical queries include "which set is a particular item in?" and "are two items in the same set?" as we modify the set by (1) changing one item, (2) merging or unioning two sets, or (3) breaking a set apart. Your basic options are:

- *Collection of containers* – Representing each subset in its own container/dictionary permits fast access to all the elements in the subset, which facilitates union and intersection operations. The cost comes in membership testing, as we must search each subset data structure independently until we find our target.

- *Generalized bit vector* – Let the ith element of an array denote the number/name of the subset that contains it. Set identification queries and single element modifications can be performed in constant time. However, operations like performing the union of two subsets take time proportional to the size of the universe, since each element in the two subsets must be identified and (at least one subset's worth) must have its name changed.

- *Dictionary with a subset attribute* – Similarly, each item in a binary tree can be associated a field that records the name of the subset it is in. Set identification queries and single element modifications can be performed in the time it takes to search in the dictionary. However, union/intersection operations are again slow. The need to perform such union operations quickly provides the motivation for the ...

- *Union-find data structure* – We represent a subset using a rooted tree where each node points to its *parent* instead of its children. The name of each subset will be the name of the item at the root. Finding out which subset we are in is simple, for we keep traversing up the parent pointers until we hit the root. Unioning two subsets is also easy. Just assign the root of one of two trees to point to the other, so now *all* elements have the same root and hence the same subset name.

  Implementation details have a big impact on asymptotic performance here. Always selecting the larger (or taller) tree as the root in a merger guarantees logarithmic height trees, as presented with our implementation in Section 6.1.3 (page 198). Retraversing the path traced on each find and explicitly pointing all nodes on the path to the root (called *path compression*) reduces the tree to almost constant height. Union find is a fast, simple data structure that every programmer should know about. It does not support breaking up subsets created by unions, but usually this is not an issue.

Implementations: Modern programming languages provide libraries offering complete and efficient set implementations. The C++ *Standard Template Library* (STL) provides `set` and `multiset` containers. *Java Collections* (JC) contains `HashSet` and `TreeSet` containers and is included in the `java.util` package of Java standard edition.

LEDA (see Section 19.1.1 (page 658)) provides efficient dictionary data structures, sparse arrays, and union-find data structures to maintain set partitions, all in C++.

Implementation of union-find underlies any implementation of Kruskal's minimum spanning tree algorithm. For this reason, all the graph libraries of Section 12.4 (page 381) presumably contains an implementation. Minimum spanning tree codes are described in Section 15.3 (page 484).

The computer algebra system *REDUCE* (*http://www.reduce-algebra.com/*) contains `SETS`, a package supporting set-theoretic operations on both explicit and implicit (symbolic) sets. Other computer algebra systems may support similar functionality.

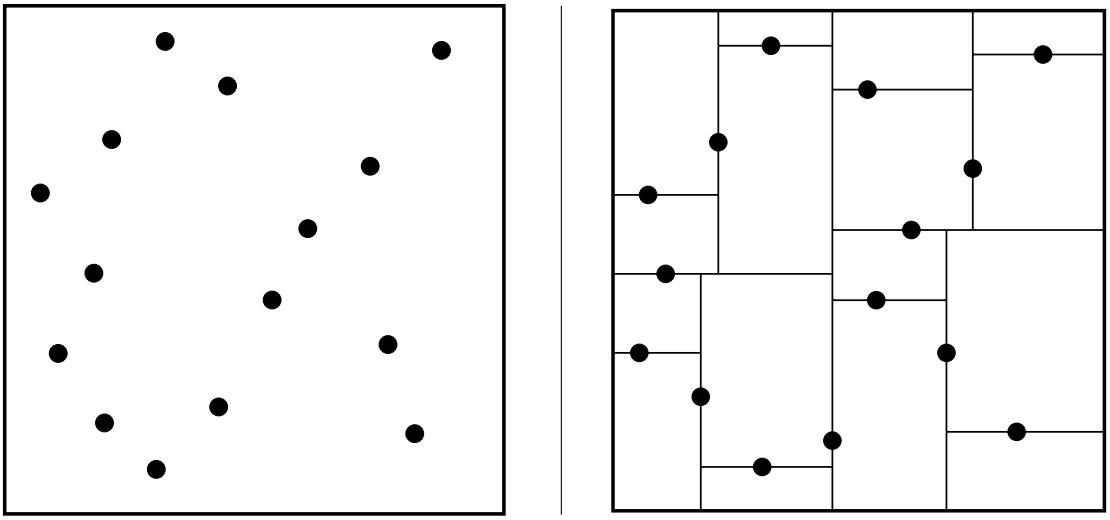
Notes: Optimal algorithms for such set operations as intersection and union were presented in [Rei72]. Raman [Ram05] provides an excellent survey on data structures for set operations on a variety of different operations. Bloom filters are ably surveyed in [BM05], with recent experimental results presented in [PSS07].

Certain balanced tree data structures support merge/meld/link/cut operations, which permit fast ways to union and intersect disjoint subsets. See Tarjan [Tar83] for a nice presentation of such structures. Jacobson [Jac89] augmented the bit-vector data structure to support select operations (where is the ith 1 bit?) efficiently in both time and space.

Galil and Italiano [GI91] survey data structures for disjoint set union. The upper bound of $O(m\alpha(m,n))$ on m union-find operations on an n-element set is due to Tarjan [Tar75], as is a matching lower bound on a restricted model of computation [Tar79]. The inverse Ackerman function $\alpha(m,n)$ grows notoriously slowly, so this performance is close to linear. An interesting connection between the worst-case of union-find and the length of Davenport-Schinzel sequences—a combinatorial structure that arises in computational geometry—is established in [SA95].

The *power set* of a set S is the collection of all $2^{|S|}$ subsets of S. Explicit manipulation of power sets quickly becomes intractable due to their size. Implicit representations of power sets in symbolic form becomes necessary for nontrivial computations. See [BCGR92] for algorithms on and computational experience with symbolic power set representations.

Related Problems: Generating subsets (see page 452), generating partitions (see page 456), set cover (see page 621), minimum spanning tree (see page 484).



INPUT OUTPUT

12.6 Kd-Trees

Input description: A set S of n points or more complicated geometric objects in k dimensions.

Problem description: Construct a tree that partitions space by half-planes such that each object is contained in its own box-shaped region.

Discussion: Kd-tree and related spatial data structures hierarchically decompose space into a small number of cells, each containing a few representatives from an input set of points. This provides a fast way to access any object by position. We traverse down the hierarchy until we find the smallest cell containing it, and then scan through the objects in this cell to identify the right one.

