A Practical Guide to Discrete Optimization

Dynamic Programming, 29 December 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 Seven

Dynamic Programming

So much for naive approaches to optimization problems. It must be realized that large-scale processes will require both electronic and mathematical resources for their solution.

Richard Bellman and Stuart Dreyfus, 1962.

Shortest paths, spanning trees, network flows, and matchings all serve as building blocks in the practice of discrete optimization. It is a delight when an applied problem comes down to the solution of one of these basic models, and a well-tuned computer code is given heroic status by workers in the field of play. Unfortunately, as Schrjiver [75] writes in the preface to *Combinatorial Optimization*, "most problems that come up in practice are $\mathcal{N}P$-complete or worse." On the bright side, $\mathcal{N}P$-complete does not mean unsolvable. Indeed, beginning with this chapter on dynamic-programming algorithms, we present practical tools for attacking specific instances of models that may in general be intractable.

7.1 MORE TINY TSPS

To get into things, let's return to the tiny-TSP code from Chapter 1. You will recall that the basic algorithm is to consider all possible traveling-salesman tours with a fixed city in the final position. The tours are created by extending a path from the final point until it either includes all other cities or until it can be pruned, that is, until we can be sure that the path is not part of an optimal solution.

The first version of the algorithm used the most simple pruning device: we keep track of the cost of the path we are creating and, assuming there are no negative travel costs, we prune the search whenever the cost of the path matches or exceeds the cost of a best-known tour. This enumeration strategy is sufficient to solve TSP instances with twenty cities or so, but to take it up a notch we brought in spanning trees to serve as a lower bound on the cost to complete a path into a tour. The idea is illustrated in Figures 7.1 and 7.2. In the drawings, x is the fixed city and the algorithm has extended a path through four cities ending at t; we let S denote the cities in the path and we let U denote the set of cities not yet visited. To complete the $x - t$ path into a tour we must

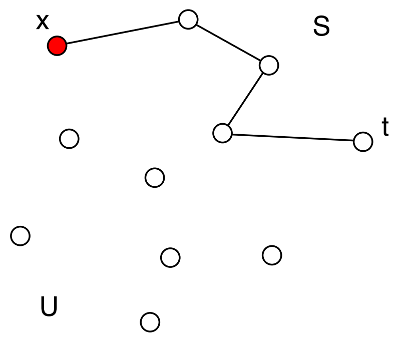

Figure 7.1: Cities S in path and cities U not yet in path.

build a path from t, through all cities in U, and ending back at x. As a lower bound on the cost of this additional $t - x$ path, we use the cost of a minimum spanning tree for the set $U \cup \{x, t\}$. Thus, the TSP search is pruned if the cost of the $x - t$ path plus the cost of the tree is at least as large as the cost of a best-known tour.

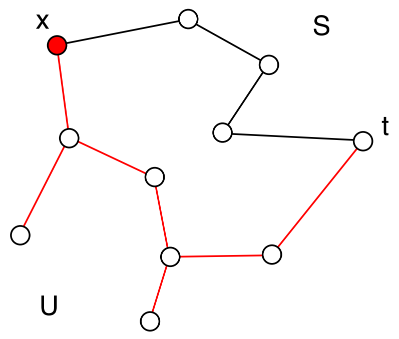

Figure 7.2: Lower bound from spanning tree of $U \cup \{x, t\}$ and path from x to t.

A major speed-up in the implementation of the algorithm was obtained by recording in a hash table the cost of each computed minimum spanning tree, indexed by the corresponding set S. When a set S is encountered a second time, the bound can be taken from the hash table rather than repeating the spanning-tree computation. The success of the approach suggests a potential further improvement that we did not explore in Chapter 1, namely, we can take advantage of the observation that during the running of the TSP algorithm we may encounter repeatedly the set S together with the same ending city t. Indeed, if we reach an $x - t$ path through S that is more costly than a previously encountered $x - t$ path through the same set S, then the search can be pruned. We can record the cost of the $x - t$ path in a hash table indexed by the pair

(S, t) and at each step of the algorithm check to see if our current $x - t$ path through S is the cheapest we have explored thus far.

To see how this path-pruning idea works in practice, we generated a set of ten 100-city random Euclidean instances to use as a test bed. In the table below, the results in the first line were obtained using our best Lin-Kernighan-equipped code from Chapter 1 and the results in the second line were obtained using the same code with path pruning; in both cases the running times are reported in seconds.

100-City Instances	1	2	3	4	5	6	7	8	9	10
No Path Pruning	6	3	97	86	23	3	91	171	5	2
Path Pruning	10	3	24	59	16	3	35	112	6	2

We see in the results a minor degradation on examples that are easy for the two codes to solve, but a significant speed-up on several of the more challenging instances.

PRECOMPUTING PATHS

The path-pruning principle is simple: in a search for an optimal tour, for any set S and $t \in S$, we need consider only the least-costly path among all those that start at x, run through S, and end at t. Our hash-table approach makes a lazy use of the principle, recording paths as they are encountered. In a more direct application, we can precompute, for some positive integer k, the cheapest path from x through S and ending at t for all k-element sets S and all $t \in S$. The full problem is then solved by carrying out the tiny-TSP search process starting from each of the precomputed paths.

To implement the precomputation, for each $i = 1, \ldots, k$, we build a hash table indexed by pairs (S, t) with $|S| = i$ and $t \in S$. At the ith level, we take each (S, t) entry from the level-$(i-1)$ table, attach each of the $n - i - 1$ possible cities q to obtain a path of length i ending at q, and, if the path cannot be pruned using the spanning-tree bound, we either add or update the $(S \cup \{q\}, q)$ entry in the level-i hash table. Thus, the collection of paths of length 1 are used to create the paths of length 2, which are used to create the paths of length 3, and so on up the line until we reach all non-pruned paths of length k. Such computations can require large amounts of storage for the path collections, so once we have the paths of length i we go ahead and delete the level-$(i-1)$ hash table.

In the tests reported below, we vary the value of k to consider precomputing paths of length 10, length 25, and the full 99-city-length paths.

100-City Instances	1	2	3	4	5	6	7	8	9	10
$k = 10$	17	3	21	66	15	3	39	102	6	2
$k = 25$	14	7	18	57	11	2	34	81	5	2
Full Paths	8	3	7	20	7	2	13	41	3	2

The extra overhead of creating the sequence of hash tables does not pay off when we consider only small values of k: some of the results are slightly better than those we obtained with the lazy implementation of path-pruning, but others are worse. Interestingly, if we go ahead and precompute the full-length paths, then we see a uniform improvement over our previous champion code. Many factors come into play, but the step-by-step creation of full sets of non-pruned paths appears to be a better way to organize the search for an optimal tour. We must remark, however, that the success of the method depends heavily on the Lin-Kernighan algorithm for obtaining a good starting tour to set the pruning value `bestlen`, since full tours are only constructed at the final stage of the search.

Although we are pushing the limits of the implementation, it is possible to solve somewhat larger random Euclidean TSP examples using full-path computations. Indeed, the following table reports results for ten 150-city instances.

150-City Instances	1	2	3	4	5	6	7	8	9	10
Full Paths	162	$**$	$**$	$**$	53	650	15	381	2238	27

The "**" entries indicate that the code exceeded a 12-GByte memory limit without solving the particular example. So on this test set we had seven successes and three failures. That is not bad for such large instances of the TSP. We should probably drop the use of the adjective "tiny" at the point!

BELLMAN-HELD-KARP ALGORITHM

The full-path implementation of our code, where we set $k = n-1$, is actually a souped-up variant of an old TSP algorithm discovered independently by Richard Bellman [6, 8] and Michael Held and Richard Karp [43] in the early 1960s. Their algorithm holds a spot in the Pantheon of the TSP, since its $O(n^2 2^n)$ running time for solving an n-city instance is the best asymptotic bound that has been achieved to date, despite great efforts over the past fifty years.

