A Practical Guide to Discrete Optimization

Chapter 1, Draft of 7 August 2014

David L. Applegate
William J. Cook
Sanjeeb Dash
David S. Johnson

The final test of a theory is its capacity to solve the problems which originated it.

George B. Dantzig, 1963.

Preface

A beautiful aspect of discrete optimization is the deep mathematical theory that complements a wide range of important applications. It is the mix of theory and practice that drives the most important advances in the field. There is, however, no denying the adage that the theory-to-practice road can be both long and difficult. Indeed, understanding an idea in a textbook or on the blackboard is often but the first step towards the creation of fast and accurate solution methods suitable for practical computation. In this book we aim to present a guide for students, researchers, and practitioners who must take the remaining steps. The road may be difficult, but the adoption of fundamental data structures and algorithms, together with carefully-planned computational techniques and good computer-coding practices, can shorten the journey. We hope the reader will find, as we do, elegance and depth in the engineering process of transforming theory into tools for attacking optimization problems of great complexity.

Chapter One

The Setting

I don't think any of my theoretical results have provided as great a thrill as the sight of the numbers pouring out of the computer on the night Held and I first tested our bounding method.

Richard Karp, 1985.

Discrete optimization is the study of problems that involve the selection of the best alternative from a field of possibilities. The *shortest-path problem* asks for the quickest way to travel from one point to another along a network of roads, the *traveling salesman problem* asks for the shortest way to visit a collection of cities, *optimal matching problems* ask for the best way to pair-up a set of objects, and so on. Discrete-optimization models, such as these, are typically defined on discrete structures, including networks, graphs, and matrices.

As a field of mathematics, discrete optimization is both broad and deep, and excellent reference books are available. The main lines of research described in this mathematics literature concern structural theory and the basic solvability of certain classes of models. Discrete optimization is also studied in theoretical computer science, where research focuses on solution algorithms that are provably efficient as problem sizes increase to infinity. In both cases, the motivating interest stems from the computational side of the subject, namely, the great number of problems of practical and theoretical importance that come down to making an optimal selection. This computational side has a natural home in the applied field of operations research, where discrete optimization is an important tool in the solution of models arising in industry and commerce.

NEEDLE IN THE HAYSTACK

At first glance, discrete optimization, as an applied subject, seems particularly simple: if you have a field of possibilities, then examine each alternative and choose the best. The difficulty comes down to numbers. Paths, traveling salesman tours, matchings, and other basic objects are all far too numerous to consider checking solutions one by one.

This numbers argument is easy to see in the case of the traveling salesman problem, or *TSP* for short. Suppose we need to visit n cities in a tour that starts and ends at the same city. The length of such a circular tour does not depend on where we begin, so we may fix any city as the starting point. We then have $n - 1$ choices for the second city, $n-2$ choices for the third city, and so on. The total number of tours is thus

$$(n-1)! = (n-1) \times (n-2) \times (n-3) \times \cdots \times 2 \times 1.$$

For ten cities this is only 362,800 possibilities and it would be easy enough to examine them all with the help of a computer. Even twenty cities looks solvable by running through all 1.2×10^{17} candidates, if you have a very fast machine and sufficient patience. But going up to thirty cities gives 8.8×10^{30} tours, and such a number is well out of the reach of any available computing platforms. In interpreting such a statement, keep in mind that although computers are fast, they do have limitations. The world's top-ranked supercomputer in 2011 has a peak performance of 34×10^{15} operations per second. So even if it could be arranged so that checking each new tour requires a single operation only, we would still need over 700,000 years for the full computation.

Now please do not read too much into this numbers game. The 700,000-year computation does not imply that TSPs with thirty cities are unsolvable. Definitely not. Instances of the salesman problem with hundreds or even thousands of cities are solved routinely with current methodology. All the numbers game shows is that simple enumeration methods cannot succeed. The haystack is far too large to find the needle by brute force, even with the help of the fastest computer in the world.

IS THE SALESMAN TOUGH?

Among discrete optimization problems, the TSP has a notorious reputation. At a conference several years ago, Sylvia Boyd, a professor at the University of Ottawa, ran a contest where fellow mathematicians were asked to find salesman tours through a set of fifty points. The restriction was that contestants could use by-hand calculations only. After several days a winner was announced, but the winning tour was in fact not a best-possible solution to the problem. Some of the world's top experts were unable to solve a small instance of the TSP without the help of their computers and software.

We relate the story of Boyd's contest to emphasize that, although the numbers game does not prove the TSP is actually tough to solve, to date no one has discovered a method that is both guaranteed to produce optimal tours and is provably efficient in a way we describe later in the chapter. This does not contradict our earlier statement that the solution of instances with hundreds of cities is routine: we may not have a provably efficient method for the TSP, but through a sixty-year research effort it is now possible to solve many typical examples of the problem, including those with a thousand or more cities. Discrete optimization may be difficult, but a take-no-prisoners approach can lead to optimal solutions in a surprising number of settings.

SHORTEST PATHS

If you have used one of the many Web services for obtaining driving directions, then you have witnessed the nearly instantaneous solution of an optimization problem. The underlying model in this application is the shortest-path problem. Finding shortest paths again amounts to locating needles in huge haystacks, but in this case there are easy-to-apply methods that would allow you, for example, to find an optimal solution using by-hand computations on road networks of modest size. The existence of such techniques does not, however, explain how a Web service can produce so quickly an optimal solution over an enormous network of points and roads. Large-scale problem instances call for sophisticated data structures and algorithms, bringing state-of-the-art computer science techniques to deliver on-the-fly solutions.

PERFECT MATCHINGS

A *perfect matching* of an even number of points is a partition of the points into pairs, that is, each point is paired up with exactly one other point. Given as input a cost assigned to each pair, the problem is to compute a perfect matching of minimum total cost. Perfect-matching models arise in a number of contexts, such as organizing cell-phone towers into pairs to serve as backups for one another, or pairing up human subjects in medical tests.

In the early 1960s, Jack Edmonds designed an algorithm for computing minimum-cost perfect matchings. His method involves deep ideas and it can hardly be called easy-to-apply, but it is provably efficient, setting matching problems apart from those like the TSP. Edmonds's method is also practical, although it took a long line of research papers to arrive at an effective computer implementation of his complex algorithm. The work here involved a combination of mathematics, algorithmic theory, data structures, software engineering, and computer hardware. Quite an effort. Indeed, the complexity of solution methods in discrete optimization can sometimes make practical work challenging, but the rewards in terms of problem solvability can be great.

THE GOAL

Difficult, large, and complex. These are the characteristics of computational settings in discrete optimization. The aim of our book is to take the reader into this arena, covering aspects of the subject that are typically skipped over in standard presentations. The book can be viewed as a how-to guide for practical work, ranging from the solution of models with tough-guy reputations, such as the TSP, through those like the shortest-path problem, where successful application requires the solution of instances of exceptionally large scale, and the perfect-matching problem, where efficiency can be gained only through complex methodology. Throughout the book the focus will be on integrating the mathematics of discrete optimization with the technology of modern computing platforms.

1.1 NEED FOR SPEED

As a warm up, let's take a look at finding solutions to instances of the TSP. You might be surprised by this choice, since a generally efficient algorithm is not known for the problem. This is the point, however, allowing us to discuss an implementation of an extremely simple method. The tradeoff is that we must restrict ourselves to tiny test instances. But don't worry, later in the book we will see implementations of much more sophisticated algorithms, including state-of-the-art attacks on the TSP. And although the method we describe now is simple, it will bring up several important general issues about speed and the need to avoid repeated computations. Indeed, our discussion follows the very nice TSP article "Faster and faster and faster yet" by Jon Bentley [10], where he writes "And finally, we'll go hog-wild and see speedups of, well, billions and billions."

To begin, suppose we have ten cities to visit and we wish to do so using the shortest possible tour. Even more precisely, suppose we wish to visit the ten cities in the United States displayed in Figure 1.1. These locations were part of a 33-city TSP contest,

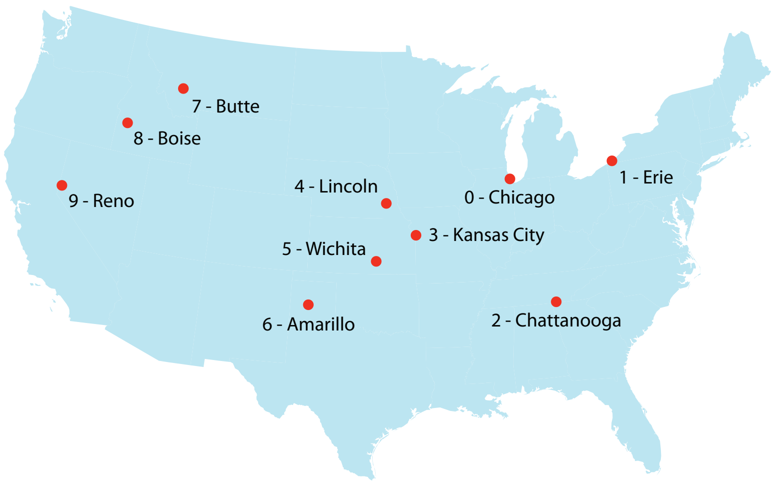

Figure 1.1: Ten cities in the United States.

ran by Procter & Gamble in the 1960s, that included a $10,000 prize for the shortest tour. The city-to-city driving distances displayed in Table 1.1 are taken from the contest rules. In this example, for each pair of cities A and B, the distance to drive from A to B is the same as the distance to drive from B to A. When the input data satisfy this property, the problem is known as the symmetric TSP. Our implementation takes advantage of this symmetry, but it is easy to modify the general method to work also with asymmetric data.

As a preliminary step, we need to decide how to represent a potential solution to the problem. The simplest choice is to describe a tour by giving the itinerary for the salesman. For example, starting at Chicago, visit the remaining cities in the order Erie, Chattanooga, Wichita, Amarillo, Reno, Boise, Butte, Lincoln, Kansas City, and back to Chicago. Using the city labels listed in Table 1.1, we can represent the tour as an array of ten numbers:

0	1	2	5	6	9	8	7	4	3

A picture of the array is fine for a discussion, but we also need to describe it in a form that can be manipulated by a computer code. Now this gets into a religious matter.

	0	1	2	3	4	5	6	7	8	9
0 Chicago	0									
1 Erie	449	0								
2 Chattanooga	618	787	0							
3 Kansas City	504	937	722	0						
4 Lincoln	529	1004	950	219	0					
5 Wichita	805	1132	842	195	256	0				
6 Amarillo	1181	1441	1080	563	624	368	0			
7 Butte	1538	2045	2078	1378	1229	1382	1319	0		
8 Boise	1716	2165	2217	1422	1244	1375	1262	483	0	
9 Reno	2065	2514	2355	1673	1570	1507	1320	842	432	0

Table 1.1: City to city road distances in miles.