Typical algorithms construct kd-trees by partitioning point sets. Each node in the tree is defined by a plane cutting through one of the dimensions. Ideally, this plane equally partitions the subset of points into left/right (or up/down) subsets. These children are again partitioned into equal halves, using planes through a different dimension. Partitioning stops after $\lg n$ levels, with each point in its own leaf cell.

The cutting planes along any path from the root to another node defines a unique box-shaped region of space. Each subsequent plane cuts this box into two boxes. Each box-shaped region is defined by $2k$ planes, where k is the number of dimensions. Indeed, the "kd" in *kd-tree* is short for "k-dimensional." We maintain

the region of interest defined by the intersection of these half-spaces as we move down the tree.

Flavors of kd-trees differ in exactly how the splitting plane is selected. Options include:

- *Cycling through the dimensions* – partition first on d_1, then $d_2, \ldots, d_k$ before cycling back to d_1.
- *Cutting along the largest dimension* – select the partition dimension to make the resulting boxes as square or cube-like as possible. Selecting a plane to partition the points in half does not mean selecting a splitter in the middle of the box-shaped regions, since all the points may lie in the left side of the box.
- *Quadtrees or Octtrees* – Instead of partitioning with single planes, use all axis-parallel planes that pass through a given partition point. In two dimensions, this means creating four child cells; in 3D, this means eight child cells. Quadtrees seem particularly popular on image data, where leaf cells imply that all pixels in the regions have the same color.
- *BSP-trees – Binary space partitions* use general (i.e. , not just axis-parallel) cutting planes to carve up space into cells so that each cell ends up containing only one object (say a polygon). Such partitions are not possible using only axis-parallel cuts for certain sets of objects. The downside is that such polyhedral cell boundaries are more complicated to work with than boxes.
- *R-trees* – This is another spatial data structure useful for geometric objects that cannot be partitioned into axis-oriented boxes without cutting them into pieces. At each level, the objects are partitioned into a small number of (possibly-overlapping) boxes to construct searchable hierarchies without partitioning objects.

Ideally, our partitions split both the space (ensuring fat, regular regions) and the set of points (ensuring a log height tree) evenly, but doing both simultaneously can be impossible on a given input. The advantages of fat cells become clear in many applications of kd-trees:

- *Point location* – To identify which cell a query point q lies in, we start at the root and test which side of the partition plane contains q. By repeating this process on the appropriate child node, we travel down the tree to find the leaf cell containing q in time proportional to its height. See Section 17.7 (page 587) for more on point location.
- *Nearest neighbor search* – To find the point in S closest to a query point q, we perform point location to find the cell c containing q. Since c is bordered by some point p, we can compute the distance $d(p, q)$ from p to q. Point p is likely

close to q, but it might not be the single closest neighbor. Why? Suppose q lies right at the boundary of a cell. Then q's nearest neighbor might lie just to the left of the boundary in another cell. Thus, we must traverse all cells that lie within a distance of $d(p, q)$ of cell c and verify that none of them contain closer points. In trees with nice, fat cells, very few cells should need to be tested. See Section 17.5 (page 580) for more on nearest neighbor search.

- *Range search* – Which points lie within a query box or region? Starting from the root, check whether the query region intersects (or contains) the cell defining the current node. If it does, check the children; if not, none of the leaf cells below this node can possibly be of interest. We quickly prune away irrelevant portions of the space. Section 17.6 (page 584) focuses on range search.

- *Partial key search* – Suppose we want to find a point p in S, but we do not have full information about p. Say we are looking for someone of age 35 and height 5'8" but of unknown weight in a 3D-tree with dimensions of age, weight, and height. Starting from the root, we can identify the correct descendant for all but the weight dimension. To be sure we find the right point, we must search *both children* of these nodes. The more fields we know the better, but such partial key search can be substantially faster than checking all points against the key.

Kd-trees are most useful for a small to moderate number of dimensions, say from 2 up to maybe 20 dimensions. They lose effectiveness as the dimensionality increases, primarily because the ratio of the volume of a unit sphere in k-dimensions shrinks exponentially compared to the unit cube. Thus exponentially many cells will have to be searched within a given radius of a query point, say for nearest-neighbor search. Also, the number of neighbors for any cell grows to $2k$ and eventually become unmanageable.

The bottom line is that you should try to avoid working in high-dimensional spaces, perhaps by discarding (or projecting away) the least important dimensions.

Implementations: *KDTREE 2* contains C++ and Fortran 95 implementations of *kd*-trees for efficient nearest nearest neighbor search in many dimensions. See *http://arxiv.org/abs/physics/0408067*.

Samet's spatial index demos (*http://donar.umiacs.umd.edu/quadtree/*) provide a series of Java applets illustrating many variants of *kd*-trees, in association with his book [Sam06].

Terralib (*http://www.terralib.org/*) is an open source geographic information system (GIS) software library written in C++. This includes an implementation of spatial data structures.

The 1999 DIMACS implementation challenge focused on data structures for nearest neighbor search [GJM02]. Data sets and codes are accessible from *http://dimacs.rutgers.edu/Challenges*.

Notes: Samet [Sam06] is the best reference on kd-trees and other spatial data structures. All major (and many minor) variants are developed in substantial detail. A shorter survey [Sam05] is also available. Bentley [Ben75] is generally credited with developing kd-trees, although they have the murky history associated with most folk data structures.

The performance of spatial data structures degrades with high dimensionality. Projecting high-dimensional spaces onto a random lower-dimensional hyperplane has recently emerged as a simple but powerful method for dimensionality reduction. Both theoretical [IM04] and empirical [BM01] results indicate that this method preserves distances quite nicely.

Algorithms that quickly produce a point provably close to the query point are a recent development in higher-dimensional nearest neighbor search. A sparse weighted-graph structure is built from the data set, and the nearest neighbor is found by starting at a random point and walking greedily in the graph towards the query point. The closest point found during several random trials is declared the winner. Similar data structures hold promise for other problems in high-dimensional spaces. See [AM93, AMN$^+$98].

Related Problems: Nearest-neighbor search (see page 580), point location (see page 587), range search (see page 584).

13

Numerical Problems

If most problems you encounter are numerical in nature, there is an excellent chance that you are reading the wrong book. *Numerical Recipes* [PFTV07] gives a terrific overview to the fundamental problems in numerical computing, including linear algebra, numerical integration, statistics, and differential equations. Different flavors of the book include source code for all the algorithms in C++, Fortran, and even Pascal. Their coverage is somewhat skimpier on the combinatorial/numerical problems we consider in this section, but you should be aware of this book. Check it out at *http://www.nr.com*.

