be the root of an optimal subtree containing $c_i, c_{i+1}, \ldots, c_j$. Write also $r_{i,i-1} = i$. Prove that $r_{i,j-1} \le r_{ij} \le r_{i+1,j}$ for every $1 \le i \le j \le n$. □

Problem 5.5.11. Use the result of Problem 5.5.10 to show how to calculate an optimal search tree in a time in $O(n^2)$. (Problems 5.5.10 and 5.5.11 generalize to the case discussed in Problem 5.5.9.) □

Problem 5.5.12. There is an obvious greedy approach to the problem of constructing an optimal search tree: place the most probable key, c_k, say, at the root, and construct the left- and right-hand subtrees for $c_1, c_2, \ldots, c_{k-1}$ and $c_{k+1}, c_{k+2}, \ldots, c_n$ recursively in the same way.

i. How much time does this algorithm take in the worst case, assuming the keys are already sorted?

ii. Show with the help of a simple, explicit example that this greedy algorithm does not always find the optimal search tree. Give an optimal search tree for your example, and calculate the average number of comparisons needed to find a key for both the optimal tree and the tree found by the greedy algorithm. □

5.6 THE TRAVELLING SALESPERSON PROBLEM

We have already met this problem in Section 3.4.2. Given a graph with nonnegative lengths attached to the edges, we are required to find the shortest possible circuit that begins and ends at the same node, after having gone exactly once through each of the other nodes.

Let $G = \langle N, A \rangle$ be a directed graph. As usual, we take $N = \{1, 2, \ldots, n\}$, and the lengths of the edges are denoted by L_{ij}, with $L[i,i] = 0$, $L[i,j] \ge 0$ if $i \ne j$, and $L[i,j] = \infty$ if the edge (i,j) does not exist.

Suppose without loss of generality that the circuit begins and ends at node 1. It therefore consists of an edge $(1,j)$, $j \ne 1$, followed by a path from j to 1 that passes exactly once through each node in $N \setminus \{1, j\}$. If the circuit is optimal (as short as possible), then so is the path from j to 1: the principle of optimality holds.

Consider a set of nodes $S \subseteq N \setminus \{1\}$ and a node $i \in N \setminus S$, with $i = 1$ allowed only if $S = N \setminus \{1\}$. Define $g(i, S)$ as the length of the shortest path from node i to node 1 that passes exactly once through each node in S. Using this definition, $g(1, N \setminus \{1\})$ is the length of an optimal circuit. By the principle of optimality, we see that

$$g(1, N \setminus \{1\}) = \min_{2 \le j \le n} (L_{1j} + g(j, N \setminus \{1, j\})). \qquad (*)$$

More generally, if $i \ne 1$, $S \ne \varnothing$, $S \ne N \setminus \{1\}$, and $i \notin S$,

$$g(i, S) = \min_{j \in S} (L_{ij} + g(j, S \setminus \{j\})). \qquad (**)$$

Furthermore,

$$g(i, \varnothing) = L_{i1}, \quad i = 2, 3, \ldots, n \ .$$

The values of $g(i, S)$ are therefore known when S is empty. We can apply (**) to calculate the function g for all the sets S that contain exactly one node (other than 1); then we can apply (**) again to calculate g for all the sets S that contain two nodes (other than 1), and so on. Once the value of $g(j, N \setminus \{1, j\})$ is known for all the nodes j except node 1, we can use (*) to calculate $g(1, N \setminus \{1\})$ and solve the problem.

Example 5.6.1. Let G be the complete graph on four nodes given in Figure 5.6.1:

$$L = \begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix} .$$

We initialize

$$g(2, \varnothing) = 5, \quad g(3, \varnothing) = 6, \quad g(4, \varnothing) = 8 \ .$$

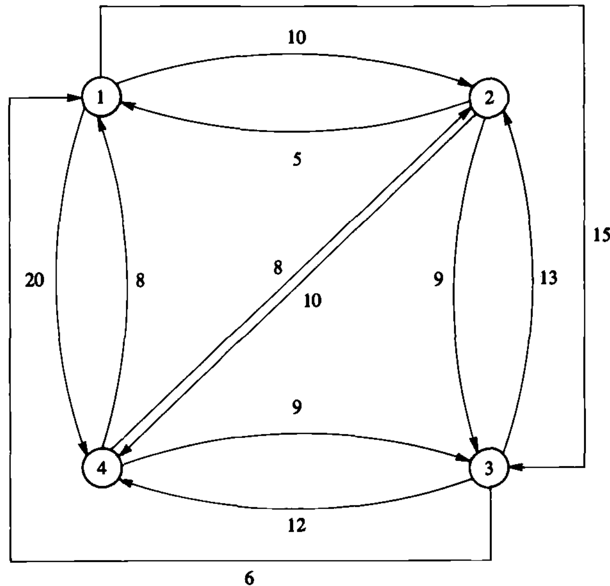

Figure 5.6.1. A directed graph for the travelling salesperson problem.