In a large group of potential readers, we will no doubt find votes for any number of programming languages to use in our presentation, such as C, C++, Java, Python, and Matlab, to name just a few. Each language has its own strengths and weaknesses. Keeping things somewhat old school, we have adopted C throughout the book, for the good reason that most of the actual computer codes we draw upon for examples are written in C. We will, however, attempt to use the language in a vanilla fashion, to keep things familiar and to allow for easy translation.

In the C language, an array of ten integers, called `tour`, is declared by the statement

```
int tour[N];
```

where `N` is defined to be the number of cities. The first element of the array is accessed as `tour[0]`, the second element is `tour[1]`, and so on. For example, equipped with an integer-valued function `dist(i,j)` that returns the travel distance between city i and city j for any pair of cities, the following code will compute the length of a tour.

```
int tour_length()
{
   int i, len = 0;

   for (i = 1; i < N; i++) {
      len += dist(tour[i-1],tour[i]);
   }
   return len+dist(tour[N-1],tour[0]);
}
```

The for-loop runs through the first `N-1` legs of the tour, adding the road distance to the integer variable `len`; the final addition takes care of the leg back to the starting city.

Using `tour_length()` we can compute that our sample tour, drawn in Figure 1.2, covers the United States trip in a total of 6,633 miles. This looks pretty good,

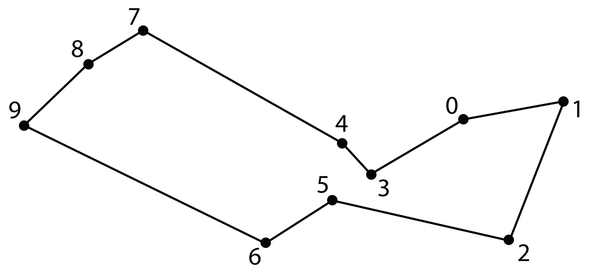

Figure 1.2: Tour of length 6,633 miles.

but is the tour actually the shortest? One way to check is to simply run through all possible tours of the ten cities and compute the length of each. An uninteresting, brute-force approach, but it works fine for tiny instances.

Enumerating all tours

To carry out the brute-force search, we need only manipulate the first `N-1` elements of the `tour` array to generate all possible permutations of the values they store, since this corresponds to fixing the final city in the tour. The tool we use in the manipulation is a `tour_swap()` function that exchanges the element stored in position i with the element stored in position j.

```
void tour_swap(int i, int j)
{
   int temp;
   temp = tour[i]; tour[i] = tour[j]; tour[j] = temp;
}
```

For example, `tour_swap(0,8)` will exchange the elements stored in positions 0 and 8. Now, to generate all permutations of the first `N-1` elements, we will one-by-one move each of the cities stored in the first `N-1` positions to position `N-2`, and then generate all permutations of the symbols now in the first `N-2` positions. This is accomplished by calling the following function as `permute(N-1)`.

```
void permute(int k)
{
   int i, len;

   if (k == 1) {
      len = tour_length();
      if (len < bestlen) {
         bestlen = len;
         for (i = 0; i < N; i++) besttour[i] = tour[i];
      }
   } else {
      for (i = 0; i < k; i++) {
         tour_swap(i,k-1);
         permute(k-1);
         tour_swap(i,k-1);
      }
   }
}
```

The argument `k` indicates that all permutations of the first k elements of `tour` should be generated. If `k` is 1, then we have reached our next full permutation of the `N` cities and we therefore compare the corresponding tour length with the best value we have found previously. Whenever we find a new best tour, we record its value in a variable `bestlen` and record the tour itself in an array `besttour`.

You might want to spend a few minutes convincing yourself that `permute()` works as claimed. At first glance it appears to be too simple, but the complexity is hidden by the fact that the function calls itself recursively to generate the permutations on one fewer elements. To give you a feeling for the operation of the function, we list in the three columns of Table 1.2 the order in which `permute()` runs through all tours for three, four, and five cities respectively.

To test the code yourself, you can refer to the listing given in Appendix A.1, where we have added missing pieces to read the distance table, print the shortest tour and its length, and wrap things in a `main()` function as required by the C language. Once the code is compiled, you won't have to wait long for the result: on a current desktop computer, the 10-city example runs in under 0.01 seconds. The optimal tour is displayed in Figure 1.3. Its length of 6,514 miles improves on our sample tour by traveling directly from city 4 to city 0, picking up city 3 on the lower side of the tour.

If the goal is to solve the 10-city instance only, then we should declare victory at this point. The algorithm is simple, the full code listing has only 72 non-blank lines, and it runs in a hundredth of a second of computing time. A job well done. But why stop at ten cities? With such a fast running time, we ought to be able to push ahead to slightly larger examples. Here we turn to the *random Euclidean TSP*, where, in our case, each city is defined by a point (x, y) with x and y integers chosen at random between 0 and 1,000. The travel costs are the straight-line distances between pairs of

1	0	2	1	2	0	3	1	2	3	0	4
0	1	2	2	1	0	3	2	1	3	0	4
			2	0	1	3	2	3	1	0	4
			0	2	1	3	3	2	1	0	4
			1	0	2	3	1	3	2	0	4
			0	1	2	3	3	1	2	0	4
							3	2	0	1	4
							2	3	0	1	4
							2	0	3	1	4
							0	2	3	1	4
							3	0	2	1	4
							0	3	2	1	4
							1	3	0	2	4
							3	1	0	2	4
							3	0	1	2	4
							0	3	1	2	4
							1	0	3	2	4
							0	1	3	2	4
							1	2	0	3	4
							2	1	0	3	4
							2	0	1	3	4
							0	2	1	3	4
							1	0	2	3	4
							0	1	2	3	4

Table 1.2: Tours generated by permutation function for 3, 4, and 5 cities.

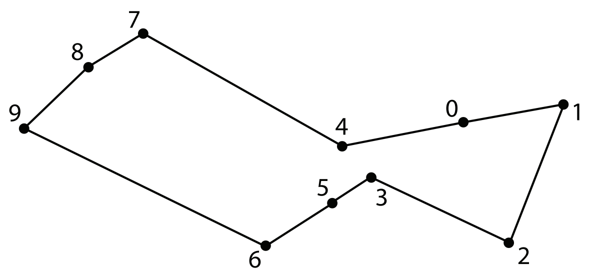

Figure 1.3: Optimal tour, length 6,514 miles.

points, rounded to the nearest integer. This test bed has the nice property that we can readily produce examples as we slowly increase the number of cities n.

So, how does the brute-force code stack up? Looking at just one sample instance for each value of n, here are the running times in seconds.

Cities	10	11	12	13	14	15
Seconds	0.01	0.1	1.1	17	215	3322

At this rate of growth, solving a twenty-city instance would likely take over a billion seconds. This is where we have the need for speed.

Saving time and pruning tours

A first opportunity for a speed upgrade is the current overuse of the `tour_length()` function. Indeed, the time-per-tour can be reduced easily by keeping track of the length of the path we build as the permutation is created. To handle this, we include a second argument, `tourlen`, in `permute()` to keep track of the accumulated path length. In the recursive call to the function we add to `tourlen` the distance between cities $k-1$ and k.

```
void permute(int k, int tourlen)
{
   int i;

   if (k == 1) {
      tourlen += (dist(tour[0],tour[1]) +
                  dist(tour[N-1],tour[0]));
      if (tourlen < bestlen) {
         bestlen = tourlen;
         for (i = 0; i < N; i++) besttour[i] = tour[i];
      }
   } else {
      for (i = 0; i < k; i++) {
         tour_swap(i,k-1);
         permute(k-1, tourlen+dist(tour[k-1],tour[k]));
         tour_swap(i,k-1);
      }
   }
}
```

At each recursive call, the `tourlen` argument is equal to the length of the path following the cities in order from `tour[k]` to `tour[N-1]`. In the $k = 1$ case, two additions are used to account for the initial road in the tour and for the return journey from the final city. The modified code reduces the running times to the following, for the same set of test instances.

Cities	10	11	12	13	14	15
Seconds	0.006	0.04	0.6	11	116	1576

The computation has been cut by half for the 15-city example, but much more important is that the code is now equipped with a useful bit of information. Namely, we know that no matter how we complete the `tour[k]` to `tour[N-1]` path into a tour, it will have length at least the value of `tourlen`, assuming, quite reasonably, that travel costs cannot be negative numbers. This means that if `tourlen` ever reaches the value of our best-known tour, then we need not examine the tours that would be obtained by completing the current path. Thus, adding the statement

```
    if (tourlen >= bestlen) return;
```

at the start of `permute()` allows us to prune the search. This one line of code drops the running time for the 15-city instance down to 5 seconds.

Cities	10	11	12	13	14	15	16	17	18	19	20
Seconds	0.003	0.01	0.05	0.3	2	5	34	22	56	5325	646

The improvement is so dramatic that we went ahead and pushed on to a 20-city example, solving it in under 11 minutes of computing time. In these results, however, you begin to see that our use of a single test instance for each value of n is not a good idea in a computational study. Indeed, the running time for 19 cities is much greater than the time needed for 20 cities. This is due to the complex operation of the algorithm, where pruning can happen more quickly for some distributions of points than it does for others. To be rigorous we would need a statistical analysis of a large number of test instances for each n, but allow us to continue on with our simple presentation. We will have a detailed discussion of appropriate test environments in Chapter 14.

Continuing with our implementation, the pruning idea looks like a winner, so let's carry it a bit further. When we cut off the search, we are assuming only that the remaining path through the cities stored in positions 0 through $k-1$ must have length at least zero. This is a simple rule, but it ignores any additional information we may have concerning travel costs. An improvement comes from the following observation: to complete a tour, we must select distinct roads entering each of the cities stored in `tour[k-1]`, `tour[k-2]`, down to `tour[0]`, as well as a road returning to the final city. We don't know beforehand which of these roads make up the shortest path through the cities, but each selected road must be at least as long as the shortest among all roads entering the specific city. So here is what we do. Using a third argument, `cheapsum`, in the function `permute()`, we keep track of the sum, over all cities stored in positions 0 through $k-1$ and the final city, of the shortest roads entering each city. The pruning test is then changed to the following line of code.

```
    if (tourlen+cheapsum >= bestlen) return;
```

To update `cheapsum` we compute, at the start of the code, an array `cheapest` that stores the length of the shortest road entering each of the n cities in the problem. The initial call to `permute()` sets `cheapsum` to be the sum of all of the values stored in the array, and the recursive call becomes

```
permute(k-1, tourlen+dist(tour[k-1],tour[k]),
             cheapsum-cheapest[tour[k-1]]);
```

Thus `cheapsum` is decremented by the length of the shortest road entering the next city to be placed into the tour. A relatively minor change to the code, but with a big impact: the running time for the 20-city instance decreases by a factor of 1,000, now coming home in ten seconds. Here are the running times obtained for instances up to 25 cities.