Numerical algorithms tend to be different beasts than combinatorial algorithms for at least two reasons:

- *Issues of Precision and Error* – Numerical algorithms typically perform repeated floating-point computations, which accumulate error at each operation until, eventually, the results are meaningless. My favorite example [MV99] concerns the Vancouver Stock Exchange, which over a twenty-two month period accumulated enough round-off error to reduce its index to 574.081 from the correct value of 1098.982.

  A simple and dependable way to test for round-off errors in numerical programs is to run them both at single and double precision, and then think hard whenever there is a disagreement.

- *Extensive Libraries of Codes* – Large, high-quality libraries of numerical routines have existed since the 1960s, which is still not yet the case for combinatorial algorithms. There are several reasons for this, including (1) the early emergence of Fortran as a standard for numerical computation, (2) the nature of numerical computations to be recognizably independent rather than

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_13,

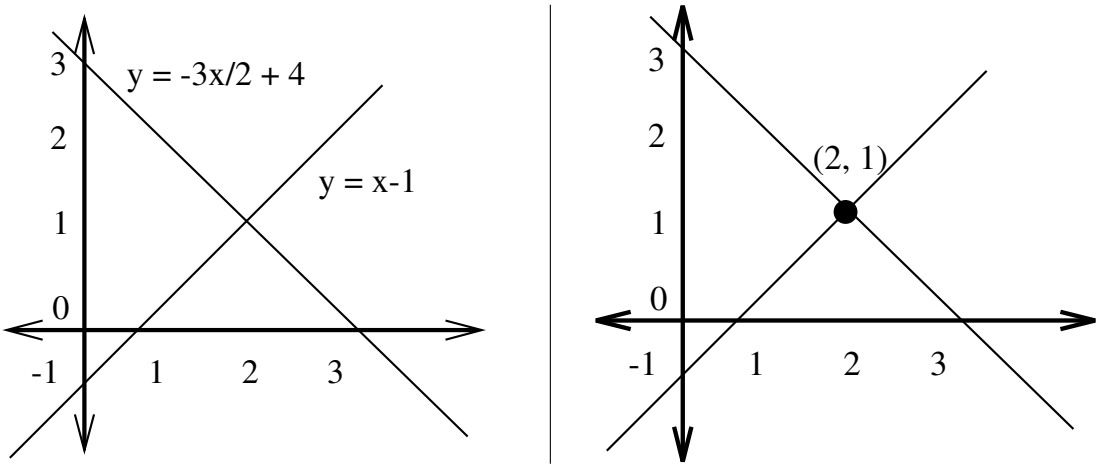
embedded within large applications, and (3) the existence of large scientific communities needing general numerical libraries.

Regardless of why, you should exploit this software base. There is probably no reason to implement algorithms for any of the problems in this section as opposed to using existing codes. Searching Netlib (see Section 19.1.5) is an excellent place to start.

Many scientist's and engineer's ideas about algorithms culturally derive from Fortran programming and numerical methods. Computer scientists grow up programming with pointers and recursion, and so are comfortable with the more sophisticated data structures required for combinatorial algorithms. Both sides can and should learn from each other, since problems such as pattern recognition can be modeled either numerically or combinatorially.

There is a vast literature on numerical algorithms. In addition to *Numerical Recipes*, recommended books include:

- *Chapara and Canale* [CC05] – The contemporary market leader in numerical analysis texts.
- *Mak* [Mak02] – This enjoyable text introduces Java to the world of numerical computation, and visa versa. Source code is provided.
- *Hamming* [Ham87] – This oldie but goodie provides a clear and lucid treatment of fundamental methods in numerical computation. It is available in a low-priced Dover edition.
- *Skeel and Keiper* [SK00] – A readable and interesting treatment of basic numerical methods, avoiding overly detailed algorithm descriptions through its use of the computer algebra system Mathematica. I like it.
- *Cheney and Kincaid* [CK07] – A traditional Fortran-based numerical analysis text, with discussions of optimization and Monte Carlo methods in addition to such standard topics as root-finding, numerical integration, linear systems, splines, and differential equations.
- *Buchanan and Turner* [BT92] – Thorough language-independent treatment of all standard topics, including parallel algorithms. It is the most comprehensive of the texts described here.



INPUT OUTPUT

13.1 Solving Linear Equations

Input description: An $m \times n$ matrix A and an $m \times 1$ vector b, together representing m linear equations on n variables.

Problem description: What is the vector x such that $A \cdot x = b$?

Discussion: The need to solve linear systems arises in an estimated 75% of all scientific computing problems [DB74]. For example, applying Kirchhoff's laws to analyze electrical circuits generates a system of equations—the solution of which puts currents through each branch of the circuit. Analysis of the forces acting on a mechanical truss generates a similar set of equations. Even finding the point of intersection between two or more lines reduces to solving a small linear system.

Not all systems of equations have solutions. Consider the equations $2x + 3y = 5$ and $2x + 3y = 6$. Some systems of equations have multiple solutions, such as $2x + 3y = 5$ and $4x + 6y = 10$. Such *degenerate* systems of equations are called *singular*, and they can be recognized by testing whether the determinant of the coefficient matrix is zero.

Solving linear systems is a problem of such scientific and commercial importance that excellent codes are readily available. There is no good reason to implement your own solver, even though the basic algorithm (Gaussian elimination) is one you learned in high school. This is especially true when working with large systems.

Gaussian elimination is based on the observation that the solution to a system of linear equations is invariant under scaling (if $x = y$, then $2x = 2y$) and adding

equations (the solution to $x = y$ and $w = z$ is the same as the solution to $x = y$ and $x + w = y + z$). Gaussian elimination scales and adds equations to eliminate each variable from all but one equation, leaving the system in such a state that the solution can be read off from the equations.

The time complexity of Gaussian elimination on an $n \times n$ system of equations is $O(n^3)$, since to clear the ith variable we add a scaled copy of the n-term ith row to each of the $n-1$ other equations. On this problem, however, constants matter. Algorithms that only partially reduce the coefficient matrix and then backsubstitute to get the answer use 50% fewer floating-point operations than the naive algorithm.

Issues to worry about include:

- *Are roundoff errors and numerical stability affecting my solution?* – Gaussian elimination would be quite straightforward to implement except for roundoff errors. These accumulate with each row operation and can quickly wreak havoc on the solution, particularly with matrices that are *almost* singular.