The Bellman-Held-Karp algorithm is based on the same level-by-level optimization we employed. Indeed, the heart of the method can be described by a simple recursive equation. To set this up, let x denote a fixed starting point for the tours and for any pair of cities (i, j) let $dist(i, j)$ denote the cost to travel from i to j. For any S such that $x \notin S$ and for any $t \in S$, let $opt(S, t)$ denote the minimum cost of a path starting x, running through all points in S, and ending at t. We have

$$opt(S, t) = \min(opt(S \setminus \{t\}, q) + dist(q, t) : q \in S \setminus \{t\}). \tag{7.1}$$

Moreover, if we let N denote the set of all cities other than x, then the optimal value of the TSP is

$$\nu^* = \min(opt(N, t) + dist(t, x) : t \in N). \tag{7.2}$$

Observe that for all $q \in N$ we have $opt(\{q\}, q) = dist(x, q)$. Starting with these values, the recursive equation (7.1) is used to build the values $opt(S, t)$ for all $S \subseteq N$ and $t \in S$, working our way through sets with two elements, then sets with three elements, and step by step up to the full set N. Once we have the values $opt(N, t)$ for all $t \in N$, we use (7.2) to find ν^*. Now, in a second pass, the optimal tour is computed by first identifying a city v_{n-1} such that $opt(N, v_{n-1}) + dist(v_{n-1}, x) = \nu^*$, then identifying a city $v_{n-2} \in N \setminus \{v_{n-1}\}$ such that $opt(N \setminus \{v_{n-1}\}, v_{n-2}) + dist(v_{n-1}, v_{n-2}) = opt(N, v_{n-1})$, and so on until we have v_1. The optimal tour is

$(x, v_1, \ldots, v_{n-1})$. This second pass is to permit the algorithm to store only the values $opt(S,t)$ and not the actual paths that determine these values.

The Bellman-Held-Karp algorithm can be implemented in just a few lines of C, making use of recursion to compute the $opt(S,t)$ values and storing them in a two-dimensional integer array `opt[][]` to avoid any recomputations. In this implementation, a set S is represented by `S`, either an unsigned integer or an unsigned long integer, where the 1-bits in `S` indicate the elements in S. To make this choice flexible, we use the following typedef in our code.

```
typedef unsigned int Set;
```

Unsigned integers can handle TSP instances with up to 33 cities and unsigned long integers can handle up to 65-city instances, although this later problem size is actually well beyond the capabilities of the Bellman-Held-Karp algorithm on current computing platforms.

Since there are few choices for t but many choices for S, we reverse the indices, storing $opt(S,t)$ in entry `opt[t][S]`. This allows us to dynamically allocate the array with a sequence of `malloc()` calls when solving large instances. For small examples, where `maxN` indicates the maximum number of cities, we can use the following static allocation.

```
int opt[maxN-1][1<<(maxN-1)];
```

The main routine initializes `opt` as follows.

```
int i
Set S, setcount = 1<<(n-1);
for (i=0; i<n-1; i++) {
   for (S=0; S<setcount; S++) opt[i][S] = -1;
   opt[i][1<<i] = dist(i,n-1);
}
```

This code fragment assigns correctly the values for single-city sets and it initializes all other entries to -1. Note that we are using city $n-1$ as the starting point x, so the first loop runs over the cities $i = 0, \ldots, n-2$.

The core of the algorithm is the following 15-line function to compute $opt(S,t)$.

```
int solve(Set S, int t)
{
   int i, v, minv = MAXCOST;
   Set R;
   if (opt[t][S] == -1) {
      R = S & ~(1<<t);                  // R = S\t
      for (i=0; i<n-1; i++) {
         if (!(R & (1<<i))) continue;   // Is i in R?
         v = solve(R,i) + dist(i,t);
         if (v < minv) minv = v;
      }
      opt[t][S] = minv;
   }
   return opt[t][S];
}
```

In the main routine, `solve()` is used to compute $opt(N, t)$ for all cities $t \in N = \{0, \ldots, n-2\}$; the optimal tour length is recorded in the variable `bestlen`.

```
int t, len, bestlen=MAXCOST;
Set N=(1<<(n-1))-1;
for (t=0; t<n-1; t++) {
   len = solve(N,t) + dist(t,n-1);
   if (len < bestlen) bestlen = len;
}
```

With the computed optimal tour length, we now run through the stored values of $opt(S, t)$ to gather the optimal tour itself, recording it in the integer array `tour`.

```
S=(1<<(n-1))-1;
tour[0]=n-1;
for (i=1; i<n; i++) {
   for (j=0; j<n-1; j++) {
      if ((S & (1<<j)) &&
                bestlen == opt[j,S]+dist(j,tour[i-1])) {
         bestlen -= dist(j,tour[i-1]);
         tour[i]=j;
         S &= ~(1<<j);
         break;
      }
   }
}
```

And that is it. A nice, compact computer code for the TSP. Of course, the running time is indeed proportional to $n^2 2^n$, so we will not be solving 150-city test instances. As we will see, however, the code is competitive on small instances when compared with simple versions of our tiny-TSP enumeration algorithms.

Before reporting computational tests, we mention one optimization to the code. In an algorithm of this type, a restricting factor is the large amount of memory needed to store the many values of $opt(S, t)$. Our implementation includes an array of 2^{n-1} integers for each city t. These integers represent all subsets of N, but we are only interested in those that contain the city t. We can therefore save half of our memory requirement by storing only the 2^{n-2} integers that we actually need. To implement this, we use the following small function to take a set represented by $n-1$ bits and compute an integer less than 2^{n-2} by removing the t'th bit and shifting one to the right all bits that are to the left of t.

```
unsigned int setnum (unsigned int S, int t)
{
    return (S & ((1<<t)-1)) | ((S & ~((2<<t)-1))>>1);
}
```

The function `solve()` is modified to use `setnum(S,t)` as the index into the array `opt[t][]`, rather than using `S` as the index. A small change, but saving half of the memory is worthwhile on test instances at the top of the range that can be handled with the Bellman-Held-Karp algorithm.

Returning to our random Euclidean test instances from Chapter 1, the following table reports running time in seconds for Bellman-Held-Karp on examples from ten cities up to twenty cities. For comparison, we have included on a second line the times for the tiny-TSP code that uses only the length of the current path as the pruning value.

Cities	10	11	12	13	14	15	16	17	18	19	20
B-H-K	.000	.000	.001	.003	.006	.015	.03	.08	.18	.8	2.4
Tiny TSP	.003	.01	.05	0.3	2	5	34	22	56	5325	646

The Bellman-Held-Karp code looks practical, but the $n^2 2^n$ running time and memory requirement overwhelms the implementation as we increase the number of cities.

Cities	21	22	23	24	25	26	27	28	29	30
Seconds	6	15	32	74	160	366	885	2175	$**$	$**$

When we reach $n = 29$, the memory requirement exceeds 12 GBytes and the test was terminated without computing an optimal tour.

It is instructive to examine the huge difference in the capabilities of our Bellman-Held-Karp code and the full-path tiny-TSP code we described above. The full-path code solves the 29-city and 30-city examples in under one-tenth of a second, and it solves routinely 100-city instances in under a minute. Yet the two algorithms are very similar in the way they build optimal paths through subsets of cities. The decisive factor is the use of a strong pruning device in the full-path code, aided by the Held-Karp cost transformation and the Lin-Kernighan initial tour, allowing us to avoid the computation and storage of the vast majority of the $opt(S,t)$ values.

SEQUENTIAL ACCESS TO SUBSETS

The recursive implementation of the Bellman-Held-Karp algorithm gives a compact code, but there is a penalty in time for the use of recursive function calls. A more direct implementation fills in the $opt(S,t)$ values by explicitly generating all subsets of size k, working from $k = 1$ up to $n - 1$. We describe two approaches to carrying out such a computation. The first is a straightforward change to the recursive algorithm, while the second rearranges the order in which the $opt(S,t)$ values are stored.