Cities	15	16	17	18	19	20	21	22	23	24	25
Seconds	0.1	0.3	0.1	0.2	4	5	350	36	246	572	9822

Compared with our starting code, the total improvement is now up to a factor of one hundred million for the 20-city Euclidean TSP, and more help in on the way.

TREES AND TOURS

The use of `cheapsum` brings in travel costs for completing a path into a tour, but only in the most elementary way. Indeed, if we put together the roads that determine the values in the `cheapest` array, they would typically not look anything like a path. In fact, the roads would likely not even join up into a connected network. Finding a shortest path to link up a subset of cities is similar in difficulty to the TSP itself, but finding the cheapest way to connect a subset of cities is something that is quite easy to do. We can use this to obtain a better lower bound on the cost to complete a path into a tour, and thus a stronger method for pruning our search.

A minimal structure for connecting a set of cities is called a *spanning tree*. A minimum-cost spanning tree for the 10-city Procter & Gamble instance is displayed in Figure 1.4. It is clearly not a single path through the set of cities, but its length of 4,497

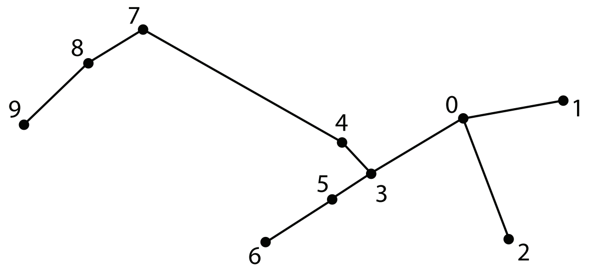

Figure 1.4: Minimum spanning tree, length 4,497 miles.

miles is at least as short as any single path, since each such path is itself a spanning tree.

Efficient methods for computing minimum spanning trees were discovered in the 1920s, and the well-known algorithms of Kruskal and Prim were published in the

1950s. It is the method of Prim that fits best within our TSP application. A fast implementation of his algorithm, suitable for small examples, can be coded in about twenty lines of C, but we leave the details until our discussion in Chapter 3. For now, suppose we have a function `mst(int d)` that returns the length of a minimum spanning tree through the first d cities stored in the `tour` array and the final city `tour[N-1]`. With this new weapon, the `cheapsum` argument and pruning test can be replaced by the statement

```
    if (tourlen+mst(k+1) >= bestlen) return;
```

at the start of the `permute()` function. This more powerful bound brings the solution time for the 20-city instance down to under a second.

Cities	20	21	22	23	24	25	26	27	28	29	30
Seconds	0.6	0.8	0.7	1.5	1.4	10	0.6	17	35	2	44

The improved running times point out that the spanning-tree bound is a valuable tool for pruning the search, so we ought to make good use of it. In this direction, we can attempt to get a low value for `bestlen` early in the computation, since this will allow for quicker pruning of our tours. A good place to start is by calling the following function to compute a *nearest-neighbor tour*.

```
void nntour()
{
    int i, j, best, bestj;

    for (i = 0; i < N; i++) tour[i] = i;
    for (i = 1; i < N; i++) {
        best = MAXCOST;
        for (j = i; j < N; j++) {
            if (dist(tour[i-1],tour[j]) < best) {
                best = dist(tour[i-1],tour[j]);
                bestj = j;
            }
        }
        tour_swap(i, bestj);
    }
}
```

The algorithm begins at city 0 and at each step adds as the next city the nearest one that has not yet been visited. At very little computational cost, we can initialize `bestlen` to be the value of the tour we obtain. Doing so gives the following improvements in running times for our test instances.

Cities	25	26	27	28	29	30	31	32	33	34	35
Seconds	7	0.5	10	7	0.7	11	155	969	189	1168	9

Notice that we are now handling examples with n in the mid 30s, thus giving us a shot at solving the full 33-city Procter & Gamble instance. Indeed, the optimal tour of 10,861 miles, displayed in Figure 1.5, is found by the code in 47 seconds. Unfortunately we are fifty years too late to make a claim for the \$10,000 prize.

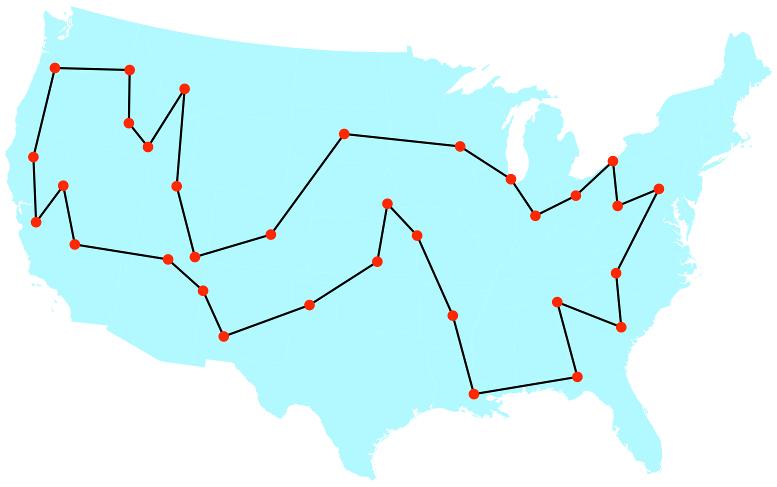

Figure 1.5: Optimal tour for 33-city Procter & Gamble contest.

BIGGER GUNS

The sequence of improvements we have described have added a total of 35 lines of C to our original implementation. That leaves a tidy computer code for small instances of the TSP; a full listing can be found in Appendix A.2. The next two upgrades are a bit more heavy duty, each involving around fifty additional lines, but the payoffs are dramatic.

First, it makes sense to try to obtain an even better starting value for `bestlen` by working harder at our preliminary-tour construction. There are many TSP heuristic methods that could be considered, but we focus on one of the simplest and earliest ideas, proposed by Merrill Flood in the mid 1950s. Flood's method is to take a nearest-neighbor tour and repeatedly make improving *two-opt moves*. A two-opt move is the operation of removing a pair of roads and reconnecting the resulting two paths. For example, the move displayed in Figure 1.6 improves by 301 miles a nearest-neighbor tour for the ten-city Procter & Gamble instance. The *two-opt algorithm* repeatedly makes improving two-opt moves until a tour is obtained such that no further improving moves exist.

The results of the two-opt algorithm are usually pretty good for small TSP instances, but to give our code a chance to find a near-optimal initial tour, we imple-

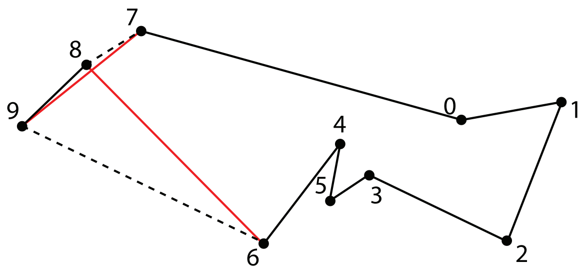

Figure 1.6: Two-opt move, replacing (6-8, 7-9) by (6-9, 7-8).

mented a for-loop to run the algorithm n times, once for each of the nearest-neighbor tours obtained by considering each possible city as the starting point.

The second improvement adopts an idea of Bentley [10]. If you observe the operation of the TSP enumeration algorithm equipped with the spanning-tree pruning mechanism, then you will notice that the same spanning trees are computed over and over again. This violates a golden rule of computer implementations. Bentley's suggestion is to use a data structure called a *hash table* to store the length of each spanning tree we compute, and then to check the table before each new spanning-tree computation to see if we know already the result. Hash tables are one of the dynamos of efficient computer codes in discrete optimization and we discuss their operation in Chapter 2.

Together with these two high-powered improvements, we could not resist making one further minor tweak. With our symmetric data, it does not matter in which direction we travel around a tour. Thus, our algorithms need only consider those tours such that city 0 comes before city 1 as we work our way back from the fixed final city `tour[N-1]`. To arrange this, we include a `havezero` argument in the `permute()` function and set it appropriately when we encounter city 0. This tweak results in a minor speed up only, but for three lines of code is seems worthwhile.

Test results for the nearest-neighbor-based code and our three improved codes are reported in Table 1.3, considering random Euclidean instances with up to 45 cities. The codes get faster as we move from Nearest, to Two-Opt, to Hash Table, to Zero-One, since we incorporate all of the previous upgrades when we add the next idea. You would have to get out a calculator to estimate the total speed up we have obtained over our original brute-force code, but solving a 45-city instance in a little over a minute definitely qualifies as "billions and billions."

Some TSP history

The above computational results put us into the range of some historically important test instances of the TSP. Indeed, as the problem first began to circulate in the mathematics community in the 1930s and 1940s, it was sometimes called the "48-states problem," referring to its instance of finding an optimal tour to visit a city in each of the 48 states of the United States. The grandfather of all TSP papers, written by George

Cities	Nearest	Two-Opt	Hash Table	Zero-One
35	9	0.9	0.1	0.1
36	11729	133	5	3
37	265	6	0.4	0.3
38	43	32	2	1
39	2713	150	5	4
40	17379	1968	84	63
41	461	187	8	8
42	4528	840	18	13
43	2739	132	5	4
44	14021	6781	77	49
45	190820	8156	172	73

Table 1.3: Running times in seconds for four simple TSP codes.

Dantzig, Ray Fulkerson, and Selmer Johnson [27] in 1954, considered such an example, throwing in also Washington, D.C. for good measure. Through a trick with their particular data set, Dantzig et al. were able to reduce their 49-city instance to a 42-city problem, which they solved with by-hand computations. Their result stood as a world record in TSP computation for 17 years, until Michael Held and Richard Karp [45] produced an algorithm, computer code, and computational study that solved several larger examples, including 48-city and 57-city tours through the United States.

Our tiny-TSP code solves the Dantzig et al. 42-city instance in 0.3 seconds, the Held-Karp 48-city instance in 0.2 seconds, and the 57-city instance in 656 seconds. And, in case you are wondering, Sylvia Boyd's 50-city instance solves in 8 hours.

The techniques we have used were all known in the mid 1950s, including minimum spanning trees, the nearest-neighbor algorithm, the two-opt algorithm, and hash tables. Of course, what emphatically was not available in the 1950s was the high-powered computer we adopted or the great platform provided by the C programming language. Maybe this was good fortune. The linear-programming methods developed by Dantzig et al. to first solve the 48-states challenge were a true revolution, and their techniques continue to dominate the research and practice of attacks on difficult problems in discrete optimization.