  To eliminate the danger of numerical errors, it pays to substitute the solution back into each of the original equations and test how close they are to the desired value. *Iterative methods* for solving linear systems refine initial solutions to obtain more accurate answers. Good linear systems packages will include such routines.

  The key to minimizing roundoff errors in Gaussian elimination is selecting the right equations and variables to pivot on, and to scale the equations to eliminate large coefficients. This is an art as much as a science, which is why you should use a well-crafted library routine as described next.

- *Which routine in the library should I use?* – Selecting the right code is also somewhat of an art. If you are taking your advice from this book, start with the general linear system solvers. Hopefully they will suffice for your needs. But search through the manual for more efficient procedures solving special types of linear systems. If your matrix happens to be one of these special types, the solution time can reduce from cubic to quadratic or even linear.

- *Is my system sparse?* – The key to recognizing that you have a special-case linear system is establishing how many matrix elements you really need to describe A. If there are only a few non-zero elements, your matrix is *sparse* and you are in luck. If these few non-zero elements are clustered near the diagonal, your matrix is *banded* and you are in even more luck. Algorithms for reducing the bandwidth of a matrix are discussed in Section 13.2. Many other regular patterns of sparse matrices can also be exploited, so consult the manual of your solver or a better book on numerical analysis for details.

- *Will I be solving many systems using the same coefficient matrix?* – In applications such as least-squares curve fitting, we must solve $A \cdot x = b$ repeatedly with different b vectors. We can preprocess A to make this easier. The lower-upper or *LU-decomposition* of A creates lower- and upper-triangular matrices

L and U such that $L \cdot U = A$. We can use this decomposition to solve $A \cdot x = b$, since

$$A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b$$

This is efficient since backsubstitution solves a triangular system of equations in quadratic time. Solving $L \cdot y = b$ and then $U \cdot x = y$ gives the solution x using two $O(n^2)$ steps instead of one $O(n^3)$ step, after the LU-decomposition has been found in $O(n^3)$ time.

The problem of solving linear systems is equivalent to that of matrix inversion, since $Ax = B \leftrightarrow A^{-1}Ax = A^{-1}B$, where $I = A^{-1}A$ is the identity matrix. Avoid it however since matrix inversion proves to be three times slower than Gaussian elimination. LU-decomposition proves useful in inverting matrices as well as computing determinants (see Section 13.4 (page 404)).

Implementations: The library of choice for solving linear systems is apparently LAPACK—a descendant of LINPACK [DMBS79]. Both of these Fortran codes, as well as many others, are available from Netlib. See Section 19.1.5 (page 659).

Variants of LAPACK exist for other languages, like CLAPACK (C) and LAPACK++ (C++). The *Template Numerical Toolkit* is an interface to such routines in C++, and is available at *http://math.nist.gov/tnt/*.

JScience provides an extensive linear algebra package (including determinants) as part of its comprehensive scientific computing library. *JAMA* is another matrix package written in Java. Links to both and many related libraries are available at *http://math.nist.gov/javanumerics/*.

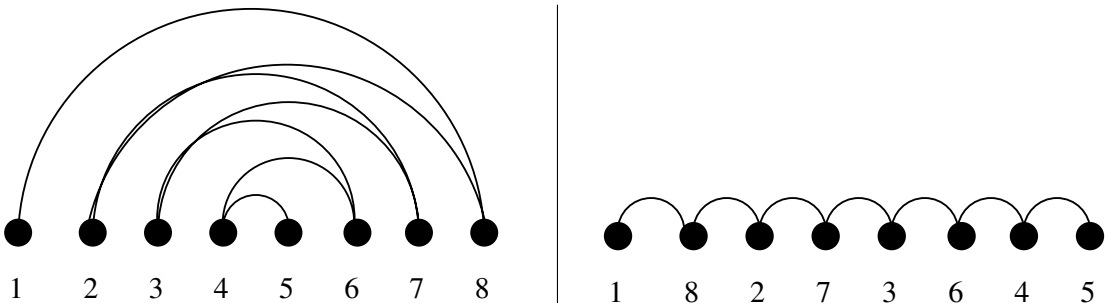
Numerical Recipes [PFTV07] (*www.nr.com*) provides guidance and routines for solving linear systems. Lack of confidence in dealing with numerical procedures is the most compelling reason to use these ahead of the free codes.

Notes: Golub and van Loan [GL96] is the standard reference on algorithms for linear systems. Good expositions on algorithms for Gaussian elimination and LU-decomposition include [CLRS01] and a host of numerical analysis texts [BT92, CK07, SK00]. Data structures for linear systems are surveyed in [PT05].

Parallel algorithms for linear systems are discussed in [Gal90, KSV97, Ort88]. Solving linear systems is one of most important applications where parallel architectures are used widely in practice.

Matrix inversion and (hence) linear systems solving can be done in matrix multiplication time using Strassen's algorithm plus a reduction. Good expositions on the equivalence of these problems include [AHU74, CLRS01].

Related Problems: Matrix multiplication (see page 401), determinant/permanent (see page 404).



13.2 Bandwidth Reduction

Input description: A graph $G = (V, E)$, representing an $n \times n$ matrix M of zero and non-zero elements.

Problem description: Which permutation p of the vertices minimizes the length of the longest edge when the vertices are ordered on a line—i.e. , minimizes $\max_{(i,j)\in E} |p(i) - p(j)|$?

Discussion: Bandwidth reduction lurks as a hidden but important problem for both graphs and matrices. Applied to matrices, bandwidth reduction permutes the rows and columns of a sparse matrix to minimize the distance b of any non-zero entry from the center diagonal. This is important in solving linear systems, because Gaussian elimination (see Section 13.1 (page 395)) can be performed in $O(nb^2)$ on matrices of bandwidth b. This is a big win over the general $O(n^3)$ algorithm if $b << n$.

Bandwidth minimization on graphs arises in more subtle ways. Arranging n circuit components in a line to minimize the length of the longest wire (and hence time delay) is a bandwidth problem, where each vertex of our graph corresponds to a circuit component and there is an edge for every wire linking two components. Alternatively, consider a hypertext application where we must store large objects (say images) on a magnetic tape. Each image has a set of possible images to visit next (i.e. , the hyperlinks). We seek to place linked images near each other on the tape to minimize the search time. This is exactly the bandwidth problem. More general formulations, such as rectangular circuit layouts and magnetic disks, inherit the same hardness and classes of heuristics from the linear versions.