Using (**), we obtain

$$g(2,\{3\}) = L_{23} + g(3,\varnothing) = 15$$
$$g(2,\{4\}) = L_{24} + g(4,\varnothing) = 18$$

and similarly

$$g(3,\{2\}) = 18, \quad g(3,\{4\}) = 20$$
$$g(4,\{2\}) = 13, \quad g(4,\{3\}) = 15.$$

Next, using (**) for sets of two nodes, we have

$$\begin{aligned} g(2,\{3,4\}) &= \min(L_{23} + g(3,\{4\}), L_{24} + g(4,\{3\})) \\ &= \min(29, 25) = 25 \end{aligned}$$

$$\begin{aligned} g(3,\{2,4\}) &= \min(L_{32} + g(2,\{4\}), L_{34} + g(4,\{2\})) \\ &= \min(31, 25) = 25 \end{aligned}$$

$$\begin{aligned} g(4,\{2,3\}) &= \min(L_{42} + g(2,\{3\}), L_{43} + g(3,\{2\})) \\ &= \min(23, 27) = 23 . \end{aligned}$$

Finally we apply (*) to obtain

$$\begin{aligned} g(1,\{2,3,4\}) &= \min(L_{12} + g(2,\{3,4\}), L_{13} + g(3,\{2,4\}), L_{14} + g(4,\{2,3\})) \\ &= \min(35, 40, 43) = 35 . \end{aligned}$$

The optimal circuit in Figure 5.6.1 has length 35. □

To know where this circuit goes, we need an extra function: $J(i, S)$ is the value of j chosen to minimize g at the moment when we apply (*) or (**) to calculate $g(i, S)$.

Example 5.6.2. (Continuation of Example 5.6.1.) In this example we find

$$\begin{aligned} J(2,\{3,4\}) &= 4 \\ J(3,\{2,4\}) &= 4 \\ J(4,\{2,3\}) &= 2 \\ J(1,\{2,3,4\}) &= 2 \end{aligned}$$

and the optimal circuit is

$$\begin{aligned} 1 &\to J(1,\{2,3,4\}) = 2 \\ &\to J(2,\{3,4\}) = 4 \\ &\to J(4,\{3\}) = 3 \\ &\to 1 \quad . \end{aligned}$$

□

The required computation time can be calculated as follows:

— to calculate $g(j,\varnothing)$: $n-1$ consultations of a table;

— to calculate all the $g(i,S)$ such that $1 \le \#S = k \le n-2$: $(n-1)\binom{n-2}{k}k$ additions in all;

— to calculate $g(1, N \setminus \{1\})$: $n-1$ additions.

These operations can be used as a barometer. The computation time is thus in

$$\Theta\Big(2(n-1) + \sum_{k=1}^{n-2}(n-1)k\binom{n-2}{k}\Big) = \Theta(n^2 2^n) \quad \text{since} \quad \sum_{k=1}^{r} k\binom{r}{k} = r\,2^{r-1}.$$

This is considerably better than having a time in $\Omega(n!)$, as would be the case if we simply tried all the possible circuits, but it is still far from offering a practical algorithm. What is more ...

Problem 5.6.1. Verify that the space required to hold the values of g and J is in $\Omega(n\,2^n)$, which is not very practical either. □

TABLE 5.6.1. SOLVING THE TRAVELLING SALESPERSON PROBLEM.

n	Time: Direct method $n!$	Time: Dynamic programming $n^2 2^n$	Space: Dynamic programming $n\,2^n$
5	120	800	160
10	3,628,800	102,400	10,240
15	1.31×10^{12}	7,372,800	491,520
20	2.43×10^{18}	419,430,400	20,971,520

Problem 5.6.2. The preceding analysis assumes that we can find in constant time a value of $g(j,S)$ that has already been calculated. Since S is a set, which data structure do you suggest to hold the values of g ? With your suggested structure, how much time is needed to access one of the values of g ? □

Table 5.6.1 illustrates the dramatic increase in the time and space necessary as n goes up. For instance, $20^2 2^{20}$ microseconds is less than 7 minutes, whereas 20! microseconds exceeds 77 thousand years.