Transforming the travel costs

Our discussion has taken us through a sequence of clear steps to improve the tiny-TSP code, in each case either avoiding certain computations or avoiding the explicit examination of large numbers of non-optimal tours. Rather than a clear improvement, our next upgrade is a heuristic attempt to transform instances of the TSP to run more quickly on our existing code. The upgrade may be heuristic, but the running-time improvements are clear: the 45-city instance from Table 1.3 will solve in 0.3 seconds and Boyd's 50-city example will solve in 0.4 seconds.

The idea is to create an equivalent TSP where the spanning-tree bound is likely to be more effective. The transformation tool, described by Merrill Flood [33] in connection with his test of the nearest-neighbor algorithm, is based on the following observation: if we subtract the same value δ from the travel cost of every road meeting a selected city, then we have not altered the TSP. Indeed, every tour in the transformed problem is exactly 2δ shorter than the same tour in the original problem. So both instances rank the tours in the same way. Moreover, we can repeat the process, for each i subtracting a value δ_i from every road meeting city i. In other words, if the cost to travel from city i to city j in the original instance is $dist(i, j)$, then the travel cost in the transformed instance is $dist(i, j) - \delta_i - \delta_j$. We are free to select any values we like for the δ's, so let's try to create new travel costs such that a minimum spanning tree in the transformed instance resembles a tour.

Flood selected δ's to maximize the sum $\sum(2\delta_i : i = 0, \ldots, n-1)$ subject to the condition that all travel costs remain nonnegative, that is, for each pair of cities i and j we have the constraint $\delta_i + \delta_j \leq dist(i, j)$. For Euclidean instances of the TSP,

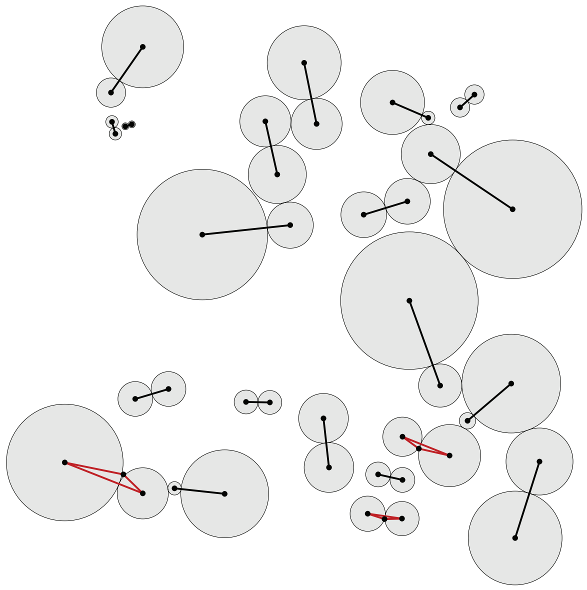

Figure 1.7: Flood's δ-values for a 45-city TSP instance.

such δ's can be visualized as disks of the given radius centered at each city, where the constraints on the δ's correspond to the condition that disks do not overlap. The values for the 45-city test instance are displayed in Figure 1.7. The black and red edges drawn between pairs of cities in the figure indicate the method Flood used to compute his δ's. Indeed, Flood employed an algorithm to solve a variation of an optimization problem known as the *assignment problem*. In this case, the problem assigns 1/2 and 1 values to edges such that each city meets a total value of one, that is, either two red edges or one black edge. We discuss the assignment problem in Chapter 2, showing how standard solution methods produce the δ values as a by-product.

Flood's transformation brings cities next to one another, allowing both the optimal tour and the minimum spanning tree to share many roads that now have cost zero. It is thus reasonable to test the use of the transformation within our tiny-TSP code, so we added around 60 lines of C to implement an assignment-problem algorithm and obtained the following results.

Cities	45	46	47	48	49	50
Seconds	3	11	201	10735	101	18

The running times are better, but the drawing in Figure 1.7 suggests that further improvements should be possible. Indeed, the assignment-problem δ's produce many isolated pairs of cities, meaning that there are no zero-cost roads to join the pairs into a tour or tree. A way to handle this is to allow disks to overlap and in the optimization problem include a penalty for the length of the overlapping portion. The resulting disks for the 45-city TSP instance are displayed in Figure 1.8, obtained by solving the *fractional-2-factor problem* that assigns 1/2 and 1 values to edges such that each city meets a total value of two; this problem is discussed in Chapter 9. The penalties en-

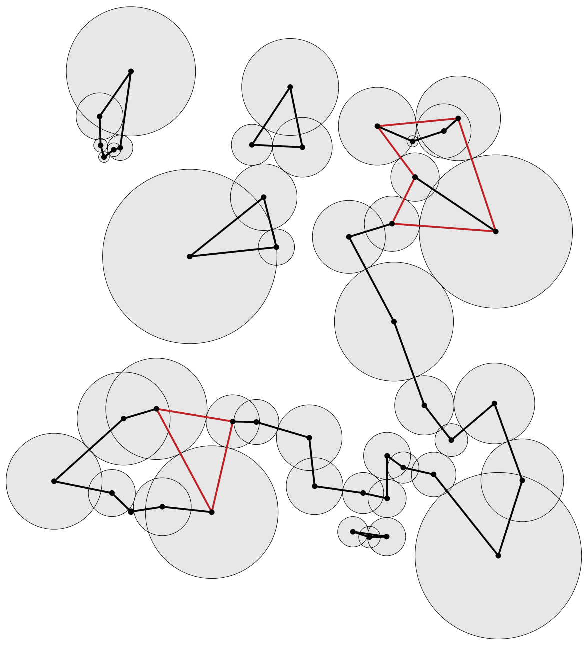

Figure 1.8: Improved δ-values for a 45-city TSP instance.

sure that each disk overlaps at most two other disks, creating chains of disks that either touch or overlap, that is, paths of roads that have either zero or negative travel costs in the transformed problem. These paths allow the optimal tour and minimum tree to resemble each other even more than in the assignment-value transformation, leading to much better running times.

Cities	45	46	47	48	49	50
Seconds	0.2	6	1	86	2	0.5

In our code, we need around 70 lines of C to compute the 2-factor δ's, so it is nearly a one-for-one replacement for the implementation of Flood's transformation.

So far we are using still Dantzig-Fulkerson-Johnson-era machinery. For our final improvement we jump ahead to the late 1960s and employ an idea due to Held and Karp [44, 45]. It was used originally within a sophisticated *branch-and-bound* algorithm for solving the TSP, repeatedly computing new spanning-tree-based lower bounds for subproblems obtained by including or excluding certain roads. We describe the Held-Karp branch-and-bound method in Chapter 8, but our use of the idea here is much simpler.

Held and Karp work with an object known as a *1-tree*, obtained by selecting one city and the two cheapest roads meeting it, together with a minimum spanning tree of the remaining cities. Every tour is an example of a 1-tree, so the minimum 1-tree gives a bound on the cost of any tour. Given travel costs transformed by δ values, Held and Karp compute a minimum 1-tree. For each city i that meets more than two roads in the 1-tree, they increase the value of δ_i, and for each city i that meets only one road in the 1-tree they decrease δ_i. The modified δ's guide the 1-tree towards the case where all cities meet exactly two roads, that is, towards the case where the 1-tree is a tour. The process is repeated many times, gradually decreasing the amounts by which the δ's are changed. In an attempt to not bias the method by a particular choice of a special city, we create our δ's by averaging the values obtained by considering in turn each of the n cities as the special one. Thus, we apply the Held-Karp process n times to obtain our δ's. The implementation adds about 40 lines of C to our tiny-TSP code.

In Figure 1.9 we display the minimum spanning tree for the 45-city example with costs transformed by the Held-Karp δ's. You can see that the tree in the transformed problem is rather tour-like, with many cities included on long paths. It is this property that leads to the outstanding performance of the new code, as indicated by the following running times.

Cities	45	46	47	48	49	50
Seconds	0.3	0.5	0.3	0.9	0.9	0.4

The code also solves the 42-city, 48-city, and 57-city instances through the United States in 0.3 seconds, 0.3 seconds, and 12 seconds, respectively, and, as we mentioned, Boyd's 50-city example solves in 0.4 seconds.

Moving up to larger random Euclidean instances shows off even better the power of the travel-cost transformations. The four rows in the following table give running times in seconds for instances with up to 65 cities, using the original travel costs, the assignment-problem costs, the fractional-2-factor costs, and the Held-Karp costs, re-

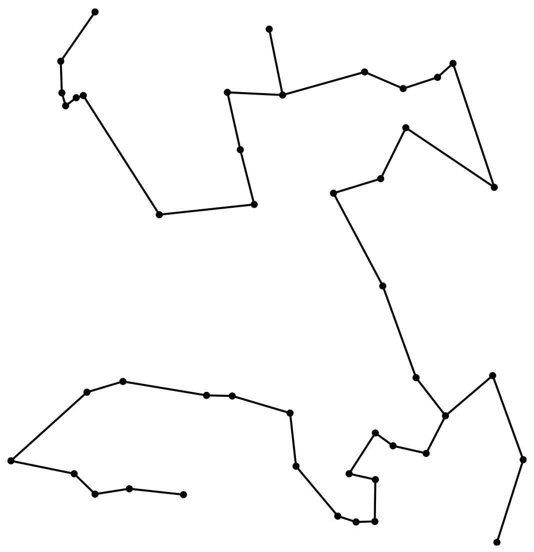

Figure 1.9: Minimum-spanning tree with Held-Karp costs for a 45-city TSP instance.

spectively; the "**" entries indicate trials that did not complete in 200,000 seconds.

Cities	50	55	60	65
Original Costs	46	9919	**	**
Assignment Costs	18	19314	1619	**
2-Factor Costs	0.5	59	13	6552
Held-Karp Costs	0.4	0.9	0.5	2

The code with the original travel costs was unable to solve the 60-city instance, whereas the Held-Karp transformation allowed us to complete the work in half a second. It thus seems safe to assume that by the time we get up to the 65-city example we can add another layer of "billions" to our list of speed ups.

THE POWER OF BETTER PRUNING VALUES

Let's wrap things up with a final upgrade, allowing us to push our tiny-TSP computations up to instances having 100 or more cities. To accomplish this we return to our earlier observation that a good starting value for `bestlen` is crucial for pruning the search for an optimal tour. We pursued this theme with the use of the nearest-neighbor and two-opt algorithms, but with the stepped up power of the Held-Karp transformed spanning-tree lower bound it is time to look to further improvements in the initialization of `bestlen`.