The *bandwidth* problem seeks a linear ordering of the vertices, which minimizes the length of the longest edge, but there are several variations of the problem. In *linear arrangement*, we seek to minimize the sum of the lengths of the edges. This

has application to circuit layout, where we seek to position the chips to minimize the total wire length. In *profile minimization*, we seek to minimize the sum of one-way distances (i.e., for each vertex v) the length of the longest edge whose other vertex is to the left of of v.

Unfortunately, bandwidth minimization and all these variants is NP-complete. It stays NP-complete even if the input graph is a tree whose maximum vertex degree is 3, which is about as strong a condition as I have seen on any problem. Thus our only options are a brute-force search and heuristics.

Fortunately, ad hoc heuristics have been well studied and production-quality implementations of the best heuristics are available. These are based on performing a breadth-first search from a given vertex v, where v is placed at the leftmost point of the ordering. All of the vertices that are distance 1 from v are placed to its immediate right, followed by all the vertices at distance 2, and so forth until all vertices in G are accounted for. The popular heuristics differ according to how many different start vertices are considered and how equidistant vertices are ordered among themselves. Breaking ties with low-degree vertices over to the left however seems to be a good idea.

Implementations of the most popular heuristics—the Cuthill-McKee and Gibbs-Poole-Stockmeyer algorithms—are discussed in the implementation section. The worst case of the Gibbs-Poole-Stockmeyer algorithm is $O(n^3)$, which would wash out any possible savings in solving linear systems, but its performance in practice is close to linear.

Brute-force search programs can find the exact minimum bandwidth by backtracking through the set of $n!$ possible permutations of vertices. Considerable pruning can be achieved to reduce the search space by starting with a good heuristic bandwidth solution and alternately adding vertices to the left- and rightmost open slots in the partial permutation.

Implementations: Del Corso and Manzini's [CM99] code for exact solutions to bandwidth problems is available at *http://www.mfn.unipmn.it/∼manzini/bandmin*. Caprara and Salazar-González [CSG05] developed improved methods based on integer programming. Their branch-and-bound implementation in C is available at *http://joc.pubs.informs.org/Supplements/Caprara-2/*.

Fortran language implementations of both the Cuthill-McKee algorithm [CGPS76, Gib76, CM69] and the Gibbs-Poole-Stockmeyer algorithm [Lew82, GPS76] are available from Netlib. See Section 19.1.5 (page 659). Empirical evaluations of these and other algorithms on a test suite of 30 matrices are discussed in [Eve79b], showing Gibbs-Poole-Stockmeyer to be the consistent winner.

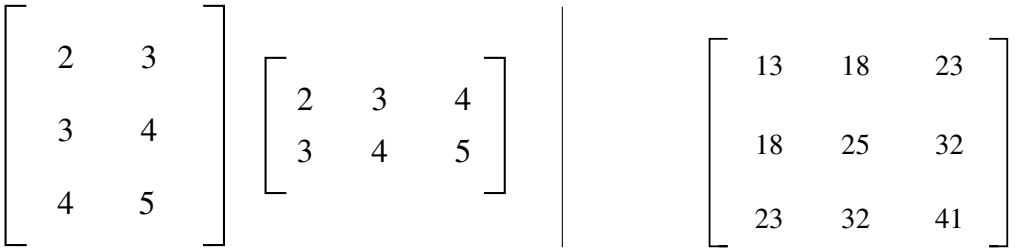
Petit [Pet03] performed an extensive experimental study on heuristics for the minimum linear arrangement problem. His codes and data are available at *http://www.lsi.upc.edu/∼jpetit/MinLA/Experiments/*.

Notes: Diaz et al. [DPS02] provide an excellent up-to-date survey on algorithms for bandwidth and related graph layout problems. See [CCDG82] for graph-theoretic and algorithmic results on bandwidth up to 1981.

Ad hoc heuristics have been widely studied—a tribute to its importance in numerical computation. Everstine [Eve79b] cites no less than 49 different bandwidth reduction algorithms! Del Corso and Romani [CR01] investigate a new class of spectral heuristics for bandwidth minimization.

The hardness of the bandwidth problem was first established by Papadimitriou [Pap76b], and its hardness on trees of maximum degree 3 in [GGJK78]. There are algorithms that run in polynomial time for fixed bandwidth k [Sax80]. Approximation algorithms offering a polylogarithmic guarantee exist for the general problem [BKRV00].

Related Problems: Solving linear equations (see page 395), topological sorting (see page 481).



13.3 Matrix Multiplication

Input description: An $x \times y$ matrix A and a $y \times z$ matrix B.

Problem description: Compute the $x \times z$ matrix $A \times B$.

Discussion: Matrix multiplication is a fundamental problem in linear algebra. Its main significance for combinatorial algorithms is its equivalence to many other problems, including transitive closure/reduction, parsing, solving linear systems, and matrix inversion. Thus, a faster algorithm for matrix multiplication implies faster algorithms for all of these problems. Matrix multiplication arises in its own right in computing the results of such coordinate transformations as scaling, rotation, and translation for robotics and computer graphics.

The following tight algorithm computes the product of $x \times y$ matrix A and $y \times z$ matrix B runs in $O(xyz)$. Remember to first initialize $M[i,j]$ to 0 for all $1 \leq i \leq x$ and $i \leq j \leq z$:

```
for i = 1 to x do
    for j = 1 to z
        for k = 1 to y
            M[i,j] = M[i,j] + A[i,k] · A[k,j]
```

An implementation in C appears in Section 2.5.4 (page 45). This straightforward algorithm would *seem* to be tough to beat in practice. That said, observe that the three loops can be arbitrarily permuted without changing the resulting answer. Such a permutation will change the memory access patterns and thus how effectively the cache memory is used. One can expect a 10-20% variation in run time among the six possible implementations, but could not confidently predict the winner (typically ikj) without running it on your machine with your given matrices.

When multiplying bandwidth-b matrices, where all non-zero elements of A and B lie within b elements of the main diagonals, a speedup to $O(xbz)$ is possible, since zero elements cannot contribute to the product.

Asymptotically faster algorithms for matrix multiplication exist, using clever divide-and-conquer recurrences. However, these prove difficult to program, require very large matrices to beat the trivial algorithm, and are less numerically stable to boot. The most famous of these is Strassen's $O(n^{2.81})$ algorithm. Empirical results (discussed next) disagree on the exact crossover point where Strassen's algorithm beats the simple cubic algorithm, but it is in the ballpark of $n \approx 100$.