In the first version, we replace the recursive `solve()` function with three nested `for` loops, running through the values of k in increasing order, then through all subsets of S of cardinality k, and finally through all cities $t \in S$. To carry this out, we need a routine that takes a bit-representation of a set of cardinality k and returns the next set of the same cardinality. This sounds tough, but the following clever and short function, known as "Gosper's hack," does the trick.

```
Set nextset(Set S)
{
   Set U = S & -S;
   Set V = U + S;
   if (V == 0) return 0;
   return V + (((V^S)/U)>>2);
}
```

We leave it as an exercise to work out the bit manipulations that prove Gosper's hack does indeed run through all sets of cardinality k when called repeatedly, beginning with a bit representation of $\{0, 1, \ldots, k-1\}$. The triple loop can be written as follows, assuming we have already taken care of the single element sets S.

```
for (k=2; k<n; k++) {
   for (S = (1<<k)-1; S && S < ((Set)1<<(n-1));
        S = nextset(S)) {
      for (t=0; t<n-1; t++) {
         if (!(S & (1<<t))) continue;
         /* Compute opt[S,t] using values already */
         /* computed for subsets of size k-1.     */
      }
   }
}
```

This implementation makes clear the $n^2 2^n$ running time: the outer two loops run through $2^{n-1} - n$ sets, the third loop runs through n cities, and inside the third loop we have an $O(n)$ computation to compute $opt(S, t)$ using equation (7.1). The nested loops also make clear that there is unfortunately no chance for the algorithm to get lucky on any particular instance. For example, in a test on ten 25-city random Euclidean instances, the running times varied only between 51.28 seconds and 51.49 seconds. This is, however, a factor of three faster than results we obtained with the recursive implementation, and the following running times for the triple-loop code show that the speed-up holds also as we vary the number of cities.

Cities	21	22	23	24	25	26	27	28	29	30
Seconds	2	5	10	23	51	119	266	592	$**$	$**$

The code still fails to solve the $n = 29$ instance, after exceeding the 12 GBytes of memory available on our test machine. The behavior of the implementation is that once the code exceeds the physical memory, the operating system resorts to heavy paging, that is, blocks of memory are written to and read from the computer's disk drive, resulting in less than 1% usage of the central processing unit (CPU). This quick failure of the paging mechanism suggests that the memory access required by the implementation is very much non-sequential. Indeed, the values of $opt(S, t)$ are located at addresses that are easy to compute given the representations of S and t, but the locations of consecutive values needed in our triple loop are far apart in the physical memory. A significant improvement in the running time can be obtained by rearranging this storage.

In a new layout of the $opt(S, t)$ values, we place consecutively all pairs (S, t) for a fixed set S and varying $t \in S$. We will also place consecutively, for $k = 1$ up to $n - 1$, all blocks of (S, t) pairs having $|S| = k$. The values are stored in a one-dimensional array `V` of length $(n-1)2^{n-2}$. With this set up, it is convenient to have an array `b[ncount]` such that `b[k]` contains the starting point for the list of values corresponding to sets of size k.

```
b[0] = 0;
for (i=1; i<n; i++) {
   b[i] = b[i-1] + (i-1)*binomial[n-1][i-1];
}
```

In this code fragment, `binomial[m][k]` stores the binomial coefficient $\binom{m}{k}$, that is, the number of k-element subsets chosen from a set of size m; these values can be precomputed using the recurrence $\binom{m}{k} = \binom{m-1}{k-1} + \binom{m-1}{k}$.

The position of a pair (S, t) in our ordering will determine the set S and city t, so we do not need to record explicit representations of the set and city. In working with a particular set, however, we will use an integer array to hold its elements.

```
typedef int aSet[maxN];
```

To specify the ordering of the sets of size k, we use the following two functions.

```
void firstset(int k, aSet S)
{
   int i;
   for (i=0; i<k; i++) S[i] = i;
}
void nextset(int k, aSet S)
{
   int i;
   for (i=0; i<k-1 && S[i]+1 == S[i+1]; i++) S[i] = i;
   S[i] = S[i]+1;
}
```

The first function sets the starting set to $S = \{0, \ldots, k-1\}$ and the second function modifies a set S to obtain the next set in the ordering. For $n = 7$ and $k = 4$, the functions produce the following sequence, where the sets are ordered by column.

0123	0125	0245	0126	0246	0156	2356
0124	0135	1245	0136	1246	0256	0456
0134	0235	0345	0236	0346	1256	1456
0234	1235	1345	1236	1346	0356	2456
1234	0145	2345	0146	2346	1356	3456

The sequence of sets provides the orders for blocks of values (S, t) for a fixed S and varying $t \in S$. Within a block the values are ordered according to t's position in S, that is, `(S,S[0])`, `(S,S[1])`, up to `(S,S[k-1])`.

Now comes the tricky part. To use the stored values, we must be able to find the location of (S, t) in the array `V` given a set S of size k and the position of a city $t \in S$. This is handled by the following function, returning a pointer to $opt(S, t)$.

```
int *Sval (int k, Set S, int t_indx) {
   unsigned long int loc = 0;
   int i;
   for (i=0; i<k; i++) {
      loc += binomial[S[i]][i+1];
   }
   return &V[b[k] + k*loc + t_indx];
}
```

The `for` loop computes `loc`, the position of the set S within the list of sets of size k. For example, the set 1245 from the above list gets `loc` set to

$$\binom{1}{1} + \binom{2}{2} + \binom{4}{3} + \binom{5}{4} = 1 + 1 + 4 + 5 = 11$$

corresponding to its eleventh position in the list. (Remember that we count from zero.) Like Gosper's hack, it is a nice exercise to work out that `Sval` matches the ordering given by `nextset()`. Now, the starting position for the list of values for sets of size k is `b[k]`, so we obtain the position of (S, t) by adding `k*loc` to `b[k]`, since each set yields k values, and adding the index of t.

Equipped with functions to build the ordering of the pairs (S, t) and to find for a given (S, t) its location in the ordering, we can rewrite the triple loop as follows.

```
void build_V() {
   int t, j, k, minv, v, *valbase;
   aSet S, S_minus_t;

   for (firstset(1,S); S[0]<n-1; nextset(1,S)) {
      *Sval(1,S,0) = dist(S[0],n-1);
   }
   for (k=2; k<n; k++) {
      for (firstset(k,S); S[k-1]<n-1; nextset(k,S)) {
         for (t=1; t<k; t++) S_minus_t[t-1] = S[t];
         for (t=0; t<k; t++) {
            valbase = Sval(k-1,S_minus_t,0);
            minv = MAXCOST;
            for (j=0; j<k-1; j++) {
               v = valbase[j] + dist(S[t],S_minus_t[j]);
               if (v < minv) minv = v;
            }
```

```
                *Sval(k,S,t) = minv;
                S_minus_t[t] = S[t];
            }
        }
    }
}
```

It is important to note that the inner `for` loop is now a fast sequential scan through values of `V` starting at the position held by the variable `valbase`. This is a significant gain, and the results below show another speed-up of a factor four on our test instances.

Cities	21	22	23	24	25	26	27	28	29	30
Seconds	0.6	1	3	6	13	27	58	125	269	2296

The $n = 29$ instance solved successfully, despite the fact that the required memory exceeded that available on the computer; in this case the paging process was successful in bringing the needed data back and forth from the computer's disk drive. In fact, the code also solved the $n = 30$ instance on the same computer, although in this case the CPU utilization was often at only 25%, which explains the jump in the running time. On a larger machine, equipped with 128 GByte of physical memory, we obtained the following running times on instances with up to 32 cities.

Cities	29	30	31	32
Seconds w/ 128GB	290	617	1320	2840

This is still well behind our full-path tiny-TSP code, but the speed-up obtained by arranging the data for sequential access is a good example to keep in mind for general computational work.

7.2 THE PRINCIPLE OF OPTIMALITY

Richard Bellman's presentation [8] of the Bellman-Held-Karp algorithm for the TSP begins with the line "Consider the problem as a multistage decision process." This is a natural approach for the man who had spent much of the 1950s developing a general framework for handling problems that involve sequences of decisions. For the TSP, the sequence consists of the choices of cities to extend a path step by step, from the starting point until we reach all cities and return home.

Bellman's line-of-attack for the TSP and other multi-stage models is based on a *principle of optimality*, quoted from his classic book *Dynamic Programming* [5].

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

Bellman studies the optimality principle in a wide swath of applied mathematics: his book includes chapters on the calculus of variations, multi-stage games, inventory equations, bottleneck problems, continuous stochastic decision problems, and Markovian decision processes. Our use of his framework is rather more restrictive, concerning only deterministic problems and discrete decision processes.