5.7 MEMORY FUNCTIONS

If we want to implement the method of Section 5.6 on a computer, it is easy to write a function that calculates g recursively. For example, consider

```
function g(i,S)
  if S = ∅ then return L[i,1]
  ans ← ∞
  for each j ∈ S do
    distviaj ← L[i,j] + g(j, S \ {j})
    if distviaj < ans then ans ← distviaj
  return ans .
```

Unfortunately, if we calculate g in this top-down way, we come up once more against the problem outlined at the beginning of this chapter: most values of g are recalculated many times and the program is very inefficient. (In fact, it ends up back in $\Omega((n-1)!)$.)

So how can we calculate g in the bottom-up way that characterizes dynamic programming? We need an auxiliary program that generates first the empty set, then all the sets containing just one element from $N \setminus \{1\}$, then all the sets containing two elements from $N \setminus \{1\}$, and so on. Although it is maybe not too hard to write such a generator, it is not immediately obvious how to set about it.

One easy way to take advantage of the simplicity of a recursive formulation without losing the efficiency offered by dynamic programming is to use a *memory function*. To the recursive function we add a table of the necessary size. Initially, all the entries in this table hold a special value to indicate that they have not yet been calculated. Thereafter, whenever we call the function, we first look in the table to see whether it has already been evaluated with the same set of parameters. If so, we return the value held in the table. If not, we go ahead and calculate the function. Before returning the calculated value, however, we save it at the appropriate place in the table. In this way it is never necessary to calculate the function twice for the same values of its parameters.

For the algorithm of Section 5.6 let *gtab* be a table all of whose entries are initialized to -1 (since a distance cannot be negative). Formulated in the following way:

```
function g(i,S)
  if S = ∅ then return L[i,1]
  if gtab[i,S] ≥ 0 then return gtab[i,S]
  ans ← ∞
  for each j ∈ S do
    distviaj ← L[i,j] + g(j, S \ {j})
    if distviaj < ans then ans ← distviaj
  gtab[i,S] ← ans
  return ans ,
```

the function g combines the clarity obtained from a recursive formulation and the efficiency of dynamic programming.

Problem 5.7.1. Show how to calculate (i) a binomial coefficient and (ii) the function *series* (n, p) of Section 5.2 using a memory function. □

We sometimes have to pay a price for using this technique. We saw in Section 5.1, for instance, that we can calculate a binomial coefficient $\binom{n}{k}$ using a time in $O(nk)$ and space in $O(k)$. Implemented using a memory function, the calculation takes the same amount of time but needs space in $\Omega(nk)$.

* **Problem 5.7.2.** If we are willing to use a little more space (the space needed is only multiplied by a constant factor, however), it is possible to avoid the initialization time needed to set all the entries of the table to some special value. This is particularly desirable when in fact only a few values of the function are to be calculated, but we do not know in advance which ones. (For an example, see Section 6.6.2.) Show how an array $T[1..n]$ can be *virtually initialized* with the help of two auxiliary arrays $B[1..n]$ and $P[1..n]$ and a few pointers. You should write three algorithms.

```
procedure init
  { virtually initializes T[1..n] }

procedure store(i, v)
  { sets T[i] to the value v }

function val(i)
  { returns the last value given to T[i], if any;
    returns a default value (such as -1) otherwise }
```

A call on any of these procedures or functions (including a call on *init* !) should take constant time in the worst case. □

5.8 SUPPLEMENTARY PROBLEMS

* **Problem 5.8.1.** Let u and v be two strings of characters. We want to transform u into v with the smallest possible number of operations of the following types:

- delete a character,
- add a character,
- change a character.

For instance, we can transform *abbac* into *abcbc* in three stages.

$$\begin{aligned} abbac &\to abac && \text{(delete } b\text{)} \\ &\to ababc && \text{(add } b\text{)} \\ &\to abcbc && \text{(change } a \text{ into } c\text{).} \end{aligned}$$

Show that this transformation is not optimal.