There is a better way to save computation when you are multiplying a chain of more than two matrices together. Recall that multiplying an $x \times y$ matrix by a $y \times z$ matrix creates an $x \times z$ matrix. Thus, multiplying a chain of matrices from left to right might create large intermediate matrices, each taking a lot of time to compute. Matrix multiplication is not commutative, but it is associative, so we can parenthesize the chain in whatever manner we deem best without changing the final product. A standard dynamic programming algorithm can be used to construct the optimal parenthesization. Whether it pays to do this optimization will depend upon whether your matrix dimensions are sufficiently irregular and your chain multiplied often enough to justify it. Note that we are optimizing over the sizes of the dimensions in the chain, not the actual matrices themselves. No such optimization is possible if all your matrices are the same dimensions.

Matrix multiplication has a particularly interesting interpretation in counting the number of paths between two vertices in a graph. Let A be the adjacency matrix of a graph G, meaning $A[i,j] = 1$ if there is an edge between i and j. Otherwise, $A[i,j] = 0$. Now consider the square of this matrix, $A^2 = A \times A$. If $A^2[i,j] \geq 1$. This means that there must be a vertex k such that $A[i,k] = A[k,j] = 1$, so i to k to j is a path of length 2 in G. More generally, $A^k[i,j]$ counts the number of paths of length exactly k between i and j. This count includes nonsimple paths, where vertices are repeated, such as i to k to i to j.

Implementations: D'Alberto and Nicolau [DN07] have engineered a very efficient matrix multiplication code, which switches from Strassen's to the cubic algorithm at the optimal point. It is available at *http://www.ics.uci.edu/~fastmm/*. Earlier experiments put the crossover point where Strassen's algorithm beats the cubic algorithm at about $n = 128$ [BLS91, CR76].

Thus, an $O(n^3)$ algorithm will likely be your best bet unless your matrices are very large. The linear algebra library of choice is LAPACK, a descendant of LINPACK [DMBS79], which includes several routines for matrix multiplication. These Fortran codes are available from Netlib as discussed in Section 19.1.5 (page 659).

Algorithm 601 [McN83] of the Collected Algorithms of the ACM is a sparse matrix package written in Fortran that includes routines to multiply any combination of sparse and dense matrices. See Section 19.1.5 (page 659) for details.

Notes: Winograd's algorithm for fast matrix multiplication reduces the number of multiplications by a factor of two over the straightforward algorithm. It is implementable, although the additional bookkeeping required makes it doubtful whether it is a win. Expositions on Winograd's algorithm [Win68] include [CLRS01, Man89, Win80].

In my opinion, the history of theoretical algorithm design began when Strassen [Str69] published his $O(n^{2.81})$-time matrix multiplication algorithm. For the first time, improving an algorithm in the asymptotic sense became a respected goal in its own right. Progressive improvements to Strassen's algorithm have gotten progressively less practical. The current best result for matrix multiplication is Coppersmith and Winograd's [CW90] $O(n^{2.376})$ algorithm, while the conjecture is that $\Theta(n^2)$ suffices. See [CKSU05] for an alternate approach that recently yielded an $O(n^{2.41})$ algorithm.

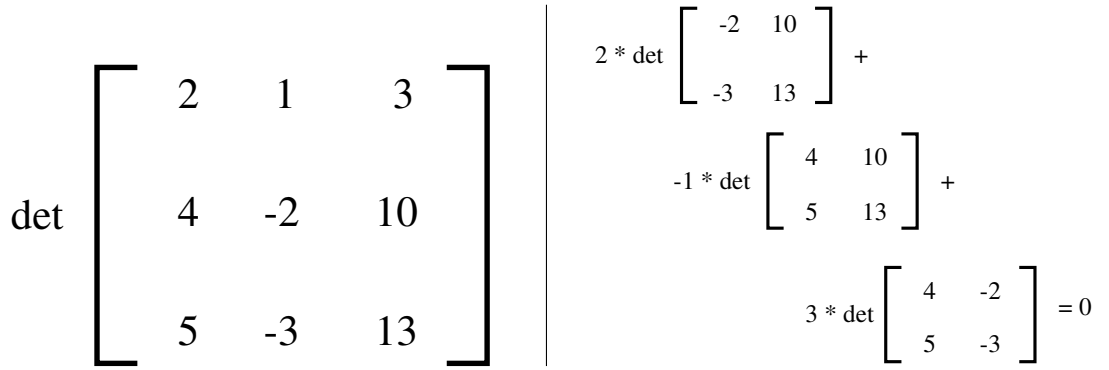
Engineering efficient matrix multiplication algorithms requires careful management of cache memory. See [BDN01, HUW02] for studies on these issues.

The interest in the squares of graphs goes beyond counting paths. Fleischner [Fle74] proved that the square of any biconnected graph has a Hamiltonian cycle. See [LS95] for results on finding the square roots of graphs—i.e. , finding A given A^2.

The problem of Boolean matrix multiplication can be reduced to that of general matrix multiplication [CLRS01]. The four-Russians algorithm for Boolean matrix multiplication [ADKF70] uses preprocessing to construct all subsets of $\lg n$ rows for fast retrieval in performing the actual multiplication, yielding a complexity of $O(n^3/\lg n)$. Additional preprocessing can improve this to $O(n^3/\lg^2 n)$ [Ryt85]. An exposition on the four-Russians algorithm, including this speedup, appears in [Man89].

Good expositions of the matrix-chain algorithm include [BvG99, CLRS01], where it is given as a standard textbook example of dynamic programming.

Related Problems: Solving linear equations (see page 395), shortest path (see page 489).



INPUT OUTPUT

13.4 Determinants and Permanents

Input description: An $n \times n$ matrix M.

Problem description: What is the determinant $|M|$ or permanent $perm(M)$ of the matrix m?

Discussion: Determinants of matrices provide a clean and useful abstraction in linear algebra that can be used to solve a variety of problems:

- Testing whether a matrix is *singular*, meaning that the matrix does not have an inverse. A matrix M is singular iff $|M| = 0$.
- Testing whether a set of d points lies on a plane in fewer than d dimensions. If so, the system of equations they define is singular, so $|M| = 0$.
- Testing whether a point lies to the left or right of a line or plane. This problem reduces to testing whether the sign of a determinant is positive or negative, as discussed in Section 17.1 (page 564).
- Computing the area or volume of a triangle, tetrahedron, or other simplicial complex. These quantities are a function of the magnitude of the determinant, as discussed in Section 17.1 (page 564).

The determinant of a matrix M is defined as a sum over all $n!$ possible permutations π_i of the n columns of M:

$$|M| = \sum_{i=1}^{n!} (-1)^{sign(\pi_i)} \prod_{j=1}^{n} M[j, \pi_j]$$

where $sign(\pi_i)$ denotes the number of pairs of elements out of order (called *inversions*) in permutation π_i.