Ideally we would somehow know the value of an optimal tour and set `bestlen` appropriately. We don't of course have this knowledge, but we can cheat and make an aggressive guess for an initial value for bestlen. Using the guessed value, if we conclude our search without finding a tour of value bestlen or better, then we repeat the process starting with a slightly larger guess. Thus, we create a loop, increasing bestlen at each iteration until we find the optimal tour. To be more specific, we can begin by setting bestlen to the value of the 1-tree bound. If the search fails, then we repeatedly increase bestlen to the value $(\alpha * \texttt{bestlen}) + 1$ rounded down to the nearest integer, where α is any constant larger than or equal to one. In our tests we set $\alpha = 1.001$. This is a general approach that can work well whenever we have a search problem that comes with a very strong lower bound for pruning candidate solutions.

As an alternative to this staged search, we can attempt to upgrade the two-opt algorithm we use for computing an initial tour. A good choice here is the *Lin-Kernighan algorithm* proposed by Shen Lin and Brian Kernighan [62] in 1973. We describe this method in detail in Chapter 11; it remains a champion technique for tour finding and a model for a class of heuristics known as *local-search algorithms*. To fit into our tiny-TSP code, we implemented a simplified version of the heuristic that requires about 50 additional lines in C.

The table below reports running times in seconds for three versions of our code, using the two-opt tour we described earlier, using the staged approach of increasing bestlen, and using the Lin-Kernighan tour, respectively.

Cities	70	75	80	85	90	95	100
Two-Opt Bound	1	26	118	2577	221	1041	2336
Staged Search	0.6	1	11	22	12	529	247
Lin-Kernighan Bound	0.6	1	27	9	10	183	87

The Lin-Kernighan algorithm often delivers the optimal tour on this class of instances and thus typically achieves the best results, but the performance of the staged approach is impressive for such a simple tweak to the existing code.

As our last word, we note that 100 cities puts us within range of another milestone in TSP computation, namely Martin Grötschel's optimal 120-city tour of Germany [41], found in 1975. Not that we encourage calculations such as the following, but this instance of the TSP would require roughly 10^{190} seconds to solve using the simple code that began our discussion some fifteen pages ago. On the other hand, the Lin-Kernighan-equipped version of our code needs only 412 seconds to solve Grötschel's example. That makes the final tiny-TSP code about twenty-one factors of a billion faster than the initial code. Implementation details can indeed make a difference!

1.2 THE ART OF IMPLEMENTATION

It would be nice to have a set of golden rules to guide implementation work, leading us quickly, for example, to the final version in the tiny-TSP example. Nice, but not likely. Indeed, in a mild complaint to one of the authors of this book, a colleague at AT&T Labs noted that hard-and-fast implementation advice he receives one month often contradicts hard-and-fast advice he received the previous month. That can happen. Nevertheless, we feel safe in laying down a number of general principles that will, at least, allow us to set the tone for the problem-specific studies to come later. But keep in mind that implementation is a creative activity, so general principles must necessarily leave plenty of room for interpretation.

KNOW WHAT YOU ARE DOING

Let's begin with a principle that seems unassailable. Namely, writing a code fragment should be like proving a lemma. You might sometimes be wrong, but you should always have in your mind a proof that the fragment does what you have planned. The idea that one can quickly type in code, toss it over to a test environment, and hope for the best is doomed to fail when the fragment is part of a large implementation project.

NO REPETITIONS

Nothing slows down an algorithm more than repeatedly performing the same computation. This is a trivial statement, and there are sometimes easy fixes. At a deeper level, however, avoiding repetitions, and other superfluous computations, is a large part of the implementation art.

MEMORY MATTERS

In these days, when a few gigabytes of random-access memory costs less than a good meal, it is tempting to think we have unlimited workspace for our computations. As a general rule, however, memory usage definitely still matters. Indeed, modern computers have complex, hierarchical memory architectures, going from registers, to several grades of on-chip cache, to off-chip random-access memory, and finally to disk space. Access to information from each level in the hierarchy comes at a great time penalty over access from the earlier level. Effective codes are those that squeeze as much computation as possible onto each level. Attempting to manage this can become overwhelmingly complex, but it is typically sufficient to keep in mind simply that memory matters, use only what you need.

GETTING LUCKY

A little luck is always good, particularly when attempting to solve a problem instance of size or complexity well beyond what could be expected from a worst-case analysis of a particular solution method. The saying "fortune favors the brave" does not really apply here, but codes should be designed so that brave users have a fighting chance. Here is what we mean. Suppose you have an algorithm that is known to require at most n^2 steps to solve a problem instance having n points. This should not in general mean that it is fine to include a double loop in your code, running through all n^2 pairs of points no matter what specific example is given as input. Such a construction turns the n^2 worst-case algorithm into an n^2 best-case running time.

Turning worst to best is bad practice. A worst-case analysis must not become an excuse to relax. Indeed, an understanding of worst-case behavior should direct you to avoid the bottleneck computation if at all possible when processing specific input. Almost any sophisticated implementation work will serve as an example of this principle.

To mention just one, there are implementations of Edmonds' matching algorithm that have as worst-case performance a multiple of n^4 steps, for an n-point input, and yet are run routinely on examples having more than a million points. A quick calculation will convince you that the authors of these codes did not surrender and create an n^4 best-case implementation.

REUSE

"Reuse, Reduce, and Recycle" is the snappy slogan of many conservation campaigns. This is not bad advice also for algorithm implementers. By this we do not mean recycling blocks of previously written code, although this itself is sound practice if you have a supply of good routines to draw upon. The principle we have in mind is to reuse or recycle previously computed objects during the execution of a solution procedure. Clear examples of this are various re-start methods that make minor adjustments to known optimal solutions when presented with slightly altered input data.

The reuse principle overlaps, of course, with the no repetitions principle: if you have computed something once, why compute it again? Going beyond this, it pays to consider how one can get multiple uses out of a computed structure.

EXPLOIT ALTERNATIVES

There is often a choice of solution methods to tackle a given problem, and it is usually the case that one of the methods will perform better than others on particular input data, while a second method performs better on other inputs. If there are easily computed parameters that allow you to detect which of the methods is likely to be superior on a given instance, then it is simple enough matter to have your code make the decision on-the-fly. A typical scenario, however, will not have such an easy selection rule. Indeed, the behavior of a complex solution process may be very difficult to determine without actually applying the process itself. In such cases it may be possible to go ahead and run several solution strategies in parallel, halting the software when the fastest of these completes its work. If only a single thread of execution is available, then it may be possible to see-saw between two methods A and B, applying first A for k seconds, then, if A was not successful, applying B for $2k$ seconds, followed by A for $4k$ seconds, and so on until the problem is solved by one or another of the methods. In short, it may pay to exploit alternatives when faced with a difficult computation.

1.3 COMPUTATIONAL COMPLEXITY

Our list of general implementation principles emphasizes the main theme of the book, namely, a dogged approach to the solution of problem instances in discrete optimization. This is at odds somewhat with the focus of algorithmic work in computer science, where results on finite examples are not of direct concern. Nonetheless, the theoretical studies of that community are a very important source of techniques for practical computation. The challenge is to sift through material to pick out algorithms and data structures that are appropriate for the particular instances that need to be solved. This topic will be discussed in the next two chapters, but it is also important to understand the basics of complexity theory, which serves as a guide in computer-science theory.

Algorithms and Turing machines

A solution technique for an optimization problem is most typically an *algorithm*, that is, a list of simple steps that together produce a solution to any instance of the problem. We have used freely the word algorithm in the text thus far, relying on the reader's intuitive concept of a step-by-step solution strategy, such as the one employed in the tiny-TSP discussion. To have a theory of complexity, however, the intuitive concept must be made precise. This issue came to the forefront in the early 1900s with David Hilbert's *Entscheidungsproblem*, that asks, roughly, whether there exists an algorithm that can decide if any given statement is, or is not, provable from a set of axioms.

At the time of Hilbert it was not clear how the concept of an algorithm should in general be defined. This is a deep question, but Alan Turing provided an answer in 1936, introducing a mathematical model known as a *Turing machine*. The hypothetical machine has a tape for holding symbols, a head that moves along the tape reading and writing symbols in individual cells, and a controller to guide the read/write head. It also has a finite set of states, with two special states being *initial* and *halt*. The controller is a table that indicates what the machine should do if it is in a particular state s and it reads a particular symbol x. The "what it should do" is to print a new symbol x' on the cell of the tape, move the head either left or right one cell, and enter a new state s'. To solve a problem, the machine starts in its *initial* state, with the input to the problem written on the tape; it terminates when it reaches the *halt* state.

Let's consider a simple case: given a string of 0's and 1's, determine if the number of 1's is odd or even. To construct a Turing machine for this problem we can have four states, *initial*, *odd*, *even*, and *halt*, two symbols, 0 and 1, and the transition table displayed in Figure 1.10. The table has a row for each symbol (including a blank "_")

	$initial$	odd	$even$
0	_, right, $even$	_, right, odd	_, right, $even$
1	_, right, odd	_, right, $even$	_, right, odd
_	0, _, $halt$	1, _, $halt$	0, _, $halt$

Figure 1.10: Transition table for a parity-checking Turing machine.

and a column for each state other than *halt*. The entry in the table is a triple, giving the symbol to write, the direction to move on the tape, and the next state. For example, if the machine is in state *odd* and we read the symbol 1, then we write a blank symbol in the cell, move one cell to the right, and change the state to *even*. Presented with a string of 0's and 1's, arranged in consecutive cells on the tape, and with the read/write head positioned on the leftmost symbol, the Turing machine will move to the right until it reaches a blank cell, indicating the end of the string. When it halts, a 0 is written on the tape if the number of 1's is even, and a 1 is written on the tape if the number of 1's is odd.

The beauty of the Turing machine model is that, although it is very simple, if something is computable on a modern day computer, then a Turing machine can be designed to carry out the computation. Indeed, it is a working assumption, known as the *Church-Turing Thesis*, that we can equate algorithms and Turing machines. This thesis is widely accepted and it gives the formal model of an algorithm used in complexity theory.

POLYNOMIAL-TIME COMPLEXITY

Turing provided a model for a theory of algorithms, but the arrival of digital computers in the 1950s quickly drew to attention to the issue of efficiency. It is one thing to know a problem can be solved by a Turing machine, it is quite another to know the Turing machine will deliver its solution in an acceptable amount of time. Operations research pioneer Merrill Flood [33] makes this point in his 1956 paper concerning the TSP.

> It seems very likely that quite a different approach from any yet used may be required for successful treatment of the problem. In fact, there may well be no general method for treating the problem and impossibility results would also be valuable.