In the application to the TSP, the optimality principle directly leads to the recursive equation (7.1) that is the heart of the Bellman-Held-Karp algorithm:

$$opt(S,t) = \min(opt(S \setminus \{t\}, q) + dist(q,t) : q \in S \setminus \{t\}).$$

Recall that $opt(S,t)$ denotes the minimum cost of a path starting at x, running through all cities in the subset S, and ending at $t \in S$. To line up the equation with Bellman's principle, from the definition of $opt(S,t)$ we have immediately

$$opt(S,t) = \min(cost(P) + dist(q,t) : q \in S \setminus \{t\}, P \in path(S \setminus \{t\}, q)),$$

where $path(R,y)$ denotes the set of all paths from x, through all cities in the subset R, and ending at $y \in R$. For each choice of q, the optimality principle allows us to replace the many $cost(P) + dist(q,t)$ terms by the single term $opt(S \setminus \{t\}, q) + dist(q,t)$; once the initial decision to travel from t to q is made, then we must select an optimal route through the remaining cities $S \setminus \{t\}$. Now from the simplified equation we obtain an algorithm, handling the subproblems $opt(S,t)$ one by one, starting with single-city sets S and working our way towards a full solution. The step-by-step process is called *dynamic-programming*.

In general, a dynamic-programming algorithm can be viewed as a traversal of an acyclic directed graph, where the nodes of the graph correspond to subproblems and the edges of the graph correspond to dependencies, that is, an edge from subproblem A to subproblem B means that to solve B we should have in hand a solution to A. In the case of the TSP, we have a node for each subproblem $opt(S,t)$ and edges directed from $opt(S \setminus \{t\}, q)$ to $opt(S,t)$ for each city $q \in S \setminus \{t\}$. To begin the algorithm, we compute solutions to the subproblems corresponding to nodes having no incoming edges—these are the single-city sets in the case of the TSP. Then in a general step we select and solve a subproblem A such that for each incoming edge we have already solved the subproblem corresponding to the edge's other end. Such a traversal of the graph is always possible; the order of the nodes we visit in the algorithm is called a *topological ordering* of the graph.

WHAT IS IN A NAME?

In the first two lines of the preface to his book, Bellman explains the use of the term dynamic programming.

> The purpose of this work is to provide an introduction to the mathematical theory of multi-stage decision processes. Since these constitute a somewhat formidable set of terms we have coined the term 'dynamic programming' to describe the subject matter.

Dynamic programming is not a specific model such as linear programming, and this has sometimes led to confusion. Concerning Bellman's terminology, George Nemhauser wrote the following in his book *Introduction to Dynamic Programming* [66].

> He invented the rather undescriptive but alluring name for the approach—*dynamic programming*. A more representative but less glamorous name would be *recursive optimization.*

Indeed, dynamic programming turns a recursive description of a process into a method to produce an optimal solution. The art of dynamic programming is to find a suitable recursion; one such that the implicit acyclic directed graph is not overwhelming large and such that it is feasible to compute and store solutions to individual subproblems.

7.3 KNAPSACK PROBLEMS

A clean and simple example of Bellman's dynamic-programming method is its use in solving instances of the *knapsack problem*. The snappy problem name is derived from the possible exercise of packing items into a knapsack with a restriction on the total weight that can be carried. Consider, for example, the six items illustrated as rectangles in Figure 7.3. The items have weights 3, 2, 4, 3, 1, and 5, respectively, indicated by

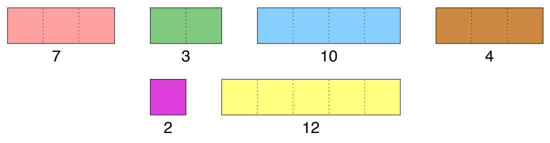

Figure 7.3: Six items to pack into a knapsack.

the length of each rectangle. Each item has an associated profit, that is, a measure of how much we desire to have it with us in our knapsack; the profits are 7, 3, 10, 4, 2, and 12, respectively, indicated by the values under each item. In Figure 7.4 we display

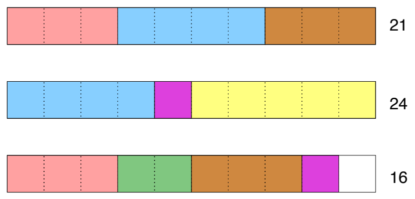

Figure 7.4: Three possible packings into a knapsack of capacity 10.

three possible ways to pack subsets of the six items into a knapsack that has weight capacity 10; the values next to each of the three packings indicates the total profit we obtain from the packed items. The knapsack problem is to find an allowable packing of maximum total profit.

A general instance of the problem is specified by a collection of n items with positive integer weights $w_0, \ldots, w_{n-1}$ and positive integer profits $p_0, \ldots, p_{n-1}$, together with an integer capacity c. Any subset of items such that the sum of their weights is not greater than c is a potential solution. The *value* of a solution is the sum of the profits of the selected items, and an *optimal packing* is one with maximum possible value. This model is also known as the *0-1 knapsack problem*, since for each item we have a 0-1 decision, either we include the item or we do not. Indeed, a packing of the knapsack can be represented by an n-dimensional 0-1 vector, with the ith component of the vector equal to one if the ith item is included in the solution and zero otherwise. For convenience, in the remainder of the section, we will say "item i" to refer to the ith item, where i is an integer between 0 and $n - 1$.

Two common variants of the knapsack problem are the *unbounded knapsack problem*, where any number of copies of each item may be placed in the knapsack, subject to the capacity of the knapsack not being exceeded, and the *bounded knapsack problem*, where a specified bound is given for each item, indicating the maximum number of copies of the item that can be placed into the knapsack.

This problem is $\mathcal{N}P$-hard, and therefore we do not expect there to be a solution algorithm that runs in polynomial-time, where the size of a problem instance is the number of bits required to represent the input data. Unlike many other $\mathcal{N}P$-hard problems, however, one can often solve very-large instances of the knapsack problem in reasonable time. Furthermore, unlike the TSP, there exists a fully polynomial-time approximation scheme for the knapsack problem: given any $\epsilon > 0$, there exists an algorithm which finds a solution of an instance of the knapsack problem with value at most $(1 + \epsilon)$ times the optimal solution value, and runs in time polynomial in the size of the input data and $1/\epsilon$.

There are a wide range of methods for attacking the knapsack problem, but some of the simplest are dynamic-programming algorithms, based on the following application of Bellman's principle of optimality. Given n items and a knapsack with capacity c, if we know the best way of packing knapsacks with capacity $1, \ldots, c$ using only the first $n - 1$ items, then it is trivial to figure out the best of way of packing a knapsack of capacity c with all n items. To see this, consider an optimal solution, and note that if it contains item $n - 1$, then the remaining items in the solution form an optimal packing of items $0, \ldots, n - 2$ into a knapsack of capacity $c - w_{n-1}$. If the optimal solution does not contain item $n - 1$, then the remaining items form an optimal packing of items $0, \ldots, n - 2$ into a knapsack of capacity c. Thus the optimal solution value can be determined from the optimal solution values of the two modified knapsack problems with $n - 1$ items.

Applying the above idea, we can write immediately a simple code to obtain the optimal solution of an instance of the knapsack problem. Assume the input data is contained in the following C structures.

```
int p[n], w[n], c;
```

Here `p[i]` contains p_i, the profit of the ith item, `w[i]` contains w_i, and `c` contains the capacity c. In a two-dimensional array of integers

```
int v[n][c+1];
```

we store in `v[i][j]`, for any $i \in \{0, \dots, n-1\}$ and $j \in \{0, \dots, c\}$, the optimal solution value of the knapsack problem with items $0, \dots, i$ and capacity j. Note that `v[n-1][c]` is the optimal value for the knapsack problem. The following function fills in the array `v`.

```
int kp_dp()
{
   int i, j;

   for (j=0; j<w[0]; j++)  v[0][j] = 0;
   for (j=w[0]; j<=c; j++) v[0][j] = p[0];

   for (i=1; i<n; i++){
      for (j=0; j<w[i]; j++) {
         v[i][j] = v[i-1][j];
      }
      for (j=w[i]; j<=c; j++) {
         v[i][j] = MAX(v[i-1][j], p[i]+v[i-1][j-w[i]]);
      }
   }
   return v[n-1][c];
}
```

The first two loops compute the optimal way of packing the first item in knapsacks of capacity $0, \dots, c$. The third loop fills in the values array by increasing number of items; the function `MAX(a,b)` returns the maximum of two numbers a and b. The running-time complexity of the algorithm is $O(nc)$; the space requirement is also $O(nc)$, since the values array contains $n(c+1)$ integers.