A direct implementation of this definition yields an $O(n!)$ algorithm, as does the cofactor expansion method I learned in high school. Better algorithms to evaluate determinants are based on LU-decomposition, discussed in Section 13.1 (page 395). The determinant of M is simply the product of the diagonal elements of the LU-decomposition of M, which can be found in $O(n^3)$ time.

A closely related function called the *permanent* arises in combinatorial problems. For example, the permanent of the adjacency matrix of a graph G counts the number of perfect matchings in G. The permanent of a matrix M is defined by

$$perm(M) = \sum_{i=1}^{n!} \prod_{j=1}^{n} M[j, \pi_j]$$

differing from the determinant only in that all products are positive.

Surprisingly, it is NP-hard to compute the permanent, even though the determinant can easily be computed in $O(n^3)$ time. The fundamental difference is that $det(AB) = det(A) \times det(B)$, while $perm(AB) \neq perm(A) \times perm(B)$. There are permanent algorithms running in $O(n^2 2^n)$ time that prove to be considerably faster than the $O(n!)$ definition. Thus, finding the permanent of a 20×20 matrix is not out of the realm of possibility.

Implementations: The linear algebra package LINPACK contains a variety of Fortran routines for computing determinants, optimized for different data types and matrix structures. It can be obtained from Netlib, as discussed in Section 19.1.5 (page 659).

JScience provides an extensive linear algebra package (including determinants) as part of its comprehensive scientific computing library. *JAMA* is another matrix package written in Java. Links to both and many related libraries are available from *http://math.nist.gov/javanumerics/*.

Nijenhuis and Wilf [NW78] provide an efficient Fortran routine to compute the permanent of a matrix. See Section 19.1.10 (page 661). Cash [Cas95] provides a C routine to compute the permanent, motivated by the Kekulé structure count of computational chemistry.

Two different codes for approximating the permanent are provided by Barvinok. The first, based on [BS07], provides codes for approximating the permanent and a Hafnian of a matrix, as well as the number of spanning forests in a graph. See *http://www.math.lsa.umich.edu/∼barvinok/manual.html*. The second, based on [SB01], can provide estimates of the permanent of 200×200 matrices in seconds. See *http://www.math.lsa.umich.edu/∼barvinok/code.html*.

Notes: Cramer's rule reduces the problems of matrix inversion and solving linear systems to that of computing determinants. However, algorithms based on LU-determination are faster. See [BM53] for an exposition on Cramer's rule.

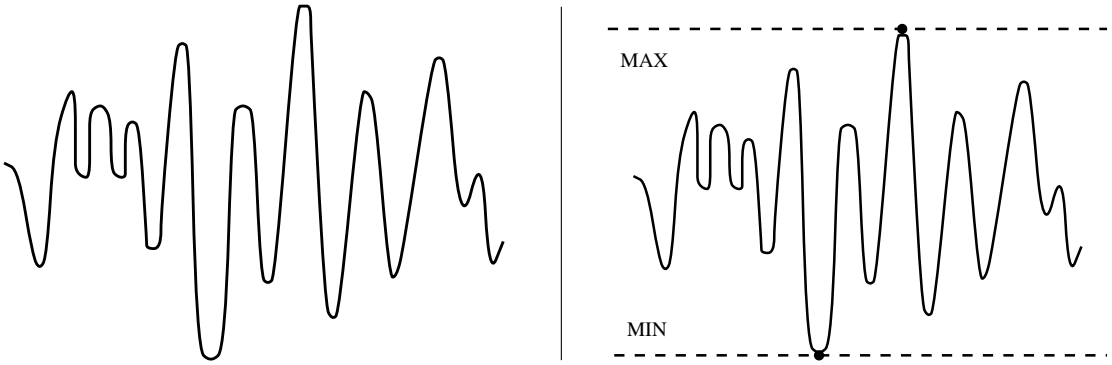
Determinants can be computed in $o(n^3)$ time using fast matrix multiplication, as shown in [AHU83]. Section 13.3 (page 401) discusses such algorithms. A fast algorithm for computing the sign of the determinant—an important problem for performing robust geometric computations—is due to Clarkson [Cla92].

The problem of computing the permanent was shown to be #P-complete by Valiant [Val79], where #P is the class of problems solvable on a "counting" machine in polynomial time. A counting machine returns the number of distinct solutions to a problem. Counting the number of Hamiltonian cycles in a graph is a #P-complete problem that is trivially NP-hard (and presumably harder), since any count greater than zero proves that the graph is Hamiltonian. Counting problems can be #P-complete even if the corresponding decision problem can be solved in polynomial time, as shown by the permanent and perfect matchings.

Minc [Min78] is the primary reference on permanents. A variant of an $O(n^2 2^n)$-time algorithm due to Ryser for computing the permanent is presented in [NW78].

Recently, probabilistic algorithms have been developed for estimating the permanent, culminating in a fully-polynomial randomized approximation scheme that provides an arbitrary close approximation in time that depends polynomially upon the input matrix and the desired error [JSV01].

Related Problems: Solving linear systems (see page 395), matching (see page 498), geometric primitives (see page 564).



13.5 Constrained and Unconstrained Optimization

Input description: A function $f(x_1, \ldots, x_n)$.

Problem description: What point $p = (p_1, \ldots, p_n)$ maximizes (or minimizes) the function f?

Discussion: Most of this book concerns algorithms that optimize one thing or another. This section considers the general problem of optimizing functions where, due to lack of structure or knowledge, we are unable to exploit the problem-specific algorithms seen elsewhere in this book.

Optimization arises whenever there is an objective function that must be tuned for optimal performance. Suppose we are building a program to identify good stocks to invest in. We have certain financial data available to analyze—such as the price-earnings ratio, the interest rate, and the stock price—all as a function of time t. The key question is how much weight we should give to each of these factors, where these weights correspond to coefficients of a formula:

$$\text{stock-goodness}(t) = c_1 \times \text{price}(t) + c_2 \times \text{interest}(t) + c_3 \times \text{PE-ratio}(t)$$

We seek the numerical values c_1, c_2, c_3 whose stock-goodness function does the best job of evaluating stocks. Similar issues arise in tuning evaluation functions for any pattern recognition task.

Unconstrained optimization problems also arise in scientific computation. Physical systems from protein structures to galaxies naturally seek to minimize their "energy" or "potential function." Programs that attempt to simulate nature thus often define potential functions assigning a score to each possible object configuration, and then select the configuration that minimizes this potential.