His call for impossibility results is a direct statement that finite is not good enough. This issue was addressed in the early 1960s, led by the work and steady campaigning of Jack Edmonds. The better-than-finite notion Edmonds proposed is known as *polynomial-time complexity*. Formally, a *polynomial-time algorithm* is a Turing machine such that if n is the number of symbols on the input tape, then the machine is guaranteed to halt after a number of steps that is at most $c \cdot n^k + d$, for some constants k, c, and d. In this definition we could replace the Turing machine by a powerful modern digital computer without altering the concept. Indeed, the simulation of a modern computer via a Turing machine will slow down computations, but the slow-down factor is only polynomial in n.

A quick calculation of n^3 versus 2^n for several values of n makes it clear why Edmonds [30] referred to polynomial-time algorithms as "good algorithms."

> For practical purposes computational details are vital. However, my purpose is only to show as attractively as I can that there is an efficient algorithm. According to the dictionary, "efficient" means "adequate in operation or performance." This is roughly the meaning I want—in the sense that it conceivable for maximum matching to have no efficient algorithm. Perhaps a better word in "good."

The polynomial-time notion provides researchers a clear target when approaching a new class of problems. Edmonds notes that computational details are vital in practice and it is indeed true that not all polynomial-time algorithms are suitable for implementation. Nonetheless, polynomial-time complexity has been amazingly successful in generating methods that are not just good in theory, but good in practice as well.

BIG-OH NOTATION

Edmonds divides the world of algorithms into good and bad, but it is also useful to distinguish among polynomial-time algorithms for a given problem. The notion adopted in the complexity community is based on asymptotic performance. By this standard, an algorithm guaranteed to run in no more than $100n^2$ steps is preferable to one with a running-time guarantee of $10n^3$, since for large enough n we have $100n^2 < 10n^3$. This concept is captured by "big-oh" notation: given two positive real-valued functions on the set of nonnegative integers, $f(n)$ and $g(n)$, we say $f(n)$ is $\mathrm{O}(g(n))$ if there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all large enough values of n. An algorithm is said to be $\mathrm{O}(g(n))$ if it comes with a running-time guarantee that is $\mathrm{O}(g(n))$. Thus we write that a matching algorithm is $\mathrm{O}(n^4)$ or a sorting algorithm is $\mathrm{O}(n \log n)$ and do not worry about the constant factors in the running-time guarantees.

Like the concept of polynomial-time complexity, the use of big-oh notation does not always rank correctly the practical performance of two algorithms. Indeed, there are well-known examples of algorithms having nice $\mathrm{O}(n^3)$ guarantees that are impossible to implement due to enormous constants hidden by the notation. But once again, this rough classification provides natural targets to researchers, and the race to obtain the best big-oh guarantees has led to many of the fundamental algorithmic techniques described in the next two chapters.

NON-DETERMINISTIC POLYNOMIAL TIME: THE CLASS $\mathcal{N}P$

Of particular importance in complexity theory are problems that have yes or no answers. For such decision problems Richard Karp [53] introduced the short notation $\mathcal{P}$ to denote those that have polynomial-time algorithms. Optimization problems can be cast in this form by asking for a solution of a specified quality, for example, is there a TSP tour of length less than 1,000 miles?

The class $\mathcal{P}$ is the gold standard for decision problems, but Stephen Cook [26] studied a possibly larger class that arises in a natural way. Adopting a notion of Edmonds, Cook considered problems such that yes answers can be verified in polynomial time. To verify an answer, a certificate is provided together with the statement of the problem instance, allowing a Turing machine to check that the answer is indeed yes. For example, to verify that a set of cities can be visited in less than 1,000 miles we can provide the machine with such a tour.

An alternative view of verifications is via *nondeterministic Turing machines*. Such "machines" are not part of the physical world, since they have the capability of duplicating themselves during a computation. If there is a polynomial-time verification, then a nondeterministic machine can guess the correct certificate in one of its many copies and determine that the answer to the problem is yes. This view led Karp to propose the shorthand $\mathcal{N}P$ for Cook's class of problems.

On the surface, it would appear to be much easier to be a member of $\mathcal{N}P$ than to be a member of $\mathcal{P}$. Indeed, the TSP is a case where checking a solution is easy, but finding the solution may be difficult. Many more examples can be constructed, but these only hint that $\mathcal{N}P$ is a larger class than $\mathcal{P}$. It is currently not known if there is a problem in $\mathcal{N}P$ that is definitely not in $\mathcal{P}$.

$\mathcal{NP}$-COMPLETE PROBLEMS

In the paper that began the formal study of $\mathcal{NP}$, Stephen Cook put forth a certain problem in logic as a candidate for a member that may itself not be in $\mathcal{P}$. His problem, commonly known as the *satisfiability problem*, is to determine whether or not true and false values can be assigned to a collection of logical variables so as to make a given formula evaluate to true. The components of the formula are the variables and their negations, joined up by logical *and*'s and logical *or*'s. More important than the problem itself is Cook's reason for making the conjecture: his theorems show that every problem in $\mathcal{NP}$ can be formulated as a satisfiability problem.

The key component of Cook's theory is the idea of reducing one problem to another. Formally, a *problem reduction* is defined as a polynomial-time Turing machine that takes any instance of problem A and creates an instance of problem B, such that the answers to A and B are the same, either both yes or both no. It is clear that reductions are useful in sorting out the many members of $\mathcal{NP}$. To show a problem is easy, you can try to reduce it to another easy problem. To show a problem is hard, you can try to reduce a known hard problem to your problem. But the amount of order provided by reductions is surprising. Indeed, Cook proved that every problem in $\mathcal{NP}$ can be reduced to the satisfiability problem. A problem reduction from A to B implies that if B is in $\mathcal{P}$, then so is A. Thus, if satisfiability is in $\mathcal{P}$, then there exist polynomial-time algorithms for every problem in $\mathcal{NP}$. Cook thought it unlikely that $\mathcal{P} = \mathcal{NP}$, hence his conjecture that there does not exist a polynomial-time algorithm for the satisfiability problem.

An $\mathcal{NP}$ problem is called $\mathcal{NP}$*-complete* if every member of $\mathcal{NP}$ can be reduced to it. Cook followed his proof that satisfiability is $\mathcal{NP}$-complete with a quick argument that a graph-theory problem known as subgraph isomorphism is also $\mathcal{NP}$-complete: he showed that satisfiability can be reduced to subgraph isomorphism. So, any member of $\mathcal{NP}$ can be first reduced to satisfiability and then reduced to subgraph isomorphism. Building a single Turing machine to carry out both problem reductions, one after the other, shows that subgraph isomorphism is $\mathcal{NP}$-complete.

This idea of chaining together problem reductions created an explosion of interest in complexity theory, led by Karp's research paper [53], written one year after the announcement of Cook's results. Karp's paper presents a now famous list of twenty-one $\mathcal{NP}$-complete problems, including two versions of the TSP, together with their reductions from Cook's satisfiability problem. The list of twenty-one has since grown to a catalog of many of hundreds of $\mathcal{NP}$-complete problems. Indeed, it is an unfortunate fact that most problems that need to be solved in applied settings turn out to be $\mathcal{NP}$-complete.

$\mathcal{P}$ VERSUS $\mathcal{NP}$

The working hypothesis is that there can be no polynomial-time algorithm for an $\mathcal{NP}$-complete problem, but there is no compelling evidence that $\mathcal{P}$ and $\mathcal{NP}$ are actually distinct. Indeed, the $\mathcal{P}$ versus $\mathcal{NP}$ question is perhaps the most prominent open problem in all of mathematics and a $1,000,000 prize has been offered by the Clay Mathematics Institute. But even if $\mathcal{P} \neq \mathcal{NP}$ we are still faced with the task of finding solutions to specific instances of $\mathcal{N}P$-complete problems. It is important to bear in mind that computational complexity concerns the worst-case asymptotic behavior of algorithms. In particular, a problem being $\mathcal{N}P$-complete does not mean that any specific instance is impossible to solve, by hook or by crook.

1.4 EXERCISES

1. Modify the code in Appendix A.1 to include the `cheapsum` argument in the function `permute()`, as described in Section 1.1. Extra: As an alternative, develop a pruning procedure based on the fact that each city not yet assigned in a partial tour must eventually be the neighbor of two cities in a full tour.

2. Modify the code in Appendix A.1 to consider only tours such that city 0 comes before city 1 as we work our way back from the final city in the tour.

3. Applications of the TSP that involve a person traveling from place to place often include time restrictions on when the person may arrive at each given city, that is, for each city i there is a window $[a_i, b_i]$ such that the visit to city i must occur no earlier than time a_i and no later than time b_i. In this version, we have as input a specified starting city and the time to travel between each pair of cities. The goal is to minimize the total travel time of a tour that visits each city in its specified time window. The model is known as the TSP with time windows, or TSPTW. Design and implement an enumeration algorithm to solve small instances of the TSPTW.

4. In the pure Euclidean version of the TSP an instance is specified by coordinates (x_i, y_i) for each city i, and the travel cost between city i and city j is the straight-line distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. The implementation of the simple enumeration algorithm presented in Section 1.1 is designed for integer-valued travel costs and thus cannot be applied directly to solve such Euclidean instances. Using an arithmetic package such as the GNU Multiple Precision Arithmetic Library (GMP), modify the TSP implementation to solve Euclidean instances, assuming that the input coordinates (x_i, y_i) are integer valued.

5. Continuing Exercise 4, in the Euclidean TSP it is easy to see that there exists always an optimal solution such that the tour does not cross itself. Use this fact as an additional pruning rule in the enumeration algorithm. Does this improve the running time of your implementation?

6. Modern computers come equipped with multiple-core processors that can be utilized in TSP enumeration. Develop a parallel implementation of the enumeration algorithm and compare the speed-ups obtained when varying the number of available processor cores. Extra: Implement the algorithm on a graphics processing unit (GPU).

7. Starting with the tiny-TSP code given in Appendix A.2, implement a version of the Held-Karp δ-transformation to pre-process travel costs to improve the performance of the enumeration algorithm.

8. In the TSP enumeration process, for a subset $S \subset \{0, \ldots, n-2\}$ and a selected city $i \in S$, among the tours that begin with a path P from city $n-1$, through S, and ending at city i, we need only consider those where P has the shortest length among all such paths. Use this observation by pre-computing all such shortest paths where S contains only two cities and using these paths to initialize the input to the `permute()` function. Extra: Design your code to handle sets S of cardinality k, for k larger than 2. What is the impact on the total running time as k is increased?

9. Design and implement an enumeration algorithm to solve small instances of the perfect-matching problem. In this case, take as input a list of integer weights, one for each unordered pair of points.