The entries `v[i][j]` correspond to *states* in standard dynamic-programming terminology; they represent complete solutions to partial problems, involving a subset of items and a portion of the capacity. The generated states for the example with six items in Figure 7.4 are given in the following table, where the rows correspond to items and the columns correspond to capacity values.

i/j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	7	7	7	7	7	7	7	7
1	0	0	3	7	7	10	10	10	10	10	10
2	0	0	3	7	10	10	13	17	17	20	20
3	0	0	3	7	10	10	13	17	17	20	21
4	0	2	3	7	10	12	13	17	19	20	22
5	0	2	3	7	10	12	14	17	19	22	24

Once we have the optimal solution value returned by `kp_dp()`, we can compute an optimal knapsack packing as follows. Note that if `v[n-1][c]` is equal to `v[n-2][c]` then there is an optimal solution consisting of a subset of the first $n-1$ items, and we can ignore item $n-1$. On the other hand, if the two values are different, then *every* optimal solution contains item $n-1$. The following code uses this idea to obtain an optimal solution, similar to how we grabbed an optimal TSP tour.

```
void kp_dp_sol(int *bestsol)
{
   int i, curcap = c;

   for (i=n-1; i>=1; i--) {
      if (v[i][curcap] == v[i-1][curcap]) {
         bestsol[i] = 0;
      } else {
         bestsol[i] = 1;
         curcap -= w[i];
      }
   }
   bestsol[0] = (curcap >= w[0]) ? 1 : 0;
}
```

To study the performance of `kp_dp()`, we consider two types of randomly generated instances: the *uncorrelated* instances and the *strongly correlated* instances. In the first class, given n and a range R, the weights and profits of items are chosen at random, independently, from the range 1 to R. We set the capacity to be half the sum of the weights. In the second class of instances, the weights are chosen in the same way, but the profit of an item is set to its weight plus a fixed constant, in our case 10.

In the table below we give the running time in seconds to solve uncorrelated instances for $n = 100, 1000, 1000$ and $R = 100, 1000, 10000$. A "**" indicates the dynamic-programming code failed to solve the problem within 10 minutes.

n/R	100	1000	10000
100	.004	.017	.144
1000	.148	1.35	12.80
10000	13.53	$**$	$**$

We solve only one instance for each choice of n and R, since when n is fixed the running time only varies with the capacity c, and the capacities are essentially the same for different random instances. Also, the capacity of uncorrelated and strongly correlated instances are essentially the same, and so `kp_dp()` takes nearly the same time on both classes of instances.

When $n = 10{,}000$ and $R = 100$, the instance we consider has $c \approx 250{,}000$. Thus the values array requires about 2.5 billion integers, or about 10 GBytes, assuming four bytes per integer. This is a lot of memory. Further, even on computers that have sufficient memory, the declaration `v[n][c+1]` may not work; in C such arrays use contiguous memory, and this large quantity of contiguous memory may not be available. We thus use `malloc()` to allocate space for `v` in our test implementation. Finally, when $n = 10{,}000$ and $R = 1000$, we need about 100 GBytes to store `v`, exceeding the quantity of random-access memory available on our test machine; in this case disk space is used to store the data and the program thus slows down considerably.

Faster computation of the knapsack optimal value

The large storage requirement of `kp_dp()` is due to the fact that, to permit the execution of `kp_dp_sol()`, all possible states are recorded in the array `v`. Suppose, however, we require only the optimal solution value to an instance of the knapsack problem, but not the solution itself. In this case there is no need to store the values of states that have already been used in the propagation process. Indeed, when calculating `v[i][j]` for a fixed i and varying j, we use only `v[i-1][j']` for different $j' \leq j$. This suggests the following approach to compute the optimal solution value using only an array of size $c + 1$, that we call `nv`.

```
int kp_dp_optval()
{
   int i, j, nv[c+1];

   for (j=0; j<w[0]; j++)  nv[j]=0;
   for (j=w[0]; j<=c; j++) nv[j]=p[0];

   for (i=1; i<n; i++) {
      for (j=c; j>=w[i]; j--) {
         nv[j] = MAX(nv[j], p[i]+nv[j-w[i]]);
      }
   }
   return nv[c];
}
```

Notice that the first two loops are essentially the same as in function `kp_dp()`; they compute the best way to pack the first item into knapsacks of capacities from 0 to c. The next loop differs in that it saves the optimal solution values for items up to i in the same array that contains the values for items up to $i - 1$; for any j, the entries of `nv` from `nv[j]` upto `nv[c]` contain the best way of packing i items, while the remaining entries of the array contain the best way of packing $i - 1$ items and are subsequently updated.

With this change, memory limitations cease to be an issue and we obtain the following computation times.

n/R	100	1000	10000
100	.002	.008	.061
1000	.065	.558	5.70
10000	5.47	58.18	$**$

It is interesting that for the same instances, where both functions terminate within our time limit, the second function takes less than half the time as the first function. Though the second function actually performs fewer computational steps than the first one (as is clear from the code), this is not enough to explain the difference. The primary explanation is the use of much less memory, a large part of which can fit into one of the levels of cache available to the computer processor.

It is clear from these tables that 0-1 knapsack instances with moderate sizes, having both the number of variables and their weights at most a few thousand, can be easily solved with a trivial dynamic-programming algorithm. Instances with 100,000 or more variables are beyond the reach of `kp_dp()`, but later in this chapter we will see that even such large uncorrelated instances can be solved by a more sophisticated dynamic-programming implementation, and, in Chapter 8, by simple branch-and-bound codes. In other words, these problems are surprisingly easy, though with the right techniques: a perfect lesson that one should not necessarily be daunted by $\mathcal{N}P$-hardness without trying out a few different ideas.

THE UNBOUNDED KNAPSACK PROBLEM

Our simple dynamic-programming code handles 0-1 instances of the knapsack problem only, but unbounded instances can by similarly solved in same $O(nc)$ time complexity. Indeed, consider an optimal packing of copies of items $0, \ldots, n-1$ into a knapsack of capacity c, where arbitrarily many copies of an item can be present in the packing. If item $n-1$ is not used, then the packing is an optimal packing of items $0, \ldots, n-2$ into the same knapsack. If a copy of item $n-1$ is used, then the remaining objects form an optimal packing of a knapsack of capacity $c - w_{n-1}$. However, the remaining objects can contain copies of item $n-1$. Therefore, to obtain an code for the unbounded knapsack problem, we can replace in `kp_dp()` the statement

```
v[i][j] = MAX(v[i-1][j], p[i]+v[i-1][j-w[i]]);
```

with the statement

```
v[i][j] = MAX(v[i-1][j], p[i]+v[i][j-w[i]]);
```

to correctly fill in the array `v`. Further, we need to compute the values `v[0][j]` for $j = 0, \ldots, c$ as follows:

```
for (j=0; j<=c; j++) v[0][j] = p[0]*(j/w[0]);
```

The bounded knapsack problem can also be solved by an $O(nc)$ algorithm [72], but we will not present the details here; dynamic-programming solution methods for bounded instances are much more involved than those for 0-1 instances or unbounded instances.

REMOVING DOMINATED STATES

The simplicity of the 0-1 algorithm makes it a good example to illustrate a number of strategies for improving the practical performance of dynamic programming in general. So let's see what we can do to speed up the computations. The primary target will be to reduce the total number of computed states, thereby reducing computing time and memory requirements. In this effort we will use the notion of removing *dominated states*, and also consider removing or pruning a state if it cannot be *extended* to an optimal solution. The algorithms developed here will be in closer in spirit to the dynamic-programming algorithms for the TSP discussed earlier, where we attempt to generate all paths involving a subset of vertices, but delete many of them by using upper bounds on the length of the best path through the remaining vertices.