Global optimization problems tend to be hard, and there are lots of ways to go about them. Ask the following questions to steer yourself in the right direction:

- *Am I doing constrained or unconstrained optimization?* – In unconstrained optimization, there are no limitations on the values of the parameters other than that they maximize the value of f. However, many applications demand constraints on these parameters that make certain points illegal, points that might otherwise be the global optimum. For example, companies cannot employ less than zero employees, no matter how much money they think they might save doing so. Constrained optimization problems typically require mathematical programming approaches, like linear programming, discussed in Section 13.6 (page 411).

- *Is the function I am trying to optimize described by a formula?* – Sometimes the function that you seek to optimize is presented as an algebraic formula, such as finding the minimum of $f(n) = n^2 - 6n + 2^{n+1}$. If so, the solution is to analytically take its derivative $f'(n)$ and see for which points p' we have $f'(p') = 0$. These points are either local maxima or minima, which can be distinguished by taking a second derivative or just plugging p' back into f and seeing what happens. Symbolic computation systems such as Mathematica and Maple are fairly effective at computing such derivatives, although using computer algebra systems effectively is somewhat of a black art. They are definitely worth a try, however, and you can always use them to plot a picture of your function to get a better idea of what you are dealing with.

- *Is it expensive to compute the function at a given point?* – Instead of a formula, we are often given a program or subroutine that evaluates f at a given point. Since we can request the value of any given point on demand by calling this function, we can poke around and try to guess the maxima.

  Our freedom to search in such a situation depends upon how efficiently we can evaluate f. Suppose that $f\ (x_1, \ldots, x_n)$ is the board evaluation function in a computer chess program, such that x_1 is how much a pawn is worth, x_2 is how much a bishop is worth, and so forth. To evaluate a set of coefficients as a board evaluator, we must play a bunch of games with it or test it on a library of known positions. Clearly, this is time-consuming, so we would have to be frugal in the number of evaluations of f we use to optimize the coefficients.

- *How many dimensions do we have? How many do we need?* – The difficulty in finding a global maximum increases rapidly with the number of dimensions (or parameters). For this reason, it often pays to reduce the dimension by ignoring some of the parameters. This runs counter to intuition, for the naive programmer is likely to incorporate as many variables as possible into their evaluation function. It is just too hard, however, to optimize such complicated

functions. Much better is to start with the three to five most important variables and do a good job optimizing the coefficients for these.

- *How smooth is my function?* The main difficulty of global optimization is getting trapped in local optima. Consider the problem of finding the highest point in a mountain range. If there is only one mountain and it is nicely shaped, we can find the top by just walking in whatever direction is up. However, if there are many false summits, or other mountains in the area, it is difficult to convince ourselves whether we really are at the highest point. *Smoothness* is the property that enables us to quickly find the local optimum from a given point. We assume smoothness in seeking the peak of the mountain by walking up. If the height at any given point was a completely random function, there would be no way we could find the optimum height short of sampling every single point.

The most efficient algorithms for unconstrained global optimization use derivatives and partial derivatives to find local optima, to point out the direction in which moving from the current point does the most to increase or decrease the function. Such derivatives can sometimes be computed analytically, or they can be estimated numerically by taking the difference between the values of nearby points. A variety of *steepest descent* and *conjugate gradient* methods to find local optima have been developed—similar in many ways to numerical root-finding algorithms.

It is a good idea to try several different methods on any given optimization problem. For this reason, we recommend experimenting with the implementations below before attempting to implement your own method. Clear descriptions of these algorithms are provided in several numerical algorithms books, in particular *Numerical Recipes* [PFTV07].

For constrained optimization, finding a point that satisfies all the constraints is often the difficult part of the problem. One approach is to use a method for unconstrained optimization, but add a penalty according to how many constraints are violated. Determining the right penalty function is problem-specific, but it often makes sense to vary the penalties as optimization proceeds. At the end, the penalties should be very high to ensure that all constraints are satisfied.

Simulated annealing is a fairly robust and simple approach to constrained optimization, particularly when we are optimizing over combinatorial structures (permutations, graphs, subsets). Techniques for simulated annealing are described in Section 7.5.3 (page 254).

Implementations: The world of constrained/unconstrained optimization is sufficiently confusing that several guides have been created to point people to the right codes. Particularly nice is Hans Mittlemann's *Decision Tree for Optimization Software* at *http://plato.asu.edu/guide.html*. Also check out the selection at GAMS, the NIST *Guide to Available Mathematical Software*, at http://gams.nist.gov.

NEOS (Network-Enabled Optimization System) provides a unique service—the opportunity to solve your problem remotely on computers and software at Argonne

National Laboratory. Linear programming and unconstrained optimization are both supported. Check out *http://www-neos.mcs.anl.gov/* when you need a solution instead of a program.

Several of the *Collected Algorithms of the ACM* are Fortran codes for unconstrained optimization, most notably Algorithm 566 [MGH81] , Algorithm 702 [SF92] , and Algorithm 734 [Buc94] . Algorithm 744 [Rab95] does unconstrained optimization in Lisp. They are all available from Netlib (see Section 19.1.5 (page 659)).

General purpose simulated annealing implementations are available, and probably are the best place to start experimenting with this technique for constrained optimization. Feel free to try my code from Section 7.5.3 (page 254). Particularly popular is *Adaptive Simulated Annealing (ASA)*, written in C by Lester Ingber and available at *http://asa-caltech.sourceforge.net/*.

Both the Java Genetic Algorithms Package (JGAP) (*http://jgap.sourceforge.net/*) and the C Genetic Algorithm Utility Library (GAUL) (*http://gaul.sourceforge.net/*) are designed to aid in the development of applications that use genetic/evolutionary algorithms. I am highly skeptical about genetic algorithms (see Section 7.8 (page 266)), but other people seem to find them irresistible.

Notes: Steepest-descent methods for unconstrained optimization are discussed in most books on numerical methods, including [BT92, PFTV07]. Unconstrained optimization is the topic of several books, including [Bre73, Fle80].

Simulated annealing was devised by Kirkpatrick et al. [KGV83] as a modern variation of the Metropolis algorithm [MRRT53]. Both use Monte Carlo techniques to compute the minimum energy state of a system. Good expositions on all local search variations, including simulated annealing, appear in [AL97].

Genetic algorithms were developed and popularized by Holland [Hol75, Hol92]. More sympathetic expositions on genetic algorithms include [LP02, MF00]. Tabu search [Glo90] is yet another heuristic search procedure with a devoted following.

Related Problems: Linear programming (see page 411), satisfiability (see page 472).