10. Design a Turing machine that can add two non-negative integers given in binary notation. Extra: Also design a machine to multiply two non-negative integers.

1.5 NOTES AND REFERENCES

The field of discrete optimization, also known as *combinatorial optimization*, has advanced rapidly over the past fifty years, reaching a milestone in 2003 with the publication of Alexander Schrijver's three-volume monograph *Combinatorial Optimization: Polyhedra and Efficiency* [75], totaling 1,881 pages and including over 4,000 references. Schrijver's beautiful scholarly writing has defined the field, giving combinatorial optimization an equal footing with much older, more established areas of applied mathematics. His books are highly recommended, as are the texts by András Frank [35] and Bernhard Korte and Jens Vygen [58].

SECTION 1.1

A great general reference for the C programming language is the classic book by Brian Kernighan and Dennis Ritchie [56]. Their presentation contains beautiful examples, great explanations, and complete coverage. There is no need to go any further if you want to program in C, but if you wish to become an expert in the language we recommend the reference manual by Samuel Harbison and Guy Steele [42].

Our presentation of the enumeration algorithm for tiny TSP instances follows Jon Bentley's paper [10]. Further discussions of methods for tiny instances can be found in Chapters 7 and 8, along with much more material on the TSP. For an informal look at the problem, we recommend, subject to the bias of one of our authors, the book by William Cook [25].

The connection between spanning trees and TSP was known in the 1950s. In fact, Kruskal's research paper [59] was titled "On the shortest spanning subtree of a graph and the traveling salesman problem." The idea of visualizing δ transformations for

Euclidean instances is presented in a paper by Michael Jünger and William Pulleyblank [51].

The 48-city United States instance solved by Held and Karp in 1971 was first described in their 1962 paper [43]. The 57-city United States instance was constructed by Robert Karg and Gerald Thompson [52] with travel distances taken from a 1962 Rand-McNally road atlas.

SECTION 1.2

Bentley's books *Programming Pearls* [12] and *More Programming Pearls* [11] are a good source for success stories in the art of implementation.

SECTION 1.3

The classic reference for complexity theory is Michael Garey and David Johnson's book *Computers and Intractability: A Guide to the Theory of NP-Completeness* [38]; we recommend a thorough reading, again subject to the bias of one of our authors. An excellent recent treatment is *Computational Complexity: A Modern Approach* by Sanjeev Arora and Boaz Barak [3].

Appendix A

Sample Codes

A.1 TSP ENUMERATION

```
#include <stdio.h>

#define maxN 20
#define MAXCOST 1000000

int tour[maxN], besttour[maxN], distmatrix[maxN][maxN];
int ncount, bestlen = maxN * MAXCOST;

int main (int ac, char **av);
void dist_read (char *in);
int dist(int i, int j);
int tour_length();
void tour_swap(int i, int j);
void permute(int k);

int main (int ac, char **av)
{
    int i;
    if (ac != 2) {
        printf ("Usage: %s distance_table\n", *av); return 1;
    }

    dist_read (av[1]);
    for (i = 0; i < ncount; i++) tour[i] = i;
    permute(ncount-1);
    printf ("Optimal Tour Length: %d\n", bestlen);
    printf ("Optimal Tour: ");
    for (i = 0; i < ncount; i++) printf ("%d ", besttour[i]);
    printf ("\n");
    return 0;
}

void dist_read (char *in)
{
    FILE *fin = fopen (in, "r");
    int i, j, k;

    fscanf (fin, "%d", &ncount);
    for (i = 0; i < ncount; i++) {
        for (j = 0; j <= i; j++) {
            fscanf (fin, "%d", &k);
            distmatrix[i][j] = distmatrix[j][i] = k;
        }
    }
}

int dist(int i, int j)
{
    return distmatrix[i][j];
}

int tour_length()
{
    int i, len = 0;

    for (i = 1; i < ncount; i++) {
        len += dist(tour[i-1],tour[i]);
    }
    return len+dist(tour[ncount-1],tour[0]);
}

void tour_swap(int i, int j)
{
    int temp;
    temp = tour[i]; tour[i] = tour[j]; tour[j] = temp;
}

void permute(int k)
{
    int i, len;

    if (k == 1) {
        len = tour_length();
        if (len < bestlen) {
            bestlen = len;
            for (i = 0; i < ncount; i++) besttour[i] = tour[i];
        }
    } else {
        for (i = 0; i < k; i++) {
            tour_swap(i, k-1);
            permute(k-1);
            tour_swap(i,k-1);
        }
    }
}
```

A.2 TSP ENUMERATION WITH PRUNING

We skip the listings of the functions `dist()`, `dist_read()`, and `tour_swap()` that are identical to those given in Appendix A.1.

```
#include <stdio.h>

#define maxN 50
#define MAXCOST 1000000

int tour[maxN], besttour[maxN];
int datx[maxN], daty[maxN], distmatrix[maxN][maxN];
int ncount, bestlen = maxN * MAXCOST;

int main (int ac, char **av);
void permute(int k, int tourlen);
int mst(int count);
int nntour();

int main (int ac, char **av)
{
    int i;
    if (ac != 2) {
        printf ("Usage: %s distance_matrix\n", *av); return 1;
    }

    dist_read(av[1]);
    bestlen = nntour();
    for (i = 0; i < ncount; i++) besttour[i] = tour[i];
    for (i = 0; i < ncount; i++) tour[i] = i;
    permute(ncount-1,0);

    printf ("Optimal Tour Length = %d\n", bestlen);
    printf ("Optimal Tour: ");
    for (i = 0; i < ncount; i++) printf ("%d ", besttour[i]);
    printf ("\n");
    return 0;
}

void permute(int k, int tourlen)
{
    int i;
    if (tourlen+mst(k+1) >= bestlen) return;

    if (k == 1) {
        tourlen += (dist(tour[0],tour[1]) + dist(tour[ncount-1],tour[0]));
        if (tourlen < bestlen) {
            bestlen = tourlen;
            for (i = 0; i < ncount; i++) besttour[i] = tour[i];
        }
    } else {
        for (i = 0; i < k; i++) {
            tour_swap(i,k-1);
            permute(k-1, tourlen+dist(tour[k-1],tour[k]));
            tour_swap(i,k-1);
        }
    }
}

int mst(int count)  /* Adopted from Bentley, Unix Review 1996 */
{
    int i, m, mini, newcity, mindist, thisdist, len = 0;
    int pcity[maxN], pdist[maxN];
    if (count <= 1) return 0;

    for (i = 0; i < count; i++) {
        pcity[i] = tour[i];  pdist[i] = MAXCOST;
    }
    if (count != ncount) pcity[count++] = tour[ncount-1];

    newcity = pcity[count-1];
    for (m = count-1; m > 0; m--) {
        mindist = MAXCOST;
        for (i = 0; i < m; i++) {
            thisdist = dist(pcity[i],newcity);
            if (thisdist < pdist[i]) pdist[i] = thisdist;
            if (pdist[i] < mindist) { mindist = pdist[i]; mini = i; }
        }
        newcity = pcity[mini];
        len += mindist;
        pcity[mini] = pcity[m-1];  pdist[mini] = pdist[m-1];
    }
    return len;
}

int nntour()
{
    int i, j, best, bestj, len = 0;

    for (i = 0; i < ncount; i++) tour[i] = i;
    for (i = 1; i < ncount; i++) {
        best = MAXCOST;
        for (j = i; j < ncount; j++) {
            if (dist(tour[i-1],tour[j]) < best) {
                best = dist(tour[i-1],tour[j]); bestj = j;
            }
        }
        len += best; tour_swap(i, bestj);
    }
    return len+dist(tour[ncount-1],tour[0]);
}
```

Bibliography

[1] Ali, A. I., H.-S. Han, 1998. Computational implementation of Fujishige's graph realizability algorithm. European Journal of Operational Research **1**08, 452–463. doi:10.1016/S0377-2217(97)00167-7.

[2] Applegate, D. L., R. E. Bixby, V. Chvátal, W. Cook. 2006. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, New Jersey, USA.

[3] Arora, S., B. Barak. 2009. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, USA.

[4] Bartz-Beielstein, T., M. Chiarandini, L. Paquete, M. Preuss, eds. 2010. *Experimental Methods for the Analysis of Optimization Algorithms*. Springer, Berlin, Germany.

[5] Bellman, R. 1957. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, USA.

[6] Bellman, R. 1960. Combinatorial processes and dynamic programming. R. Bellman, M. Hall, Jr., eds. *Combinatorial Analysis*. American Mathematical Society, Providence, Rhode Island, USA. 217–249.

[7] Bellman, R. 1961. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, Princeton, New Jersey, USA.

[8] Bellman, R. 1962. Dynamic programming treatment of the travelling salesman problem. Journal of the Association for Computing Machinery **9**, 61–63.

[9] Bellman, R. E., S. E. Dreyfus. 1962. *Applied Dynamic Programming*. Princeton University Press, Princeton, New Jersey, USA.

[10] Bentley, J. L. 1997. Faster and faster and faster yet. Unix Review **1**5, 59–67.

[11] Bentley, J. 1988. *More Programming Pearls*. Addison-Wesley, Reading, Massachusetts, USA.

[12] Bentley, J. 2000. *Programming Pearls (2nd Edition)*. Addison-Wesley, Reading, Massachusetts, USA.

[13] Berge, C. 1961. Färbung von Graphen, deren sämtliche bzw. deren ungerade Kreise starr sind (Zusammenfassung). Wiss. Z. Martin-Luther-Univ. Halle-Wittenberg Math.-Natur. Reihe **1**0, 114.

[14] Berge, C. 1970. Sur certains hypergraphes généralisant les graphes bipartis. P. Erdős, A. Rẽnyi, V. Sós, eds. *Combinatorial Theory and its Applications I.* Colloq. Math. Soc. János Bolyai, Vol. 4. North-Holland. 119–133.

[15] Berge, C. 1972. Balanced matrices. Mathematical Programming **2**, 19–31.

[16] Bixby, R. E., W. H. Cunningham. 1995. Matroid optimization and algorithms. In: R. L. Grapham, M. Grötschel, L. Lovász, eds. *Handbook of Combinatorics, Volume 1*. North-Holland. 551–609.

[17] Bixby, R. E., D. K. Wagner. 1988. An almost linear-time algorithm for graph realization. Mathematics of Operations Research **1**3, 99–123.

[18] Cargill, T. 1992. *C++ Programming Style.* Addison-Wesley, Reading, Massachusetts, USA.

[19] Chudnovsky, M., G. Cornuéjuls, X. Liu, P. Seymour, K. Vuškovic. 2005. Recognizing Berge graphs. Combinatorica **2**5, 143–186. doi:10.1007/s00493-005-0012-8.