Consider the *state table* of `v[i][j]` values for the example in Figure 7.4. The entries `v[0][3]`, `v[0][4]`, ..., `v[0][10]` are all equal to 7. In this case the optimal solution value when we pack the first item into a knapsack of capacity of 3 remains the same when we pack the item into a knapsack of capacity between 4 and 10. This means that after deciding how to pack items $1, \ldots, n-1$ into the original knapsack of capacity 10, if the remaining capacity is a number greater than or equal to 3 then an optimal way of packing the first item will be yielded by information from the state `v[0][3]`. Thus the states `v[0][4]`, ..., `v[0][10]` need not be saved. In general, if `v[i][j]` $=$ `v[i][k]`, where $k > j$, then the first state *dominates* the second, as it yields the same profit with less capacity usage. By this definition, out of the 66 states generated by `kp_dp()` for the example in Figure 7.4, a total of 22 are dominated by other states. Note that for an index i and integer $k \in [1, c]$, if no subset of the items $0, \ldots, i$ has weight equal to k, then the state `v[i][k]` will be dominated by a state `v[i][j]` where $0 \leq j < k$ and j equals the weight of some subset of the items $0, \ldots, i$. Therefore, each non-dominated state `v[i][j]` corresponds to a subset of items and the "capacity" j is equal to the weight of the items in the subset. Thus the number of non-dominated states after i items have been considered is at most 2^i, which can be considerably less than the capacity for small values of i.

To implement an algorithm that stores only non-dominated states, we start off with two states corresponding to the item 0, one where the item is used and the other where it is not. Subsequently, for $i = 1, \ldots, n-1$, we extend each existing state involving items $0, \ldots, i-1$ to two states involving items $0, \ldots, i$, one by adding item i and the other where we do not add item i. To prune new states, we sort them by increasing weight of the item subsets and then apply the domination criteria mentioned above. We apply this strategy in a function `kp_dp_states()`, presented below, that computes the optimal knapsack value only, like our earlier function `kp_dp_optval()`. We assume the following integer arrays are declared prior to invoking `kp_dp_states()`, in addition to the arrays containing the input data.

```
int sw[c+1], sp[c+1];
int nsw[2*(c+1)], nsp[2*(c+1)], perm[2*(c+1)];
```

The weight of state k is saved in `sw[k]` and its profit in `sp[k]`.

```
int kp_dp_states()
{
   int i, j, ns, newns;
   sw[0]=0; sw[1]=w[0];
   sp[0]=0; sp[1]=p[0];
   ns=2;
   for (i=1; i<n; i++){
      newns=0;
      for (j=0; j<ns; j++){
            nsw[newns]=sw[j];
            nsp[newns]=sp[j];
            newns++;
            if (sw[j]+w[i] <= c){
               nsw[newns]=sw[j]+w[i];
               nsp[newns]=sp[j]+p[i];
               newns++;
            }
         }
         ns=dom_prune (perm, newns, nsw, nsp, sw, sp);
   }
   return sp[ns-1];
}
```

At the start of the ith iteration of the outermost `for` loop, every state in `sw` corresponds to a subset of items $0, \ldots, i-1$ with weight at most c, the capacity, and therefore `sw` and `sp` need to have at most $c+1$ elements. The inner `for` loop computes two new states per state in `sw`, and saves their weights and profits in `nsw` and `nsp`; these arrays need at most $2(c+1)$ elements as a consequence. These states are passed to the function `dom_prune()`, which applies the domination criteria described above, copies back the undominated states to `sw` and `sp`, and returns the number of such states. The following code implements this function.

```
int dom_prune(int *perm, int ns, int *sw, int *sp,
              int *nsw, int *nsp)
{
   int j, k=0;
   for (j=0; j<ns; j++) perm[j]=j;
   int_perm_quicksort(perm, sw, ns);
   nsw[k] = sw[perm[0]];
   nsp[k] = sp[perm[0]];

   for (j=1; j<ns; j++){
      if (sp[perm[j]] > nsp[k]){
         if (sw[perm[j]] > nsw[k]) k++;
         nsw[k]=sw[perm[j]];
         nsp[k]=sp[perm[j]];
      }
   }
   return k+1;
}
```

The first two lines of code sort the input states, given in `sw` and `sp`, by increasing weight; more precisely, the array `perm` contains the permutation of states that yields a sorted order. The second `for` loop copies the states to the ‘new’ state weight and profit arrays, removing dominated states.

We compare below this new code with `kp_dp_optval()` on uncorrelated instances having 100 items and different values of R . Each number is the running time in seconds averaged over ten instances.

$R\ (n = 100)$	10000	100000	1000000
`kp_dp_optval()`	.061	.588	5.99
`kp_dp_states()`	.009	.010	.011

For $R = 10{,}000$, the basic code is already considerably slower, taking .061 seconds instead of the hundredth of a second taken by `kp_dp_states()`. When R is a million, the first algorithm is almost 600 times slower. The difference in times is almost entirely due to the small number of non-dominated states generated by `kp_dp_states()`; this number is about $57{,}000$ when $R = 10{,}000$, and 58,000 and 63,000 for the next two values of R. When R is a million, the total number of states processed by `kp_dp_optval()` equals $100(c + 1)$ and c is close to 25 million.

Removing non-extendable states

The total number of states removed by the above process is often impressive, but many of the remaining non-dominated states are also not actually needed in the computation, since these states are impossible to extend to optimal packings of the knapsack. Pruning at least some of these non-extendable states can further improve the running time of the algorithm. To accomplish this we adopt bounding ideas as in our TSP algorithms. To begin, we need one reasonably good knapsack solution, in order to provide a lower bound on the optimal solution value. An easy choice is the *greedy solution*, obtained by first sorting items by decreasing *efficiency*, that is, profit to weight ratio, and then iterating through the items from $0, \ldots, n-1$, adding an item to the knapsack if it fits into the remaining capacity and discarding it otherwise. Code to produce this solution is given in the following `kp_greedy()` function, when it is invoked with `beg=0`.

```
int kp_greedy(int beg, int so_w[], int so_p[], int c)
{
   int i, twt=0, tpt=0;
   for (i=beg; i<n && twt<c; i++){
      if (twt + so_w[i] <= c){
         twt += so_w[i];
         tpt += so_p[i];
      }
   }
   return tpt;
}
```

We assume the arrays `so_w[]` and `so_p[]` give the weights and profits of the items after sorting by decreasing efficiency, and `c` is the capacity. We will see later that `kp_greedy()` can be usefully invoked with `beg` greater than 0, and `c` less than the capacity.

Suppose the weight of the greedy solution is l. Consider a state s, having weight w_s and profit p_s, corresponding to a subset of items $0, \ldots, i$. If an upper bound on the best possible solution that includes the subset of items corresponding to state s is at most l, then s cannot be extended to a solution better than the greedy solution. Such an upper bound can be obtained by adding p_s to any upper bound on the value of a best packing of items $i + 1, \ldots, n - 1$ into a knapsack of capacity $c - w_s$. The optimal solution value of this $c - w_s$ instance is bounded above by the optimal solution value of the unbounded knapsack problem on the same items with capacity $c - w_s$. This in turn can be bounded above by solving a *fractional unbounded knapsack problem*, where fractional quantities of each item can be chosen. This latter problem is easy to solve: simply take the item with maximum profit per unit weight consumed, that is, maximum efficiency, and fill the knapsack with fractionally many copies of this item. In other words, if item $k \in [i + 1, n - 1]$ has the greatest value of $p[k]/w[k]$, then the optimal solution value is $(c - w_s) * p[k]/w[k]$, and the floor of this value yields an upper bound on the optimal solution value of the 0-1 knapsack problem with items $i + 1$ to $n - 1$.