[20] Chudnovsky, M., N. Robertson, P. Seymour, R. Thomas. 2006. The strong perfect graph theorem. Annals of Mathematics **1**64, 51–229.

[21] Chvátal, V. 1975. On certain polyhedra associated with graphs. Journal of Combinatorial Theory, Series B **1**8, 138–154. doi:10.1016/0095-8956(75)90041-6.

[22] Clay Mathematics Institute. 2000. Millennium problems. `http://www.claymath.org/millennium/`.

[23] Conforti, M., G. Cornuéjols, A. Kapoor, K. Vuškovic. 2001. Balanced $0, \pm 1$ matrics II. Recognition algorithm. Journal of Combinatorial Theory, Series B **8**1, 275–306. doi:10.1006/jctb.2000.2011.

[24] Conforti, M., G. Cornuéjols, M. R. Rao. 1999. Decomposition of balanced matrices. Journal of Combinatorial Theory, Series B **7**7, 292–406. doi:10.1006/jctb.1999.1932.

[25] Cook, W. J. 2012. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, Princeton, New Jersey, USA.

[26] Cook, S. A. 1971. The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*. ACM Press, New York, USA. 151–158.

[27] Dantzig, G., R. Fulkerson, S. Johnson. 1954. Solution of a large-scale traveling-salesman problem. Operations Research **2**, 393–410.

[28] Dantzig, G. B. 1963. *Linear Programming and Extensions*. Princeton University Press, Princeton, New Jersey, USA.

[29] Ding, G., L. Feng, W. Zang. 2008. The complexity of recognizing linear systems with certain integrality properties. Mathematical Programming **1**14, 321–334.

[30] Edmonds, J. 1965. Paths, trees, and flowers. Canadian Journal of Mathematics **1**7, 449–467.

[31] Edmonds, J. 1991. A glimpse of heaven. J. K. Lenstra et al., eds. *History of Mathematical Programming—A Collection of Personal Reminiscences*. North-Holland. 32–54.

[32] Edmonds, J., R. Giles. 1977. A min-max relation for submodular functions on a graph. P. L. Hammer, E. L. Johnson, B. H. Korte, G. L. Nemhauser, eds. *Studies in Integer Programming*. Annals of Discrete Mathematics **1**. North-Holland. 185–204.

[33] Flood, M. M. 1956. The traveling-salesman problem. Operations Research **4**, 61–75.

[34] Fujishige, S. 1980. An efficient PQ-graph algorithm for solving the graph-realization problem. Journal of Computer and System Sciences **2**1, 63–86. doi:10.1016/0022-0000(80)90042-2.

[35] Frank, A. 2011. *Connections in Combinatorial Optimization*. Oxford University Press, Oxford, United Kingdom.

[36] Fulkerson, D. R. 1972. Anti-blocking polyhedra. Journal of Combinatorial Theory, Series B **1**2, 50–71. doi:10.1016/0095-8956(72)90032-9.

[37] Fulkerson, D. R., A. J. Hoffman, R. Oppenheim. 1974. On balanced matrices. Mathematical Programming Study **1**, 120–132.

[38] Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, California, USA.

[39] Gomory, R. E. 1958. Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society **6**4, 275–278.

[40] Gilmore, P. C., R. E. Gomory. 1961. A linear programming approach to the cutting-stock problem. Operations Research **9**, 849–859.

[41] Grötschel, M. 1980. On the symmetric travelling salesman problem: Solution of a 120-city problem. Mathematical Programming Study **1**2, 61–77.

[42] Harbison, S. P., G. L. Steele. 2002. *C: A Reference Manual (5th Edition)*. Prentice Hall, Englewood Cliffs, New Jersey, USA.

[43] Held, M., R. M. Karp. 1962. A dynamic programming approach to sequencing problems. Journal of the Society of Industrial and Applied Mathematics **1**0, 196–210.

[44] Held, M., R. M. Karp. 1970. The traveling-salesman problem and minimum spanning trees. Operations Research **1**8, 1138–1162.

[45] Held, M., R. M. Karp. 1971. The traveling-salesman problem and minimum spanning trees: Part II. Mathematical Programming **1**, 6–25.

[46] Hilbert, D. 1902. Mathematical problems, lecture delivered before the International Congress of Mathematicians at Paris in 1900. Bulletin of the American Mathematical Society **8**, 437–479. Translated from German by Dr. Mary Winston Newson.

[47] Hoffman, A. J. 1974. A generalization of max flow-min cut. Mathematical Programming **6**, 352–359.

[48] Hoffman, A. J., J. B. Kruskal. 1956. Integral boundary points of convex polyhedra. H. W. Kuhn, A. W. Tucker, eds. *Linear Inequalities and Related Systems*. Princeton University Press, Princeton, New Jersey, USA. 223–246.

[49] Hooker, J. N. 1994. Needed: an empirical science of algorithms. Operations Research **4**2, 201–212.

[50] Johnson, D. S. 2002. A theoretician's guide to the experimental analysis of algorithms. In: M. Goldwasser, D. S. Johnson, C. C. McGeoch, eds. *Data Structures, Near Neighbor Searches, and Methodology: Proceedings of the Fifth and Sixth DIMACS Implementation Challenges*. American Mathematical Society, Providence, Rhode Island, USA. 215–250.

[51] Jünger, M., W. R. Pulleyblank. 1993. Geometric duality and combinatorial optimization. S. D. Chatterji, B. Fuchssteiner, U. Kluish, R. Liedl, eds. *Jahrbuck Überblicke Mathematik*. Vieweg, Brunschweig/Wiesbaden, Germany. 1–24.

[52] Karg, R. L., G. L. Thompson. 1964. A heuristic approach to solving travelling salesman problems. Management Science **1**0, 225–248.

[53] Karp, R. M. 1972. Reducibility among combinatorial problems. In: R. E. Miller, J. W. Thatcher, eds. *Complexity of Computer Computations*. IBM Research Symposia Series. Plenum Press, New York, USA. 85–103.

[54] Karp, R. M. 1986. Combinatorics, complexity, and randomness. Communications of the ACM **2**9, 98–109.

[55] Kernighan, B. W., P. J. Plauger. 1974. *The Elements of Programming Style*. McGraw-Hill, New York, USA.

[56] Kernighan, B. W., D. M. Ritchie. 1978. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, USA.

[57] Knuth, D. E. 2011. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1*. Addison-Wesley, Upper Saddle River, New Jersey, USA.

[58] Korte, B., J. Vygen. *Combinatorial Optimization: Theory and Applications*, Fourth Edition. Springer, Berlin, Germany.

[59] Kruskal, J. B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society **7**, 48–50.

[60] Lawler, E. L., J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys, eds. 1985. *The Traveling Salesman Problem*. John Wiley & Sons, Chichester, UK.

[61] Lin, S. 1965. Computer solutions of the traveling salesman problem. The Bell System Technical Journal **4**4, 2245–2269.

[62] Lin, S., B. W. Kernighan. 1973. An effective heuristic algorithm for the traveling-salesman problem. Operations Research **2**1, 498–516.

[63] Little, J. D. C., K.G. Murty, D.W. Sweeney, C. Karel. 1963. An algorithm for the traveling salesman problem. Operations Research **1**1, 972–989.

[64] Lovász, L. 1972. A characterization of perfect graphs. Journal of Combinatorial Theory, Series B **1**3, 95–98. doi:10.1016/0095-8956(72)90045-7.

[65] Müller-Hannemann, M., A. Schwartz. 1999. Implementing weighted b-matching algorithms: towards a flexible software design. Journal of Experimental Algorithms **4**. doi:10.1145/347792.347815.

[66] Nemhauser, G. L. 1966. *Introduction to Dynamic Programming*. John Wiley & Sons, New York, USA.

[67] Nešetřil, J. 1993. Mathematics and art. In: *From the Logical Point of View 2,2*. Philosophical Institute of the Czech Academy of Sciences, Prague.

[68] von Neumann, J. 1947. The Mathematician. In: *Works of the Mind*, Volume 1, Number 1. University of Chicago Press, Chicago, Illinois, USA. 180–196.

[69] von Neumann, J. 1958. *The Computer and the Brain*. Yale University Press. New Haven, Connecticut, USA.

[70] Orchard-Hays, W. 1958. Evolution of linear programming computing techniques. Management Science **4**, 183–190.

[71] Orchard-Hays, W. 1968. *Advanced Linear-Programming Computing Techniques*. McGraw-Hill, New York, USA.

[72] Pferschy, U. 1999. Dynamic programing revisited: improving knapsack algorithms. Computing **6**3, 419–430.

[73] Pisinger, D. 1997. A minimal algorithm for the 0-1 knapsack problem. Operations Research **4**5, 758–767.

[74] Robertson, N., P. Seymour. 2004. Graph minors. XX. Wagner's conjecture. Journal of Combinatorial Theory, Series B **9**2, 325–357.

[75] Schrijver, A. 2003. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, Berlin, Germany,

[76] Seymour, P. D. 1980. Decomposition of regular matroids. Journal of Combinatorial Theory, Series B **2**8, 305–359. doi:10.1016/0095-8956(80)90075-1.

[77] Seymour, P. 2006. How the proof of the strong perfect graph conjecture was found. Gazette des Mathematiciens **1**09, 69–83.

[78] Tarjan, R. E. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia, Pennsylvania, USA.

[79] Truemper, K. 1990. A decomposition theory for matroids. V. Testing of matrix total unimodularity. Journal of Combinatorial Theory, Series B. **4**9, 241–281. doi:10.1016/0095-8956(90)90030-4.

[80] Walter, M., K. Truemper, 2011. Impementation of a unimodularity test. In preparation. `http://www.utdallas.edu/~klaus/TUtest/index.html`.

[81] Wolfe, P., L. Cutler. 1963. Experiments in linear programming. In: R. L. Graves and P. Wolfe, eds. *Recent Advances in Mathematical Programming*. McGraw-Hill, New York, USA. 177–200.

[82] Woeginger, G. J. 2003. Exact algorithms for NP-hard problems: A survey. M. Jünger, G. Reinelt, G. Rinadli, eds. *Combinatorial Optimization—Eureka, You Shrink!* Lecture Notes in Computer Science **2**570. Springer, Heidelberg, Germany. 185–207.

[83] Zambelli, G. 2004. *On Perfect Graphs and Balanced Matrices*. Ph.D. Thesis. Tepper School of Business, Carnegie Mellon University.

[84] Zambelli, G. 2005. A polynomial recognition algorithm for balanced matrices. Journal of Combinatorial Theory, Series B **9**5, 49–67. doi:10.1016/j.jctb.2005.02.006.