This leads to the following easy modification of `kp_dp_states()`. We assume the efficiencies of the knapsack items are computed and saved in an array `eff[n+1]`, as in the following code fragment.

```
int i;
double eff[n+1], maxeff[n+1];

eff[n] = maxeff[n] = 0.0;
for (i=n-1; i>=0; i--){
   eff[i] = ((double)p[i])/w[i];
   maxeff[i] = MAX(maxeff[i+1], eff[i]);
}
```

The ith entry of `maxeff` stores the maximum efficiency of items $i, \ldots, n - 1$; we will see in a moment why it is convenient to store the value 0 in each of `eff[n]` and `maxeff[n]`. We also assume the greedy solution value is stored in `lb`. We replace the inner `for` loop in `kp_dp_states()` by

```
for (j=0; j<ns; j++){
   if (sp[j] + floor(maxeff[i+1]*(c-sw[j])) > lb){
      nsw[newns]=sw[j];
      nsp[newns]=sp[j];
      newns++;
   }
   if (sw[j] + w[i] <= c){
      if (sp[j]+p[i] +
          floor(maxeff[i+1]*(c-sw[j]-w[i])) > lb){
         nsw[newns]=sw[j]+w[i];
         nsp[newns]=sp[j]+p[i];
         newns++;
```

```
        }
    }
}
ns=0;
if (newns == 0) break;
```

When we exit the outermost `for` loop, it could be because all states have been pruned after processing fewer than n items, in which case we have verified that no solution better than `lb` exists. We therefore replace the statement

```
return sp[ns-1];
```

in `kp_dp_states()` by the statement

```
if (ns > 0) return sp[ns-1]; else return lb;
```

We call this modified code `kp_dp_statesbnd()`. In the next table, we compare the number of states, rounded to the nearest 100, generated by `kp_dp_states()` and by `kp_dp_statesbnd()`.

R $(n = 100)$	10000	100000	1000000
`kp_dp_states`	57100	58000	63000
`kp_dp_statesbnd`	54200	56000	59700

The results are somewhat disappointing in that the modified code generates only 5% fewer states. On closer examination, we see that the simple upper bounds used in `kp_dp_statesbnd()` can be weak. For example, if item $n - 1$ has maximum efficiency among all items, then it will always be used in solving the fractional unbounded knapsack subproblem associated with the states. In particular, if a state at level i (after processing i items) and another at level $i+1$ have the same profit and weight, the upper bound on the best solution the states can be extended to does not change. On the other hand, if the items were a priori sorted in order of decreasing efficiency before invoking `kp_dp_statesbnd()`, then the maximum efficiency of items $i, \dots, n-1$ is greater than or equal to that of items $i + 1, \dots, n - 1$, and we get tighter upper bounds as we process more items. We call this modified function `kp_dp_statesbnd2()`. Note that in this case, `maxeff[i]=eff[i]`. This simple change results in a dramatic improvement.

R $(n = 100)$	10000	100000	1000000
`kp_dp_statesbnd2`	12300	13700	13800

On average, for the specific instances in the table, we generate less than a fourth of the states generated by `kp_dp_statesbnd()`. This illustrates clearly the importance of organizing computation in a way that upper bounds on the optimal solution value improve as the algorithm progresses.

Running times in seconds for the new code are reported in the following table.

R $(n = 100)$	10000	100000	1000000
`kp_dp_statesbnd2()`	.004	.004	.005

For R = 1,000,000 and n = 100, our tests put `kp_dp_statesbnd2()` at one thousand times faster than `kp_dp_optval()`. That is great, but the new code does not always win the competition. Indeed, when n = 1,000 or 10,000 and R = 100, the new code is five or six times slower than the original code. This discrepancy can be explained as follows. When the ratio of the number of processed items to capacity is high, then typically the number of stored states becomes fairly large and approaches the capacity. In such cases a large amount of time is then spent sorting the states; in fact, in the above two cases where `kp_dp_optval()` is faster, approximately 80% of the time is spent in sorting. In the worst case, the new code has running-time complexity $O(nc\, logc)$ for instances with n items and capacity c, which is quite a bit worse than the $O(nc)$ running time of `kp_dp_optval()`; when n = 10,000 and R = 100, then $c \approx$ 250,000 and thus $\log c \approx 18$.

It is important to note that, whereas `kp_dp_optval()` performs similarly for uncorrelated instances and strongly correlated instances with the same n and R values, the performance of `kp_dp_statesbnd2()` changes dramatically with problem type. In the next table we report the number of states and running time for `kp_dp_statesbnd2()` for both types of instances for different values of R, and for n fixed to 100; here m stands for million.

	uncorrelated			strongly correlated		
R (n = 100)	10000	100000	1m	10000	100000	1m
time	.004	.004	.005	2.68	36.9	423
states	12300	13700	13800	11m	108m	845m

When R = 1,000,000 and n = 100, for a strongly correlated instance the running time for `kp_dp_statesbnd2()` its almost 100,000 times its running time for an uncorrelated instance. Furthermore, while the number of states and running time increases slowly with R for the uncorrelated instances, the increase is much more rapid for the strongly correlated examples. This is a common feature of $\mathcal{N}P$-hard problems; a technique that performs well on one class of instances often performs very poorly on a different class, and one may need to try out a number of different approaches to solving a given class of instances.

Let's try to squeeze out a bit more performance on the strongly correlated examples. Notice that the lower bound on the optimal solution value does not change in `kp_dp_statesbnd2()` as the algorithm progresses. This can be adjusted by using `kp_greedy` to extend the partial solution represented by a state. For example, consider a non-dominated state s at the end of level i, with weight w_s and profit p_s. Then

```
p_s + kp_greedy(i+1, so_w, so_p, c-w_s)
```

is the value of a knapsack solution obtained by taking the items in s and adding greedily the remaining items in the remaining capacity $c - w_s$. This computation is done in the following fragment of code, added just after the invocation of `dom_prune()` in `kp_dp_statesbnd2()`.

```
for (j=ns-1; j>=0; j -= 1){
   double tlb = (sw[j] < c) ?
   sp[j] + kp_greedy (i+1, w, p, c-sw[j]) : sp[j];
   if (tlb > lb) lb = tlb;
}
```

This additional code is not very useful for uncorrelated instances, since the lower bound from the first invocation of `kp_greedy` is often very close to the optimal solution value. However, it seems useful for the strongly correlated instances, and we now compare the running time of `kp_dp_statesbnd2()` for these instances with the variant where we repeatedly invoke `kp_greedy()`.

R $(n = 100)$	10000	100000	1000000
`kp_dp_statesbnd2()`	2.68	36.9	423
`kp_dp_statesbnd2()` + `kp_greedy()`	.709	8.65	49.9
`kp_dp_statesbnd3()`	.482	6.66	39.9

We get an improvement of nearly a factor of ten on these instances. This combined code is, however, much slower for larger values of n. For example, when $n = 1{,}000$ and $R = 100$, it takes 4.16 seconds versus only .659 seconds for `kp_dp_statesbnd2()`, even though it generates fewer states. This is because we perform $O(n)$ computations in `kp_greedy()` for every state. If we simply decrement j by, say, 20 in the `for` loop fragment above instead of 1, that is, replace `j -= 1` by `j -= 20`, then we reduce time spent on generating heuristic solutions, but still increase `lb` enough to reduce overall computing time for $n = 100$; see the last row in the previous table. We call this variant `kp_dp_statesbnd3()`; it is also now comparable to `kp_dp_statesbnd2()` when $n = 1{,}000$ and $R = 100$. However, for $n = 1{,}000$ or more, it is still slower than `kp_dp_optval()`.

A NEIGHBORHOOD-SEARCH ALGORITHM

The final dynamic-programming code we discuss differs from the previous ones in important ways. It can be viewed as a *very-large-neighborhood search* algorithm, and is based on the MINKNAP algorithm of Pisinger [73]. Instead of beginning with an empty knapsack, and generating states by adding items, the algorithm starts off with a state corresponding to a good initial knapsack solution, and then explores states/solutions generated, in an iterative fashion, by adding items not present in the solution and *removing* items currently in the solution, thus in the *neighborhood* of the first solution. As a state can be extended by removing items, we need to consider states with weight more than the capacity c. However, as we wish to extend any state to a final state with weight c or less, and we can only remove items present in the original solution (which has weight at most c), it suffices to consider intermediate states with weight at most $2c$. Dominated states are eliminated using `dom_prune()`, and upper bounds on the best optimal solution a state can be extended to are obtained as in `kp_dp_statesbnd2()` via an associated fractional unbounded knapsack problem, though in a more subtle fashion. A crucial aspect of this algorithm is that any state that has weight at most c and profit more than the initial solution value yields a better solution, and thus an improved lower bound that is used in pruning states. This lower bound is obtained at essentially no additional cost, unlike in `kp_dp_statesbnd3()`.

An implementation of the above neighborhood-search algorithm is given in function `kp_dp_nbsearch()` below. Just as organizing items in a certain manner (by decreasing efficiency) in `kp_dp_statesbnd2()` allowed easy computation of upper bounds, we will use a particular heuristic solution called the *critical* solution to allow easy upper-bound computation. The justification for this solution will be given only later when we discuss the use of LP-duality information in the branch-and-bound chapter. We will assume items are sorted by decreasing efficiency as in the function `kp_dp_statesbnd2()`. The *critical* item, denoted by i_c, is simply the first item (after sorting by decreasing efficiency) such that the weight of the item and previous items exceeds the knapsack capacity, and the critical solution, denoted as sol_c, consists of all items preceding the critical item. The following function generates the critical solution and returns its weight and profit.

```
void heuristic_crit(int *critical, int *cwt, int *cpt)
{
   int i;
   *cwt = *cpt = 0;
   for (i=0; i<n && *cwt+w[i]<=c; i++){
      *cwt += w[i];
      *cpt += p[i];
   }
   *critical=i;
}
```

Notice that the items in sol_c are consecutive items, and so are the items not present in the solution.

The first state in `kp_dp_nbsearch()` has the weight and profit of sol_c, that is, we assume all items in sol_c must be present, or are *fixed to 1*, and all other items must be absent, or are *fixed to 0*. At level 0, we “unfix” item $i_c + 1$, that is, all subsequent items must be absent, and all items preceding i_c must be present, but i_c can be present or absent. Now the first state had i_c absent, so we create two new states, one with i_c absent (thus the profit and weight are unchanged), and another where we add i_c; the weight will exceed the capacity c (as i_c is the critical item). At level 1, we unfix item $i_c - 1$; thus for each previous states ($i_c - 1$ is present in them), we create two states, one where we retain $i_c - 1$ and another where we remove i_c (and decrement profit and weight values). At level 2, we unfix item $i_c + 2$, and so on. We use variables `s` and `t` to indicate the range of unfixed variables (they are consecutive). When $s = 0$ and $t = n - 1$, we have solved the problem. We assume the following arrays are declared globally in addition to the arrays containing the problem data.

```
int sw[2*(c+1)], sp[2*(c+1)];
int nsw[4*(c+1)], nsp[4*(c+1)], perm[4*(c+1)];
double eff[n+1];
```

Further, we assume that `eff[i]` contains the efficiency of item i, and `eff[n]` contains 0.0.

```
int kp_dp_nbsearch()
{
   int i, j, s, t, twt, tpt;
   int ns, newns, critical, lb;
   double effs1;

   heuristic_crit (&critical, &(sw[0]), &(sp[0]));
   lb = sp[0];
   ns = 1;
   s = critical;
   t = critical-1;

   for (i=0; i<n; i++){
      if (i%2 == 0) t++; else s--;
      newns = 0;
      effs1 = (s > 0) ? eff[s-1] : 0.0;

      for (j=0; j<ns; j++){
         if ((sw[j] <= c &&
              sp[j] + floor(eff[t+1]*(c-sw[j])) > lb) ||
             (sw[j] > c &&
              sp[j] - ceil(effs1*(sw[j]-c)) > lb)) {
            nsw[newns] = sw[j];
            nsp[newns] = sp[j];
            newns++;
         }

         twt = (i%2 == 0) ? sw[j] + w[t] : sw[j] - w[s];
         tpt = (i%2 == 0) ? sp[j] + p[t] : sp[j] - p[s];
         if (twt <= 2*c && twt >= 0){
            if ((twt <= c &&
                 tpt + floor(eff[t+1]*(c-twt)) > lb) ||
                (twt > c &&
                 tpt - ceil(effs1*(twt-c)) > lb)){
               nsw[newns] = twt;
               nsp[newns] = tpt;
               newns++;
            }
         }
      }
      if (newns == 0) break;
      ns = dom_prune (perm, newns, nsw, nsp, sw, sp);
```

```
        for (j=0; j<ns && sw[j]<=c; j++){
            if (sp[j] > lb) lb = sp[j];
        }
    }
    return lb;
}
```

We report running times for `kp_dp_nbsearch()` in the following table.

	uncorrelated			strongly correlated		
n/R	100	1000	10000	100	1000	10000
100	.002	.002	.002	.002	.002	.076
1000	.002	.002	.003	.019	.078	.172
10000	.004	.005	.010	.050	1.19	25.2

This code is dramatically faster than the prior codes on all instances. On the uncorrelated instances, it is super fast! Indeed, for uncorrelated instances having $R = 10{,}000$ and $n = 1{,}000{,}000$, the running time is only .439 seconds. Such large instances are unsolvable with our earlier codes. Moreover, in later chapters we will see how to speed up this new code even a bit more using LP-duality information.

The development of the algorithms up to `kp_dp_statesbnd3()` shows again that simple ideas can significantly speed up a code. In fact, many of the ideas we discussed here are common ingredients in fast algorithms for other problem classes. However, there is no obvious path leading to the development of an algorithm like `kp_nbsearch()`, which involves a conceptual leap from previous methods. The discovery of the MINKNAP algorithm and its effectiveness by Pisinger was preceded and motivated by a substantial body of research into solution methods for knapsack problems.

7.4 EXERCISES

1. Explain the operation of "Gosper's hack" used in Section 7.1.

2. Write a function to fill a two-dimensional array `binomial[][]` with the binomial coefficients $\binom{m}{k}$ for m and k ranging from 0 up to $n-1$, setting an entry to 0 if $m < k$.

3. Show that `Sval` matches the ordering given by `nextset()` in the code fragments presented in Section 7.1.

4. When the Bellman-Held-Karp algorithm is used to solve a symmetric instance of the TSP, it suffices to compute the values of $opt(S,t)$ for sets S such that $S <= \lceil n/2 \rceil$. Modify the implementation of the algorithm to take advantage of this fact.

5. *Floyd-Warshall Algorithm.* Let $G = (V, E)$ be a directed graph with nodes $V = \{0, 1, \ldots, n-1\}$ and edge costs $c = (c_e : e \in E)$. For a path P from node

i to node j, all nodes in P other than i and j are called *internal*. Letting $d(i, j, k)$ denote the length of the shortest directed path P from i to j such that all internal nodes of P are in the set $\{0, 1, \ldots, k-1\}$, we have the recursive equation

$$d(i, j, k) = min(d(i, j, k-1),\ d(i, k-1, k-1) + d(k-1, j, k-1)).$$

Use this to design and implement an $O(n^3)$ dynamic-programming algorithm for the all-pairs shortest-path problem of finding a shortest directed path between each pair of nodes.

6. Modify the code in `kp_dp_optval()` to compute the optimal solution value for an instance of the unbounded knapsack problem using an integer array of size $c + 1$.

7. Write a code which solves an instance of the bounded knapsack problem using an integer array of size $n(c + 1)$ as in function `kp_dp()`, and has time complexity $O(knc)$ where k is the largest bound on the different items.

7.5 NOTES AND REFERENCES

SECTION 7.1

The running-time bound of $O(n^2 2^n)$ is better than checking all tours, but it would be disappointing if Bellman-Held-Karp is the best we can do. In looking to beat the record, one needs to focus on the 2^n term: replacing $n^2 2^n$ by $n 2^n$ would not be considered an important step. A nice reference for this type of work is Gerhard Woeginger's survey paper [82] on exact algorithms for $\mathcal{N}P$-hard problems.

SECTION 7.2

Bellman created a large body of work in dynamic programming, including his 1957 book *Dynamic Programming* [5], a second book covering connections with control theory *Adaptive Control Processes: A Guided Tour* [7], and together with S. E. Dreyfus the book *Applied Dynamic Programming* [9].

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. Vušković. 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. Vušković. 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 **10**, 196–210.

[44] Held, M., R. M. Karp. 1970. The traveling-salesman problem and minimum spanning trees. Operations Research **18**, 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 **42**, 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 **10**, 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 **29**, 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.